Tarea 6 - Método de Jacobi y Gauss-Seidel
Métodos Numéricos

Rafael Alejandro García Ramírez

22 de septiembre de 2023

1. Introducción

Tanto el método de Jacobi como el de Gauss-Seidel se basan en una "generalización"del método del punto fijo. Antes de presentar las partes esenciales del algoritmo veremos la justificación formal. Para un sistema $A\mathbf{x} = \mathbf{b}$ haremos descompondremos tal que $A = D - L - U$, para D matriz diagonal, L matriz triangular inferior y U matriz triangular superior. Esto es

$$(D - L - U)\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad D\mathbf{x} = (L + U)\mathbf{x} + \mathbf{b} \tag{1}$$

Para algún D^{-1} que existe, es decir sí y solo sí $d_{ii} \neq 0$ para $i \in \{1, 2, \ldots, n\}$, podemos hacer

$$\mathbf{x} = D^{-1}(L + U)\mathbf{x} + D^{-1}\mathbf{b}$$

Haciendo esto iterativo, esto es suponiendo un proceso de resolución entre los elementos de los involucrados

$$\mathbf{x}^{(k)} = D^{-1}(L + U)\mathbf{x}^{(k-1)} + D^{-1}\mathbf{b}$$

Por lo tanto podemos generar las nuevas iteraciones siempre y cuando se tanga un valor inicial o previo. Si hacemos $M = D^{-1}(L + U)$ (una matriz cuadrada) y $\mathbf{c} = D^{-1}\mathbf{b}$ (un vector) tenemos

$$\mathbf{x}^{(k)} = M\mathbf{x}^{(k-1)} + \mathbf{c}$$

Lo cuál es solo el mismo esquema inicial, es decir, tenemos del lado derecho algo de la forma "$A\mathbf{x}+\mathbf{b}$", por lo que podemos generar un nuevo vector a partir de la información de nuestra matriz original y un estado inicial. Esta forma anterior es llamado método de **Jacobi**. Análogamente, si tomamos la forma izquierda de la expresión (1) y redistribuímos podemos hacer

$$(D - L - U)\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad (D - L)\mathbf{x} = U\mathbf{x} + \mathbf{b}$$

Si $\exists (D - L)^{-1}$ (lo cual se cumple si y solo si $a_{ii} \neq 0$ para $i \in \{1, 2, \ldots, n\}$ dado que $(D - L)$ es una matriz triangular inferior) podemos hacer transformar esta última ecuación en

$$\mathbf{x} = (D - L)^{-1}U\mathbf{x} + (D - L)^{-1}\mathbf{b} \quad \Rightarrow \quad \mathbf{x}^{(k)} = (D - L)^{-1}U\mathbf{x}^{(k-1)} + (D - L)^{-1}\mathbf{b}$$

Sea $G = (D - L)^{-1}U$ y $\mathbf{t} = (D - L)^{-1}\mathbf{b}$ tal que definimos el método de **Gauss-Seidel** como

$$\mathbf{x}^{(k)} = G\mathbf{x}^{(k-1)} + \mathbf{t}$$

Ambos métodos, en parte también por la necesitad de la existencia de los inversos, implican que su condición de convergencia sea que A una matriz diagonal estrictamente dominante[1], es decir

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| < |a_{ii}|$$

Sin embargo, para estos métodos en concreto ocurre que, dependiendo de que tan grande sea la diferencia en la ecuación anterior, puede existir convergencia aunque no se cumpla para alguna i. Para una convergencia mucho más estricta se puede revisar el radio espectral

$$\rho(D^{-1}(L + U)) < 1$$

Esta condición es necesaria y asegura la convergencia, sin embargo dado que conlleva determinar el valor propio de A, se limitará a determinar solamente si el elemento de la diagonal es mayor; en caso de serlo tendremos asegurada la convergencia del método.

A continuación, se explica el cálculo de manera más *"ligera"* y práctica describiendo su ejecución y retomando la idea del punto fijo.

[1] Al igual que los demás métodos que hemos visto hasta ahora para sistemas de ecuaciones.

2. Pseudocódigo

Los pseudocódigos que se utilizaron para realizar la implementación en `C` para los algoritmos de Jacobi y Gauss-Seidel son lo siguientes:

Algorithm 1 Resolver sistema de ecuaciones $Ax = b$ por el método de Jacobi

Require: $A = [a_{i,j}],\ b = [b_j],\ x_0 = [x_{0_j}]$
Ensure: Solución almacenada en x_0

procedure JACOBI($N, TOL, matriz, b, x_0, maxIter$)

    $suma \leftarrow 0$ ▷ Revisamos que la matriz sea diagonal dominante
    for $i \leftarrow 0$ **hasta** $N - 1$ **do**
        $suma \leftarrow 0$
        for $j \leftarrow 0$ **hasta** $N - 1$ **do**
            if $i \neq j$ **then**
                $suma \leftarrow suma + |matriz[i][j]|$
        if $suma > |matriz[i][i]|$ **then**
            Imprimir(ADVERTENCIA Jacobi: La matriz no es diagonal dominante : `A[i][i]`)

    $x_1 \leftarrow$ **malloc**(N * **sizeof**($double$)) ▷ Inicializamos el vector auxiliar
    for $i \leftarrow 0$ **hasta** $N - 1$ **do**
        $x_1[i] \leftarrow 0.0$

    while $maxIter > 0$ **do** ▷ Comenzamos Jacobi
        for $i \leftarrow 0$ **hasta** $N - 1$ **do**
            $suma \leftarrow 0$
            for $j \leftarrow 0$ **hasta** $N - 1$ **do**
                if $i \neq j$ **then**
                    $suma \leftarrow suma + matriz[i][j] \cdot x_0[j]$
            $x_1[i] \leftarrow (b[i] - suma)/matriz[i][i]$

        if $\max_i |x_{0_i} - x_{1_i}| <$ TOL **then** ▷ Condición Convergencia
            END

        for $i \leftarrow 0$ **hasta** $N - 1$ **do** ▷ Reescribimos x_1
            $x_0[i] \leftarrow x_1[i]$
        $maxIter \leftarrow maxIter - 1$
    Imprimir(Error Jacobi : Máximo iteraciones alcanzada.)
    Liberar memoria(x_1)

El proceso que realiza Jacobi se puede resumir en los siguientes puntos:

1. Establecer un vector solución inicial x_0 y un vector temporal x_1 donde ser guardarán los cambios dentro de la solución.
2. Evaluamos $(A\mathbf{x_0} - \mathbf{b})_{ij}$ y a cada valor i se lo asignamos a $x_1[i]$.
3. Repetimos el paso 2 con $(A\mathbf{x_1} - \mathbf{b})_{ij}$ hasta que $\max_i |x_{i-1} - x_i| <$ TOL.

Esto implica la necesidad de tener un espacio de n elementos extra para almacenar las soluciones temporales. Este espacio se libera dentro de la implementación del código. Notamos que dentro del anterior algoritmo se define como condición de convergencia que el mayor valor absoluto de la resta de los dos vectores sea menor a una tolerancia dada. Asimismo, se añade una advertencia cuando no se tiene un elemento diagonal dominante, dentro de la implementación se da como opción al usuario si continuar o no.

Si bien se presenta la necesitad de n más espacio, podemos ver que se requiere una cantidad de cálculos muchísimo menor en comparación de otros métodos de resolución de sistemas de ecuaciones, el problema será determinar un x_0 que permita la convergencia dentro del número de iteraciones que se tenga disponible[2]. En nuestro caso, se inicializó $\mathbf{x} = \mathbb{1}$.

Para el algoritmo de Gauss-Seidel se tiene un pseudocódigo bastante similar, con la única diferencia en cómo se actualiza el vector temporal x_1 con relación al vector solución x_0. Dentro de Jacobi, para un momento j dentro del cálculo

[2]Realmente las iteraciones además de controlar la cantidad de operaciones a las cuales estamos interesados en que se realice el proceso, es un control directo del tiempo que deseamos invertir o dedicar a la ejecución del programa.

de las $x^{(k)}$ se habrán calculado $i-1$ elementos los cuales son una aproximación al vector x_j, utilizando estos $x_j^{(j)}$ ya calculados dentro del cálculo de $x_i^{(k)}$ con $i \in \{j+1, j+2, \ldots, n\}$ podemos realizar una mejor estimación pues los datos actualizados se toman más próximamente en cuenta para realizar el cálculo, esto permite en lo generar una convergencia mucho más rápida a comparación con Jacobi.

Algorithm 2 Resolver sistema de ecuaciones $Ax = b$ por el método de Gauss-Seidel

Require: $A = [a_{i,j}]$, $b = [b_j]$, $x_0 = [x_{0_j}]$
Ensure: Solución almacenada en x_0

procedure GAUSS_SEIDEL(N, TOL, matriz, b, x_0, maxIter)

    $suma \leftarrow 0$ ▷ Revisamos que la matriz sea diagonal dominante
    for $i \leftarrow 0$ **hasta** $N-1$ **do**
        $suma \leftarrow 0$
        for $j \leftarrow 0$ **hasta** $N-1$ **do**
            if $i \neq j$ **then**
                $suma \leftarrow suma + |matriz[i][j]|$
        if $suma > |matriz[i][i]|$ **then**
            Imprimir(ADVERTENCIA Gauss-Seidel: La matriz no es diagonal dominante : `A[i][i]`)

    $x_1 \leftarrow$ **malloc**(N * **sizeof**($double$)) ▷ Inicializamos el vector auxiliar
    for $i \leftarrow 0$ **hasta** $N-1$ **do**
        $x_1[i] \leftarrow 0.0$

    $suma1, suma2 \leftarrow 0, 0$
    while $maxIter > 0$ **do** ▷ Iniciamos Gauss-Seidel
        for $i \leftarrow 0$ **hasta** $N-1$ **do**
            $suma1, suma2 \leftarrow 0, 0$
            for $j \leftarrow 0$ **hasta** $i-1$ **do**
                $suma1 \leftarrow suma1 + matriz[i][j] \cdot x_1[j]$
            for $j \leftarrow i+1$ **hasta** $N-1$ **do**
                $suma2 \leftarrow suma2 + matriz[i][j] \cdot x_0[j]$
            $x_1[i] \leftarrow (b[i] - suma1 - suma2)/matriz[i][i]$

        if $\max_i |x_{1_i} - x_{0_i}| < TOL$ **then** ▷ Condición de Convergencia
            END

        for $i \leftarrow 0$ **hasta** $N-1$ **do** ▷ Reescribimos x_1
            $x_0[i] \leftarrow x_1[i]$
        $maxIter \leftarrow maxIter - 1$
    Imprimir(Gauss-Seidel no converge en maxIter iteraciones)
    Liberar memoria(x_1)

Como se había descrito antes, podemos ver que Gauss-Seidel requiere del uso de un momento extra para el cálculo de principal. El desempeño de cada uno es considerablemente diferente, esta diferencia en su número de iteraciones para converger es mucho más notoria para matrices grandes que para matrices ordinarias; veamos algunos resultados.

3. Resultados

Para una tolerancia $\texttt{TOL} = 1e-16$, se realizaron pruebras en dos conjuntos de datos, uno `Small` que consistía en un sistema 3×3 y uno `Big` de 125×125, sea $\texttt{N}_{\texttt{d}(\texttt{x}_1,\texttt{x}_0)\neq 0}$ la cantidad de valores comprobados tal que $A\mathbf{x}_0 - \mathbf{b} \neq 0$ para una entrada j de $\mathbf{x}_0$[3]; los resultados fueron los siguientes:

	Método	
Datos	Jacobi	Gauss - Seidel
Small	`Converge en 13 iteraciones` $\texttt{N}_{\texttt{d}(\texttt{x}_1,\texttt{x}_0)\neq 0} = 0$	`Converge en 9 iteraciones` $\texttt{N}_{\texttt{d}(\texttt{x}_1,\texttt{x}_0)\neq 0} = 0$
Big	`Converge en 5443 iteraciones` $\texttt{N}_{\texttt{d}(\texttt{x}_1,\texttt{x}_0)\neq 0} = 0$	`Converge en 2692 iteraciones` $\texttt{N}_{\texttt{d}(\texttt{x}_1,\texttt{x}_0)\neq 0} = 0$

[3]Para la modificación final de $\mathbf{x}_0$ una vez concluido el algoritmo escogido.

Notamos que en todos los casos se tiene una comprobación $A\mathbf{x} - \mathbf{b} = 0$ en cada una de las entradas. Para la cantidad de iteraciones vemos que mientras que para un sistema pequeño se tiene una disminución del 30 % de las iteraciones, el sistema grande se tiene un 50 %, una cantidad bastante considerable.

Es importante añadir, que la cantidad de iteraciones naturalmente debe crecer conforme el sistema crezca en tamaño, por lo que la cantidad mínima de iteraciones con la que se puede resolver un sistema (suponiendo que exista alguna forma de conocerla) será siempre variable con respecto a su tamaño.

4. Conclusiones

Los métodos iterativos de Jacobi y Gauss-Seidel ofrecen varias ventajas significativas debido a su facilidad de comprensión, implementación y eficiencia computacional. Aunque en ciertos casos pueden surgir problemas de estabilidad al calcular matrices específicas, sus beneficios superan en gran medida sus limitaciones.

Otro aspecto crucial a considerar es su flexibilidad en cuanto a la tolerancia. Dado que estos algoritmos tienen una condición de parada explícita basada en la diferencia entre iteraciones sucesivas, uno puede ajustar esta diferencia según los requisitos específicos del problema.

No obstante, es importante destacar la importancia de las condiciones de convergencia. Si bien existe una condición de convergencia débil, determinar una condición de convergencia fuerte puede ser complicado y costoso en el caso de matrices grandes. Por lo tanto, la elección entre estos métodos debe basarse en la información disponible sobre el sistema al que se está aplicando.

5. Referencias

1. Richard. L. Burden y J. Douglas Faires, Análisis Numérico, 7a Edición, Edito-rial Thomson Learning, 2002.
2. Samuel S M Wong, Computational Methods in Physics and Enginering, Ed. World Scientific, 3rd Edition, 1997.
3. Teukolsky, S. A., Flannery, B. P., Press, W. H., & Vetterling, W. T. (1992). Numerical recipes in C. SMR, 693(1), 59-70.
4. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd Edition, Cambridge University Press, 2007

Anexo: otros pseudocódigos

A continuación se presentan los pseudocógios que complementan la sección 2 de este documento que brindan ayuda para la resolución del problema completo.

Algorithm 3 Calcular la máxima diferencia entre elementos correspondientes de dos vectores

```
1: function CALCULARMAXIMADIFERENCIA(N, vector_1, vector_2)
2:     maximaDiferencia ← 0                          ▷ Inicializamos la máxima diferencia en cero
3:     for i ← 0 hasta N − 1 do
4:         diferencia ← |vector_1[i] − vector_2[i]|
5:         if diferencia > maximaDiferencia then
6:             maximaDiferencia ← diferencia
7:     return maximaDiferencia
```