Programmation pour le calcul scientifique
Année : 2019-2020
Formation : L3 Ingénierie Mathématique

TP9 : Classes de solveurs d'EDP

Au cours de ce TP, nous allons structurer un code de résolution de l'équation de la chaleur à l'aide de différences finies, en suivant une organisation qui est fréquemment rencontrée dans les codes de calcul. Ce n'est bien sûr pas la seule façon de faire, mais celle-ci est une première illustration du fonctionnement d'un code orienté objet.

Il y a beaucoup à faire dans ce TP, qui sera probalement réparti sur plusieurs séances, n'hésitez pas à travailler à plusieurs. Lisez bien l'ensemble des indications avant de vous lancer dans l'écriture, en faisant un schéma au fur et à mesure si besoin.

Problème à résoudre, planification de la structure du programme.

Nous cherchons à approcher la solution à l'équation

$$\begin{cases} \partial_t u(t,\boldsymbol{x}) - \nabla \cdot (k(\boldsymbol{x})\nabla u(t,\boldsymbol{x})) = q(t,\boldsymbol{x}) \text{ dans } \Omega, \\ \partial_n u(t) = 0 \text{ sur } \partial\Omega, \\ u(t_0,\boldsymbol{x}) = u_0(\boldsymbol{x}), \end{cases} \tag{1}$$

pour t dans $[t_0, t_{\text{final}}]$ et $\boldsymbol{x}$ dans le domaine $\Omega = [0,1] \times [0,1]$. q désigne le terme source de l'équation et k est le coefficient de diffusion. k peut varier en espace, et peut même être tensoriel, mais ici nous nous contenterons d'un scalaire homogène en espace.

La discrétisation de l'équation se fera sur une grille $N_x \times N_y$ de pas d'espace réguliers h_x et h_y avec un pas de temps δt. En 2D, la discrétisation de l'équation s'écrit

$$\begin{aligned} u_{i,j}^{n+1} &+ k\frac{\delta t}{h_x^2}\left(2u_{i,j}^{n+1} - u_{i+1,j}^{n+1} - u_{i-1,j}^{n+1}\right) \\ &+ k\frac{\delta t}{h_y^2}\left(2u_{i,j}^{n+1} - u_{i,j+1}^{n+1} - u_{i,j-1}^{n+1}\right) = u_{i,j}^{n} + \delta t q_{i,j}^{n}. \end{aligned}$$

La condition de Neumann homogène aux bords est prise en compte en écrivant

$$\frac{u_{0,j}^{n+1} - u_{-1,j}^{n+1}}{h_x} = 0 \text{ en } x = 0, \quad \frac{u_{N,j}^{n+1} - u_{N-1,j}^{n+1}}{h_x} \text{ en } x = 1,$$

et des relations similaires dans la direction y.

La résolution de l'équation sera testée en prenant $u_0 = 0$ sur tout le domaine et comme terme source :

$$q(t,x) = \exp\left(-\frac{(\boldsymbol{x} - \boldsymbol{x_0})^2}{0.1}\right) \mathbb{1}_{[t_0,t_0+2]}(t)$$

On s'attend à ce que la solution augmente progressivement autour de $\boldsymbol{x_0}$ pendant l'intervalle de temps $[t_0, t_0 + 2]$, et qu'il n'y ait plus que de la diffusion une fois le terme source coupé.

La solution sera écrite dans des fichiers `.vtk` qui sont lisibles dans Paraview, par exemple [1]. Ces fichiers pouvant être assez lourds pour de grands nombres de points N, il ne vaut mieux pas écrire la solution à chaque itération en temps, mais avec une fréquence donnée en paramètres (toutes les N_{output} itérations).

L'ensemble des paramètres physiques du problème (temps, coefficient de diffusion), les paramètres de discrétisation, ainsi que les options d'écriture des résultats seront tous lus depuis un fichier d'entrées.

Nous allons utiliser les classes suivantes :

- La gestion des paramètres se fera à l'aide de la classe `Parameters` qui a été vue dans un TP précédent.
- Une classe `Grid` contiendra les informations sur la grille cartésienne servant à discrétiser Ω.
- Une classe `LinearSolver` permettra de résoudre un système linéaire $Ax = b$ où A est creuse, soit par une factorisation LU, soit par un gradient conjugué (décidé *via* `Parameters`).
- Une classe `Assembler` permettra de construire la matrice et le second membre correspondant à la discrétisation de l'équation 1. L'assembleur contiendra les coefficients de la matrice et du second membre.
- Une classe `Writer` sera chargée d'écrire les fichiers de sortie.
- Enfin, une classe `HeatProblem` regroupera tous ces éléments au sein d'une même classe. Les tableaux contenant les solutions u^n et u^{n+1} seront contenus dans la classe. Ceux-ci seront des tableaux 1D dont l'indice sera calculé avec une numérotation classique ligne par ligne de la grille cartésienne. Le point de grille (i, j) aura pour indice dans ce tableau $iN + j$.

La fonction `main` se contentera de créer une instance de la classe `Parameters` à partir d'un nom de fichier donné dans la ligne de commande, ainsi qu'une instance de `HeatProblem`. Ensuite, il n'y aura qu'à appeler une méthode `run` de la classe `HeatProblem`.

La classe `Grid`

Celle-ci aura pour attributs le nombre de points et l'espacement entre les points dans chaque direction.

1. Écrivez un constructeur qui prend une référence ou un pointeur vers une instance de `Parameters` pour initialiser la grille cartésienne.
2. Écrivez les méthodes `getSpacing(int dim)` et `getNbPoints(int dim)` qui retournent les caractéristiques de la grille dans chaque direction (x :`dim=0`, y :`dim=1`).
3. Écrire une méthode `coords(int i, int j, double&x , double& y)` qui modifie les arguments `x` et `y` aux coordonnées du point d'indices (i, j).
4. Écrire les méthodes `index1D(int i, int j)` et `index2D(int row, int& i, int& j)` qui calculent l'indice 1D correspondant aux indices de grille cartésienne (i, j) et *vice versa*.

1. Le format vtk est très répandu dans la communauté scientifique, dans la mesure où les fichiers sont assez facilement écrits et éventuellement binaires, ce qui permet un gain de place important. Documentation sur le format vtk "legacy" en suivant ce lien.

La classe `LinearSolver`

La classe ne contiendra pas la matrice A ni le second membre b du système $Ax = b$. Ceux-ci seront stockés dans la classe `Assembler`. Il y aura cependant un attribut `_solverType` qui déterminera le type de solveur (direct ou itératif) en fonction du fichier de paramètres.

1. Écrivez un constructeur avec `Parameters` en argument pour initialiser le solveur. La taille de la matrice ainsi que le type de solveur seront initialisés.
2. Reprendre l'algorithme de résolution par gradient conjugué des TPs précédents pour en faire une méthode `solveCG` de la classe.
3. Implémentez une méthode `solveLU` qui effectue la factorisation LU de A et ensuite résoud $LUx = b$. La factorisation étant l'opération la plus coûteuse et la matrice A ne changeant pas au cours du temps, les matrices L et U seront stockées dans la classe pour être réutilisées aux itérations suivantes. Note : ici la matrice est symmétrique définie positive. Il est donc possible d'effectuer une factorisation de Cholesky LL^T, qui est plus économe en stockage.
4. Implémentez une méthode `solve` qui sélectionne la bonne méthode en fonction du type de solveur.

La classe `Assembler`

La classe contiendra les coefficients de la matrice de discrétisation (stockés à votre convenance), ainsi que les coefficients du second membre.

1. Le constructeur de cette classe prendra une instance de `Grid` en argument. La classe contiendra un pointeur vers cette instance, ainsi qu'un pointeur vers la méthode `double sourceTerm(double t, double x, double y)` qui sera membre de la classe `HeatProblem`, et qu'il faudra également passer en argument.
2. Écrire la méthode `assembleMatrix` qui calcule les coefficients de la matrice. Cette méthode ne sera appelée qu'une seule fois, dans la mesure où la matrice ne change pas au cours de la résolution du problème.
3. Écrire la méthode `assembleRHS` qui calcule le second membre. Cette méthode sera appelée à chaque itération en temps dans la mesure où le second membre contient u^n, et où le terme source varie lui aussi en fonction du temps.

La classe `Writer`

La classe contiendra des pointeurs vers des instances de `Parameters` et `Grid` passées en arguments du constructeur. La classe aura pour attribut la base des noms de fichiers ainsi qu'un compteur du nombre de fichiers écrits. Les fichiers de sortie auront pour nom `baseNameXXX.vtk` où XXX sera un entier de trois chiffres de long (000, 001, 002, etc).

1. Écrire une méthode privée `getFileName` qui retourne le nom du fichier à écrire à partir du compteur et du nom de base. Vous pourrez utiliser la classe `stringstream` de la bibliothèque standard (`# include <sstream>`), ainsi que la fonction `std::setw` (`# include <iomanip>`).
2. Inclure et adapter la méthode privée `writeHeader` qui sert à écrire l'en-tête du fichier de sortie

```
void Writer::writeHeader() {

  std::ofstream of(getFileName().c_str());
  int M = _grid->getNbPoints(0);
  int N = _grid->getNbPoints(1);
```

```
  of << "# vtk DataFile Version 3.0" << std::endl;
  of << "Test heat equation" << std::endl;
  of << "ASCII" << std::endl;
  of << "DATASET STRUCTURED_POINTS" << std::endl;
  of << "DIMENSION" << M << " " << N << " 1" << std::endl;
  of << "ORIGIN 0. 0. 0." << std::endl;
  of << "SPACING" << _grid->getSpacing(0) << " " << _grid->getSpacing(1) << " 0." << std::endl
     ;
  of << "POINT_DATA " << M*N << std::endl;
  of << "SCALARS u double" << std::endl;
  of << "LOOKUP_TABLE default" << std::endl;

}
```

3. Écrivez la méthode `write(std::vector<double>&)` qui génère le fichier de sortie avec les valeurs du tableau donné en argument. Cette méthode commencera par vérifier si la taille du tableau est compatible et appellera ensuite `writeHeader`. Les valeurs de la solution sont à écrire à la suite du fichier. Pour écrire à la suite dans un fichier déjà existant, il faut ajouter un argument à l'ouverture du flux de sortie :

   ```
   std::ofstream of;
   of.open("myFile",std::ios_base::app);
   ```

   Enfin, n'oubliez pas d'incrémenter le compteur de fichiers de la classe une fois l'écriture terminée !

La classe `HeatProblem`

1. Écrivez un constructeur qui prend une instance de `Parameters` en argument et initialise un pointeur vers celle-ci, pointeur qui sera un attribut de la classe et servira à accéder aux paramètres.
2. Écrivez une fonction membre `startUp` qui effectue les opérations nécessaires avant toute itération :
   - Lecture des paramètres de temps (t_0, t_{final}, δt) et du coefficient de diffusion k (qui doit être strictement positif).
   - Création des instances de `Grid`, `Assembler`, `LinearSolver` et `Writer` qui seront des attributs de la classe.
   - Allocation et initialisation des tableaux pour la solution.
   - Écriture de la solution à t_0.
3. Écrivez une fonction `sourceTerm(double t, double x, double y)` qui retourne la valeur du terme source $q(t, \boldsymbol{x})$.
4. Écrivez la méthode `run` qui appelera `startUp` et ensuite effectuera les itérations en temps. La méthode appellera la méthode `write` du `Writer` lorsque le nombre d'itérations effectuées est un multiple du paramètre donné dans le fichier d'entrée.

La fonction `main`

Écrivez la fonction `main` qui teste si l'utilisateur a bien donné un nom de fichier dans la ligne de commande, crée une instance de `Parameters` avec ce nom de fichiers, un `HeatProblem` et le fait tourner. Vous pourrez tester le programme pour différentes valeurs de k et observer le comportement de la solution. S'il vous reste du temps et de la curiosité, vous pouvez considérer k comme tensoriel. Il faudra cependant réécrire la discrétisation du terme $\nabla \cdot (K \nabla u)$ et modifier en conséquence la méthode d'assemblage de la matrice.