TAREA 6 - METODOS NUMERICOS

Guillermo Segura Gómez
Centro de Investigación en Matemáticas
Métodos Numéricos
25 de septiembre de 2023

1 Introducción

En el ámbito de la matemática y la ingeniería, la solución de sistemas de ecuaciones lineales es una tarea esencial. Estos sistemas emergen en una amplia gama de problemas, desde la simulación de procesos físicos hasta optimizaciones en ingeniería y ciencias de la computación. Ya hemos analizado previamente los sistemas directos para la solución de sistemas lineales. Sin embargo, los sistemas que emergen de la discretización de ecuaciones diferenciales parciales o de la modelización de grandes sistemas son tan grandes y complejos que las técnicas directas de solución, como la eliminación de Gauss, se vuelven imprácticas debido a su alto costo computacional. [1]

En este contexto, los métodos iterativos presentan una alternativa atractiva debido a su potencial para resolver sistemas grandes de manera eficiente, especialmente cuando estos sistemas son dispersos. Dos de los métodos iterativos más fundamentales y ampliamente utilizados son el método de Jacobi y el método de Gauss-Seidel. [1]

El método de Jacobi, nombrado en honor al matemático alemán Carl Gustav Jacob Jacobi, es un proceso iterativo en el cual cada variable es resuelta de forma aislada con respecto a las otras en función de los valores de la iteración previa. Por otro lado, el método de Gauss-Seidel, desarrollado por los matemáticos alemanes Carl Friedrich Gauss y Philipp Ludwig von Seidel, mejora el enfoque de Jacobi al utilizar las variables ya calculadas dentro de la misma iteración para determinar las siguientes, acelerando así la convergencia. [1]

2 Método de Jacobi

Si tenemos el sistema $A\mathbf{x} = \mathbf{b}$, el método de Jacobi se define por [1]

$$x_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(-\frac{a_{ij} x_j}{a_{ii}} \right) + \frac{b_i}{a_{ii}}, \quad \text{para } i = 1, 2, \ldots, n$$

Iterativamente, la actualización de x_i se puede definir como:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k-1)} + b_i \right), \quad \text{para } i = 1, 2, \ldots, n$$

2.1 Pseudocódigo

```
ENTRADA: n, a_{ij}, b_i, X0_i, TOL, N

1. k = 1;
2. MIENTRAS k <= N HACER
  2.1. PARA i=1 A n
         x_i = (1 / a_{ii}) * (-SUM_{j=1, j!=i}^{n} a_{ij} * X0_j + b_i);
       FIN PARA
  2.2. SI ||x - X0|| < TOL ENTONCES
         SALIDA(x_1, ..., x_n);
         PARE;
       FIN SI
  2.3. k = k + 1;
```

```
  2.4. PARA i=1 A n
         X0_i = x_i;
       FIN PARA
  FIN MIENTRAS
3. SALIDA ('numero maximo de iteraciones excedido');
```

Listing 1: Algoritmo de Jacobi

Ahora, presentamos la rutina que calcula el método de Jacobi

```
double* jacobiMethod(double *A, double *b, int n, int maxIter, double tol) {
    double *x = (double*) malloc(n * sizeof(double));  // Vector de solución
    double *oldX = (double*) malloc(n * sizeof(double));  // Vector para
    almacenar la solución de la iteración anterior
    int iter = 0;    // Contador de iteraciones

    // Inicializar el vector de soluciones x con ceros
    for(int i = 0; i < n; i++) {
        x[i] = 0.0;
    }

    while(iter < maxIter) {
        // Copiar el vector x a oldX
        for(int i = 0; i < n; i++) {
            oldX[i] = x[i];
        }

        // Actualizar cada valor de x según la formula de Jacobi
        for(int i = 0; i < n; i++) {
            double sigma = 0.0;
            for(int j = 0; j < n; j++) {
                if(i != j) {
                    sigma += A[i * n + j] * oldX[j];
                }
            }
            x[i] = (b[i] - sigma) / A[i * n + i];
        }

        // Calcular la norma de la diferencia entre x y oldX
        double diffNorm = 0.0;
        for(int i = 0; i < n; i++) {
            diffNorm += pow(x[i] - oldX[i], 2);
        }
        diffNorm = sqrt(diffNorm);

        // Chequear la convergencia
        if(diffNorm < tol) {
            break;
        }

        iter++;
    }

    // Liberar memoria de oldX ya que x será retornado
    free(oldX);

    // Si se alcanzan las iteraciones máximas, mostrar un mensaje
    if(iter == maxIter) {
```

```
        printf("Jacobi method did not converge within %d iterations\n",
    maxIter);
    }

    return x;
}
```

Listing 2: Rutina que calcula el algoritmo de Jacobi

Las rutinas principales se escriben en una bibliteca adjunta llamada **matrix.h**. Llamamos a esa librería en la cual estan contenidas entre otras funciones las funciones de Jacobi y Gauss-Seidel. En el main de nuestro programa tenemos lo siguiente.

```
# include <stdio.h>
# include <stdlib.h>
# include "matrix.h"

int main(int argc, char *argv[]){

    // Manejo de argumentos de la línea de comandos
    if(argc != 3) {
        printf("Uso: %s <nombre_archivo_matriz> <nombre_archivo_vector>\n",
    argv[0]);
        return 1;
    }

    const char* filename1 = argv[1];
    const char* filename2 = argv[2];

    // Declaramos un apuntador hacia un bloque de memoria específico de la
    matriz A
    double *A;
    int rows, cols;

    // Leemos la matriz en el archivo, pasamos por referencia el valor del
    apuntador
    if (ReadMatrix(filename1 ,&A, &rows, &cols) == 1) {
        free(A); // Liberamos memoria
        return 0;
    }

    // Leemos B
    double *B;
    int rowsB, colsB;  // Para no sobreescribir las dimensiones originales de
     A

    if (ReadMatrix(filename2 ,&B, &rowsB, &colsB) == 1) {
        free(B); // Liberamos memoria
        return 0;
    }

    int maxIter = 1000; // Número máximo de iteraciones
    double tol = 1e-10; // Tolerancia para la convergencia

    double *x = jacobiMethod(A, B, rows, maxIter, tol);

    // Imprimir la solución
    printf("La solución X es: \n");
```

```
    MatrixShow(1, rowsB, x);

    // Liberamos la memoria
    free(A);
    free(B);
    free(x);

    return 0;
}
```

Listing 3: Estructura main

Para construir la compilación utilizamos un archivo **makefile** utilizando el entorno *make*.

```
guillermo_sego@MacBook-Air Tarea06 % make
gcc -g -o build/Debug/Matrix.o -c Matrix.c
gcc -g -o build/Debug/P1_T6.o -c P1_T6.c
gcc -g -o build/P2_T6 build/Debug/Matrix.o build/Debug/P1_T6.o
guillermo_sego@MacBook-Air Tarea06 % ./build/P1_T6 SmallMatrix.txt
    SmallVector.txt
La solución x es:
3.000000        -2.500000       7.000000
```

Listing 4: Ejecución del método de jacobi

3 Método de Gauss - Seidel

El método de Gauss - Seidel, parte del algoritmo de Jacobi. Es prácticamente similar, solo que en vez de esperar hasta el final de la iteración para actualizar los valores de x, el método de Gauss-Seidel utiliza los nuevos valores tan pronto como están disponibles. [1]

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} + b_i \right), \quad \text{para } i = 1, 2, \ldots, n$$

3.1 Pseudocódigo

```
ENTRADA: n, a_{ij}, b_i, X0_i, TOL, N
1. k = 1;
2. MIENTRAS k <= N HACER
  2.1. PARA i=1 A n
         x_i = (1 / a_{ii}) * (-SUM_{j=1}^{i-1} a_{ij} * x_j - SUM_{j=i+1}^{n
    } a_{ij} * X0_j + b_i);
       FIN PARA
  2.2. SI ||x - X0|| < TOL ENTONCES
         SALIDA(x_1, ..., x_n);
         PARE;
       FIN SI
  2.3. k = k + 1;
  2.4. PARA i=1 A n
         X0_i = x_i;
       FIN PARA
  FIN MIENTRAS
3. SALIDA ('numero maximo de iteraciones excedido');
```

Listing 5: Algoritmo de Gauss-Seidel

Para la implementación de este nuevo método se construyo la siguiente rutina.

```
double* gaussSeidelMethod(double *A, double *b, int n, int maxIter, double
    tol) {
    int i, j, iter = 0;
    double sigma, diff, diffNorm;

    double *x = (double*) malloc(n * sizeof(double));  // Vector de solución

    // Inicializar el vector de soluciones x con ceros
    for(i = 0; i < n; i++) {
        x[i] = 0.0;
    }

    while(iter < maxIter) {
        diffNorm = 0.0; // Se usa para chequear la convergencia

        // Actualizar cada valor de x según la fórmula de Gauss-Seidel
        for(i = 0; i < n; i++) {
            sigma = 0.0;

            for(j = 0; j < n; j++) {
                if(i != j) {
                    sigma += A[i * n + j] * x[j];
                }
            }

            diff = (b[i] - sigma) / A[i * n + i] - x[i]; // Diferencia entre
    nuevo y antiguo valor de x[i]
            x[i] += diff; // Actualizar x[i]

            diffNorm += pow(diff, 2); // Agregar al cuadrado de la norma de
    la diferencia
        }

        diffNorm = sqrt(diffNorm); // Finalizar cálculo de la norma de la
    diferencia

        // Chequear la convergencia
        if(diffNorm < tol) {
            break;
        }

        iter++;
    }

    // Si se alcanzan las iteraciones máximas, mostrar un mensaje
    if(iter == maxIter) {
        printf("Gauss-Seidel method did not converge within %d iterations\n",
     maxIter);
    }

    return x;
}
```

Listing 6: Rutina que calcula el algoritmo de Gauss-Seidel

La función main prácticamente no cambia salvo para los llamados a la función Gauss-

Seidel, por lo que, la terminal se vería algo como lo siguiente. Seguimos utilizando makefile para poder compilar el archivo incluyendo las librerías.

```
guillermo_sego@MacBook-Air Tarea06 % make
gcc -g -o build/Debug/Matrix.o -c Matrix.c
gcc -g -o build/Debug/P2_T6.o -c P2_T6.c
gcc -g -o build/P2_T6 build/Debug/Matrix.o build/Debug/P2_T6.o
guillermo_sego@MacBook-Air Tarea06 % ./build/P2_T6 SmallMatrix.txt
    SmallVector.txt
La solución x es:
3.000000        -2.500000       7.000000
```

Listing 7: Ejecución del método de jacobi

4 Conclusiones

En el proceso de implementar y analizar los métodos iterativos de Jacobi y Gauss-Seidel, se identificaron varias características y consideraciones esenciales. Aunque ambos métodos buscan resolver sistemas lineales, Gauss-Seidel tiende a converger más rápidamente que Jacobi en muchos sistemas, aunque no hay garantía de esto. No obstante, una posible ventaja de Jacobi es su potencial para la paralelización, ya que sus cálculos de componentes son independientes, facilitando la distribución en múltiples procesadores. Mientras que ambos algoritmos son intuitivos en su implementación, Gauss-Seidel, debido a su naturaleza secuencial, presenta una lógica ligeramente más compleja. Es importante tener presente que su convergencia está asociada con propiedades específicas de la matriz del sistema, en particular, con matrices diagonalmente dominantes. A pesar de su utilidad, especialmente en sistemas grandes y dispersos, es vital incorporar diagnósticos para detectar convergencia lenta o no convergencia, lo que puede indicar la necesidad de optar por otros métodos, como la factorización LU, que ya hemos visto a lo largo del semestre.

References

[1] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical analysis*. Cengage learning, 2015.