Résolution d'un système triangulaire supérieur

Février 2020

1 Introduction

Le but de ce TP est de se familiariser avec la syntaxe de base de Fortran (déclarations de variables, définitions et utilisations de procédures, de fonctions, structures de contrôles : boucles, etc.) et d'illustrer l'intérêt d'implanter un algorithme qui suive le schéma de stockage imposé par le langage utilisé.

Il s'agit d'implanter deux versions différentes d'un algorithme de résolution d'un système triangulaires supérieur : la résolution triangulaire *sans report* et la résolution triangulaire *avec report*. Les deux algorithmes sont rappelés et illustrés ci-dessous. Notez qu'ils effectuent exactement les mêmes calculs, seul l'ordre change.

Algorithme 1
Résolution triangulaire sans report

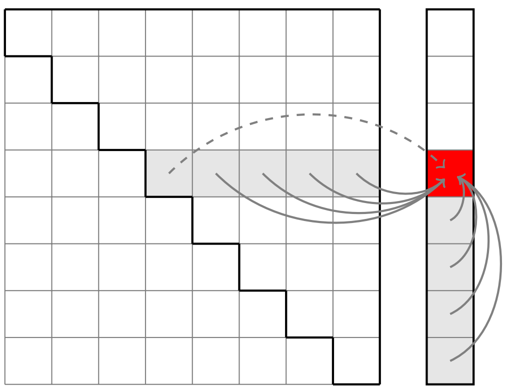
Entrées : matrice triangulaire U
second membre b
Sortie : solution $x = U^{-1}b$

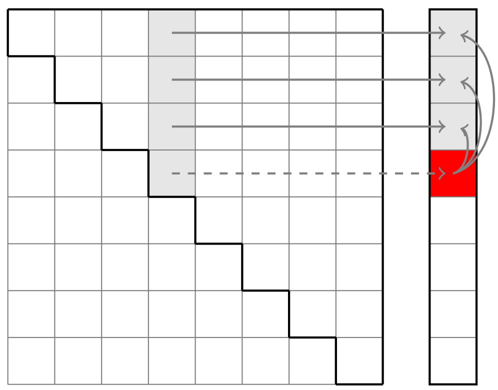
$x = b$
pour $j = n$ **à** 1 **faire**
  pour $i = n$ **à** $j + 1$ **faire**
    $x_j = x_j - u_{ji}x_i$
  fin pour
  $x_j = \dfrac{x_j}{u_{jj}}$
fin pour

Algorithme 2
Résolution triangulaire avec report

Entrées : matrice triangulaire U
second membre b
Sortie : solution $x = U^{-1}b$

$x = b$
pour $j = n$ **à** 1 **faire**
  $x_j = \dfrac{x_j}{u_{jj}}$
  pour $i = 1$ **à** $j - 1$ **faire**
    $x_i = x_i - u_{ij}x_j$
  fin pour
fin pour





2 Implantation

Le fichier `test_solve_sup.F90` contient un squelette de programme principal qui initialise la matrice et le second membre ; la matrice est stockée dans un tableau carré dont la partie triangulaire inférieure ne doit pas être accédée. Complétez `test_solve_sup.F90` en rajoutant deux procédures `left_looking_solve` (résolution sans report) et `right_looking_solve` (résolution avec report), qui doivent avoir l'interface suivante :

`[left/right]_looking_solve(U,x,b,n)`

Sémantique : effectue la résolution sans/avec report du système triangulaire $\mathtt{Ux} = \mathtt{b}$.

Entrées :
- `U`, matrice de taille $\mathtt{n}\times\mathtt{n}$ de nombres réels double précision.
- `b`, second membre, vecteur de taille `n` de nombres réels double précision.
- `n`, entier.

Sortie : `x`, vecteur de taille `n`.

Pré-conditions :
- `U` est initialisée et aucun terme de sa diagonale n'est nul.
- $\mathtt{n} > 0$.

Post-conditions : `x` contient la solution de $\mathtt{Ux} = \mathtt{b}$.

Ajoutez également une fonction de calcul de l'erreur inverse avec l'interface suivante :

`backward_error(U,x,b,n)`

Sémantique : calcule l'erreur inverse $\frac{||\mathtt{Ux-b}||_2}{||\mathtt{b}||_2}$.

Entrées :
- `U`, matrice de taille $\mathtt{n}\times\mathtt{n}$ de nombres réels double précision.
- `x`, solution calculée, vecteur de taille `n` de nombres réels double précision.
- `b`, second membre, vecteur de taille `n` de nombres réels double précision.
- `n`, entier.

Retour : un nombre réel double précision.

Pré-conditions : $\mathtt{n} > 0$.

Post-conditions : $\emptyset$.

Pour compiler, utilisez `make`. Pour lancer le code principal, lancez `test_solve_sup`.

3 Performances

Utilisez la fonction `cpu_time` afin de mesurer le temps passé dans les procédures `left_looking_solve` et `right_looking_solve`. Faites des tests sur des matrices de tailles raisonnables ($n \leqslant 20000$) et expliquez les différences de performances entre les deux algorithmes.

4 Question subsidiaire

Dans l'algorithme `left_looking_solve`, transformez la boucle i qui est décroissante de $n, \ldots, j+1$ en une boucle croissante $j+1, \ldots, n$, ce qui en arithmétique exacte calcule la même chose. Que constatez-vous expérimentalement ?

Code à rendre

Déposer sous moodle une archive contenant le ou les fichiers nécessaires (a priori un seul, si vous avez tout écrit dans *test_solve_sup.F90*).