Calcul Numérique TP

YANG Yizhi

8 janvier 2024

Table des matières

1 Introduction : Problème Poisson 1D

On s'intéresse au problème du Poisson à dimension 1. Le problème de Poisson 1D est un système des équations aux dérivées partielles couramment rencontrée en mathématiques appliquées, en physique et en ingénierie. Dans le monde de physique macroscopique, tous les problèmes sont continus dans l'espace réel, et pourtant, dans le monde de l'informatique, la discrétisation est toujours nécessaire.

On applique le maillage uniforme, c'est-à-dire sur le même axe, on prend des ponits équidistants, et à chaque point on aura une équation de dérivées partielles de second degré par rapport à une condition locale. Pendant les cours de TD précédents, on a écrit le système de ces équations en forme matricielle tridiagonale. Donc résoudre ce problème est équivalent à résoudre l'équation du produit matriciel $Ax = b$. Par une simplification de condition des seconds menbres, on sait que la solution analytique est sous forme linéaire $T(x) = T_0 + x(T_1 - T_0)$.

Dans la suite de ce TP, on va d'abord introduire ce problème et puis proposer des méthodes afin d'approximer la solution, qui sera comparée avec la solution exacte.

2 Test d'environnement

En créant une image dans Docker Container, on configure l'environnement du travail pour que les bibliothèques CBLAS et LAPACK soient reconnues lors de la compilation. Un fichier de test a été créé et le résultat se donne :

```
The exponantial value is e = 2.718282
The maximum single precision value from values.h is maxfloat = 3.402823e+38
The maximum single precision value from float.h is flt_max = 3.402823e+38
The maximum double precision value from float.h is dbl_max = 1.797693e+308
The epsilon in single precision value from float.h is flt_epsilon = 1.192093e-07
The epsilon in double precision value from float.h is dbl_epsilon = 2.220446e-16

Test of ATLAS (BLAS/LAPACK) environment
x[0] = 1.000000, y[0] = 6.000000
x[1] = 2.000000, y[1] = 7.000000
x[2] = 3.000000, y[2] = 8.000000
x[3] = 4.000000, y[3] = 9.000000
x[4] = 5.000000, y[4] = 10.000000

Test DCOPY y <- x
y[0] = 1.000000
y[1] = 2.000000
y[2] = 3.000000
y[3] = 4.000000
y[4] = 5.000000
```

3 Méthode directe

3.1 Stockage Bande

3.1.1 Déclaration et allocation en C

En effet, on doit déclarer et allouer les matrices de manière à ce qu'elles soient compatibles avec les fonctions BLAS/LAPACK, donc de forme General Band ici. C'est une forme efficace pour les matrices multidiagonales, car la plupart de leur éléments sont nuls. Au lieu d'occuper une mémoire de taille $n * n$, il suffira de $lab * n$, avec lab le nombre des diagonales.

3.1.2 la Constante `LAPACK_COL_MAJOR`

Cette constante signifie que la matrice donnée est stockée colonne par colonne. Afin de convertir les indices usuels, il faut définir la fonction suivante :

```
// i, j from 0 to *la -1
int indexABCol(int i, int j, int *lab)
{
  return (j + 1) * (*lab - 1) + i - 1;
}
```

Listing 1 – General Band Indices

3.1.3 la Dimension Principale

Cela signifie la largeur de la bande stockée, i.e. le nombre total des diagonales (la diagonale principale, les subdiagonales et sousdiagonales).

3.2 Explication des Fonctions BLAS/LAPACK

3.2.1 dgbmv

C'est une fonction BLAS qui effectue une multiplication d'une matrice (stockée en bande) par un vecteur. La lettre "d" signifie que les éléments contenus dans les matrices sont de type double.

3.2.2 dgbtrf

C'est une routine LAPACK qui effectue la factorisation LU d'une matrice générale stockée en bande, donc les éléments sont de type double.

3.2.3 dgbtrs

C'est une routine LAPACK qui résout un système d'équations linéaires pour une matrice stockée en bande après une factorisation LU. Ses éléments sont de type double.

3.2.4 dgbsv

C'est similaire de dgbtrs mais avec pivotage partiel, qui est une combinaision de $dgbtrf$ et $dgbtrs$.

3.2.5 Calcul de la norme du résidu relatif

La norme du résidu relatif s'écrit sous la forme :

$$\|r\|_2 = \frac{\|Ax - b\|_2}{\|b\|_2}$$

On applique d'abord la fonction *cblas_dgbmv* pour calculer le produit matrice-vecteur Ax, puis on effectue l'addition de deux vecteurs Ax et b par la fonction *cblas_daxpy*.

On pourra utiliser la fonction *cblas_dnrm*2 pour calculer la norme euclidienne.

3.2.6 Méthode de validation

Pour la solution du système $Ax = b$, on pourra comparer l'erreur relative de x par rapport à la solution exacte x_{exact} :

$$\|err\|_2 = \frac{\|x - e_{exact}\|_2}{\|x_{exact}\|_2}$$

On pourra aussi utiliser le résidu relatif, cela donne un ordre de grandeur 10^{-16} :

```
The relative forward error is relres = 2.393518e-16
```

Pour la factorisation LU, on pourra calculer $\|A - LU\|_2$, si cette valeur est très petit, on peut dire que la méthode est validée.

3.2.7 Performance et complexité

On rajoute une mesure de temps pour les méthodes différentes en utilisant l'entête ¡time.h¿ et les fonctions suivantes :

```
clock_t start_timer, end_timer;
double elapsed_time;
start_timer = clock();
// code to calculate solution
end_timer = clock();
elapsed_time = (double)(end_timer - start_timer) / (double)CLOCKS_PER_SEC;
printf("Elapsed time for direct method: %lf seconds\n", elapsed_time);
```

Listing 2 – Time Measure

La méthode avec LU factorisation $dgbtrf$ puis $dgbtrs$ donne :

```
Elapsed time for direct method: 0.000823 seconds
```

La méthode direct *dgbsv* donne :

```
Elapsed time for direct method: 0.000909 seconds
```

Complexité : $gdbtrf$ admet une complexité $O(n^3)$ pour la factorisation LU, $dgbtrs$ entre $O(n^2)$ et $O(n^3)$; pour *dgbsv*, il faut $O(n)$ pour la factorisation LU et $O(n)$ pour résoudre le système. (avec n la taille de matrice)

4 Méthodes itératives

Pour la méthode alpha-Richardson, on calcule le facteur α en utilisant les valeurs propres de la matrice du Poisson 1D :

$$\alpha = \frac{2}{max(eigval) + min(eigval)}$$

Pour la méthode Jacobi (respectivement Gauss-Seidel), on calcule d'abord la matrice $M = D$ (respectivement $M = D - E$), ensuite l'inverse de matrice M^{-1}. Par itération, on calcule $x^{k+1} = x^k + M^{-1}r^k$ avec le résidu $r^k = b - Ax$.

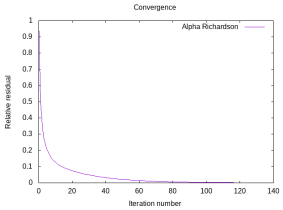

FIGURE 1 – Alpha-Richardson Iteration Convergence

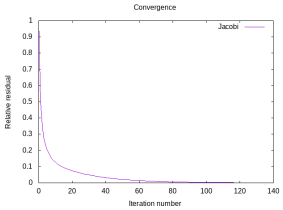

FIGURE 2 – Jacobi Iteration Convergence

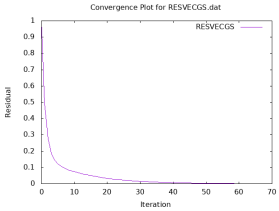

FIGURE 3 – Gauss-Seidel Iteration Convergence

Comme la valeur $alpha\_opt = 0.5$ et $D^{-1} = 0.5I_n$, la méthode alpha-Richardson et la méthode Jacobi sont équivalentes :

```
Optimal alpha for simple Richardson iteration is : 0.500000
The relative forward error is relres = 5.093853e-03
Elapsed time for iterative methods: 0.000867 s
```

Elles partagent aussi la même vitesse de convergence, la méthode de Gauss-Seidel converge deux fois plus vite que celles-ci.

5 Stockage CSR et CSC

Pour la méthode CSR (Compressed Sparse Row), on stocke les valeurs significatives, les indices de colonnes et les pointeurs de lignes de la matrice ciblée dans trois tableaux unidimensionnels. Ici un algorithme possible pour la matrice tridiagonale du Poisson 1D :

Algorithm 1 MatriceTridiagonaleVersCSR

Data: matrice
Result: values, columns, row_ptr

```
// Initialisation des tableaux
values ← [] ;                          // Tableau pour les valeurs non nulles
columns ← [] ;                         // Tableau pour les indices de colonnes
row_ptr ← [] ;                         // Tableau pour les pointeurs de lignes
for chaque ligne i de la matrice do
    // Parcourir les éléments de la ligne
    for chaque élément j de la ligne i do
        if j est sur la diagonale principale then
            Ajouter A[i, j] à values  Ajouter j à columns
        else if j est sur la diagonale supérieure ou inférieure then
            Ajouter A[i, j] à values  Ajouter j à columns
    // Ajouter la taille actuelle de values à row_ptr
    Ajouter la taille actuelle de values à row_ptr
return values, columns, row_ptr
```

Similairement, pour un stockage CSC (Compressed Sparse Column), il faut just inverser les rôles de lignes et colonnes.

6 Conclusion

La convergence plus rapide de Gauss-Seidel par rapport à Jacobi dans le contexte spécifique du problème de Poisson 1D avec une matrice tridiagonale peut être attribuée à la nature séquentielle de la mise à jour, qui peut mieux exploiter la structure de la matrice et conduire à une convergence plus rapide.

Le choix entre le stockage en format général band (GB), Compressed Sparse Row (CSR), et Compressed Sparse Column (CSC) dépend des caractéristiques spécifiques de la matrice et des opérations que l'on a l'intention de réaliser. Le format GB est plus convenable pour les matrices tridiagonales et il donne un accès efficace aux éléments adjacents dans la bande ; les formats CSR

et CSC sont plus convenables pour les matrices creuses, car ils ne nécessitent pas un stockage pour les zéros, mais ils ont un surcoût pour accéder aux éléments dans le même colonne (pour CSR) ou la même ligne (pour CSC).