Mètodes Numerics II

Problema 26: Explicació dels passos seguits

Martí Rubio Estrada

October 2018

Consideracions

En aquest exercici tenim un sistema de la forma $Ax = b$ on:

$$A = \begin{pmatrix} 3 & 0 & -1 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 4 & 0 & -1 & 0 & \dots & 0 & 0 & 1 \\ -1 & 0 & 3 & 0 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & & \dots & \vdots & \vdots & \vdots \\ 1 & 0 & \dots & & & \dots & 0 & 3 & 0 \\ 0 & 1 & \dots & & & \dots & -1 & 0 & 4 \end{pmatrix}; \qquad b = \begin{pmatrix} \frac{2}{n} \\ \frac{2}{n} \\ \frac{4}{n} \\ \frac{4}{n} \\ \vdots \\ 1 \\ 1 \end{pmatrix}.$$

Primer de tot, veiem que podem descompondre A en les matrius L, D, U com:

$$D = \begin{pmatrix} 3 & 0 & \dots & 0 \\ 0 & 4 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 3 \end{pmatrix}, \quad U = \begin{pmatrix} 0 & 0 & -1 & \dots & 1 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 1 \\ \vdots & & & & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & -1 \\ 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 \end{pmatrix}, \quad L = U^T.$$

També veiem que A és definida positiva ja que és diagonal dominant. En aquest moment, i sabent que:

$$\begin{cases} B_J = -D^{-1}(L+U) \\ B_{GS} = (D+L)^{-1}U \end{cases}$$

podem calcular fàcilment la matriu B_J, ja que la inversa de D és ella mateixa amb els elements de la diagonal elevats a -1. Raonant el producte entre D^{-1} i $L+U$ ens queda:

$$B_J = \begin{pmatrix} 0 & 0 & 1/3 & \dots & -1/3 & 0 \\ 0 & 0 & 0 & \ddots & 0 & -1/4 \\ \vdots & & & & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 1/4 \\ -1/3 & 0 & \dots & 0 & 0 & 0 \\ 0 & -1/4 & \dots & 1/4 & 0 & 0 \end{pmatrix}.$$

I ara, si calculem la norma infinit induïda per a les matrius ens queda:

$$\|A\|_\infty = \max_{0\leq i\leq n} \{ \sum_{j=0}^{n} \mid a_{ij} \mid \}.$$

Veiem de manera evident que: $\|B_J\|_\infty = 2/3 < 1$ i per tant, el mètode de Jacobi convergirà. També veiem que com és estrictament diagonal dominant per files, el mètode de Gauss-Seidel també convergeix. Si volem acotar ara i per fi el mòdul infinit de la matriu de Gauss-Seidel ho podem fer mitjançant la formula:

$$\|B_{GS}\|_\infty \leq \max_{1\leq l\leq n} \frac{r_l}{1-s_l} \quad \text{on } r_l = \sum_{j>l} \frac{|a_{l,j}|}{|a_{l,l}|} \quad \text{i} \quad s_l = \sum_{j<l} \frac{|a_{l,j}|}{|a_{l,l}|}$$

Fent els càlculs veiem que:

$$\|B_{GS}\| \leq \max\{2/3, 1/2, 1/3, 0\} = 2/3$$

i per tant, com ja hem dit, convergent. I és més, veiem que en el pitjor dels casos serà igual de lent que Jacobi. Així que si el nombre d'iteracions de Gauss-Seidel és major voldrà dir que tenim algun problema.

El programa

La meva resolució en C usa el fet que sabem de manera teòrica la inversa de la matriu D per al càlcul del mètode de Jacobi.

Mètode de Jacobi

Primer de tot, comencem amb el vector $x^{(0)}$ posat a zero. Aquest, juntament amb el vector b i el vector solució anterior tenen les dades dins seu. En el cas de la matriu A tenim la funció `matrix_position(int i, int j)` que retorna el valor de A_{ij}. En quant a l'algorisme, seguim la següent estructura:

- Posem el vector solució a zero.
- Mentre l'error sigui major a 10^{-12}
  - Multipliquem $(U + L)$ per el vector solució
  - Restem el vector b al vector solució
  - Es multiplica la matriu D^{-1} per el vector solució
  - Re-calculem l'error i comprovem si estem per sota de l'error mínim acceptable.

Només comentar que la matriu D^{-1} tampoc la tenim guardada, tenim la funció `D_inversa_mult(double *vector_solucio)` a la qual se li passa el vector solució i segons la posició es multiplica pel que toca.

Mètode de Gauss-Seidel

Sabem que el mètode iteratiu de Gauss-Seidel és el següent:

$$x^{(k+1)} = (L + D)^{-1}(b - Ux^{(k)})$$

Tot i això, tenim un problema: no podem calcular $(L + D)^{-1}$. Com ja hem comentat abans, les matrius no es poden guardar en memòria ja que, hauriem de guardar 10^{12} xifres en format double, hauriem d'ocupar

$$10^{12} * 8\text{B} = 8 * 10^{12}\text{B} \simeq 7.27\text{TB}$$

que és més del que pot arribar a tenir un ordinador qualsevol. És per això que, usant el fet que $L + D$ és una matriu triangular superior, podem redefinir l'algoritme de la següent manera:

$$x_i^{(k+1)} = \frac{1}{A_{i,i}} \left(b_i - \sum_{j=1}^{i-1} A_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{i,j} x_j^{(k)} \right)$$

que ho podem fer ja que la funció `matrix_position(i, j)` ens retorna $A_{i,j}$.

Una cosa curiosa que ha passat durant el testeig d'aquest algoritme ha estat el tema de les referències dels punters. A l'algoritme de Jacobi, quan comença una nova iteració fem una assignació `vector_solution_ant = vector_solution` i acte seguit cridem a la funció `U_mes_L_mult` que ens retorna no un resultat sinó un punter a un nou vector, i per tant els dos vectors apunten a referències diferents. En aquest segon algoritme tornem a fer la mateixa assignació però no cridem a cap funció que ens retorni un vector, sinó a una que ens retorna només valors. És per això que un cop acabava la primera iteració havíem canviat els valors dels vectors solució i solució anterior per igual, donant error zero! Per això s'ha fet la funció `assign_vectors(double* v_ant, double* v, int n)` que assigna els valors del vector solució al vector solució anterior un a un, evitant així que siguin el mateix objecte.

Mètode SOR

Aquest mètode és molt semblant al de Gauss-Seidel, només ens cal trobar el valor òptim d'ω per tal de trobar una convergència més ràpida que amb als mètodes anteriors. Si provem uns 20 valors d'ω el programa ens retorna la següent sortida:

I) ω: 0.1 Total d'iteracions: 693

II) ω: 0.2 Total d'iteracions: 339

III) ω: 0.3 Total d'iteracions: 217

IV) ω: 0.4 Total d'iteracions: 156

V) ω: 0.5 Total d'iteracions: 118

VI) ω: 0.6 Total d'iteracions: 93

VII) ω: 0.7 Total d'iteracions: 74

VIII) ω: 0.8 Total d'iteracions: 60

IX) ω: 0.9 Total d'iteracions: 49

X) ω: 1.0 Total d'iteracions: 40

XI) ω: 1.1 Total d'iteracions: 33

XII) ω: 1.2 Total d'iteracions: 31

XIII) ω: 1.3 Total d'iteracions: 38

XIV) ω: 1.4 Total d'iteracions: 48

XV) ω: 1.5 Total d'iteracions: 62

XVI) ω: 1.6 Total d'iteracions: 83

XVII) ω: 1.7 Total d'iteracions: 117

XVIII) ω: 1.8 Total d'iteracions: 184

XIX) ω: 1.9 Total d'iteracions: 386

Si ara provem valors entre 1.10 i 1.20 que és on es troba el mínim ens queda:

I) ω: 1.10 Total d'iteracions: 33

II) ω: 1.11 Total d'iteracions: 32

III) ω: 1.12 Total d'iteracions: 31

IV) ω: 1.13 Total d'iteracions: 31

V) ω: 1.14 Total d'iteracions: 30

VI) ω: 1.15 Total d'iteracions: 29

VII) ω: 1.16 Total d'iteracions: 29

VIII) ω: 1.17 Total d'iteracions: 30

IX) ω: 1.18 Total d'iteracions: 30

X) ω: 1.19 Total d'iteracions: 30

XI) ω: 1.20 Total d'iteracions: 31

I veiem que els valor òptim d'ω, ω, es troba a: $\tilde{\omega} \in (1.15, 1.16)$ donant un nombre d'iteracions de 29 o menor. Veiem que és bastant millor a Gauss-Seidel, si comparem el nombre d'iteracions i el temps d'execució veiem que, com era d'esperar, el mètode SOR és el millor.

	Jacobi	Gauss-Seidel	SOR
Iteracions	67	40	29
Temps (s)	2.69	1.23	0.96