Método de Jacobi para resolver sistemas de ecuaciones lineales

Jonatan Enrique Badillo Tejeda y Javier Dolores Tolentino

May 7, 2024

Abstract

Este documento presenta una explicación del método de Jacobi para resolver sistemas de ecuaciones lineales, junto con dos implementaciones(secuencial y paralela) de este método en Java.

1 Explicación del método de Jacobi

El método de Jacobi es un algoritmo iterativo para resolver sistemas de ecuaciones lineales de la forma $Ax = b$ donde A es una matriz de coeficientes, x es el vector de incógnitas y b es el vector de términos independientes.

El método de Jacobi descompone la matriz A en una matriz diagonal D y el resto R, de modo que $A = D + R$. Luego, se resuelve iterativamente la ecuación $Dx = b - Rx$ hasta que la solución converja.

Para cada iteración k y para cada elemento i del vector x, se calcula:
$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)} \right)$

Donde a_{ij} son los elementos de la matriz A, b_i son los elementos del vector b, y $x_j^{(k-1)}$ son los elementos del vector x en la iteración anterior.

El algoritmo se detiene cuando la norma del error (la suma de las diferencias absolutas entre las soluciones actuales y las de la iteración anterior) es menor que un criterio de convergencia predefinido:
$\sum_{i=1}^{n} |x_i^{(k)} - x_i^{(k-1)}| < \epsilon$

Donde ϵ es el criterio de convergencia.

2 Implementación en Java

```
public class JacobiSecuencial {
    public static void main(String[] args) {
        // Define tu matriz A y el vector B aquí
        double[][] A = {
            {4, -1, 0, 0},
            {-1, 4, -1, 0},
```

```
            {0, -1, 4, -1},
            {0, 0, -1, 3}
        };

        double[] B = {15, 10, 10, 10};
        double[] X = new double[B.length]; // Vector de soluciones iniciales, inicializado a
        double[] lastX = new double[B.length]; // Vector para almacenar la solución de la it
        double error = 1e-10; // Criterio de convergencia

        while (true) {
            for (int i = 0; i < A.length; i++) {
                double sum = B[i]; // Inicializa la suma con el elemento correspondiente del
                for (int j = 0; j < A[i].length; j++) {
                    if (j != i) sum -= A[i][j] * lastX[j]; // Resta los productos Aij*Xj par
                }
                X[i] = 1/A[i][i] * sum; // Divide por el coeficiente diagonal Aii
            }

            // Calcula la norma del error como la suma de las diferencias absolutas entre la
            double errorNorm = 0;
            for (int i = 0; i < X.length; i++) {
                errorNorm += Math.abs(X[i] - lastX[i]);
                lastX[i] = X[i]; // Actualiza lastX con la solución actual para la próxima i
            }

            // Si la norma del error es menor que el criterio de convergencia, termina el bu
            if (errorNorm < error) break;
        }

        // Imprime el resultado
        for (double x : X) System.out.println(x);
    }
}
```

3 Explicación del código Java paralelo

Este código implementa el método de Jacobi en Java utilizando la biblioteca ForkJoinPool para paralelizar las operaciones. La matriz A y el vector b se definen al principio del código. El vector X se inicializa a 0 y se usa para almacenar las soluciones actuales, mientras que `lastX` se usa para almacenar las soluciones de la iteración anterior. El criterio de convergencia ϵ se establece en 1×10^{-10}.

El bucle `while` principal implementa las iteraciones del método de Jacobi. Para cada elemento i del vector x, se calcula la suma de b_i y los productos $A_{ij}X_j$ para $j \neq i$ (esta es la parte `sum -= A[i][j] * lastX[j]` en el código). Luego,

se divide esta suma por el coeficiente diagonal A_{ii} para obtener la solución actual X_i.

Después de calcular todas las soluciones actuales, se calcula la norma del error como la suma de las diferencias absolutas entre las soluciones actuales y las de la iteración anterior. Si la norma del error es menor que el criterio de convergencia, se termina el bucle.

Finalmente, se imprime el resultado, que es el vector de soluciones x.

La principal diferencia entre este código y el código secuencial anterior es que este código utiliza la biblioteca ForkJoinPool para paralelizar las operaciones. Esto puede resultar en un rendimiento significativamente mejorado para problemas grandes, ya que varias operaciones pueden realizarse simultáneamente en diferentes núcleos de la CPU.

```
import java.util.concurrent.*;

public class JacobiParalelo {
    static final ForkJoinPool fjPool = new ForkJoinPool();

    static class JacobiTask extends RecursiveAction {
        final double[][] A;
        final double[] B;
        final double[] X;
        final double[] lastX;
        final int start;
        final int end;
        final double error;

        JacobiTask(double[][] A, double[] B, double[] X, double[] lastX, int start, int
            this.A = A;
            this.B = B;
            this.X = X;
            this.lastX = lastX;
            this.start = start;
            this.end = end;
            this.error = error;
        }

        protected void compute() {
            if (end - start <= 500) { // Ajusta este valor según el tamaño de tu problem
                for (int i = start; i < end; i++) {
                    double sum = B[i];
                    for (int j = 0; j < A[i].length; j++) {
                        if (j != i) sum -= A[i][j] * lastX[j];
                    }
                    X[i] = 1/A[i][i] * sum;
                }
```

```
            } else {
                int mid = (start + end) / 2;
                invokeAll(new JacobiTask(A, B, X, lastX, start, mid, error),
                          new JacobiTask(A, B, X, lastX, mid, end, error));
            }
        }
    }

    public static void main(String[] args) {
        // Define tu matriz A y el vector B aquí
        double[][] A = {{...}};
        double[] B = {...};
        double[] X = new double[B.length];
        double[] lastX = new double[B.length];
        double error = 1e-10;

        while (true) {
            fjPool.invoke(new JacobiTask(A, B, X, lastX, 0, A.length, error));

            double errorNorm = 0;
            for (int i = 0; i < X.length; i++) {
                errorNorm += Math.abs(X[i] - lastX[i]);
                lastX[i] = X[i];
            }

            if (errorNorm < error) break;
        }

        // Imprime el resultado
        for (double x : X) System.out.println(x);
    }
}
```