Mini-projet

Valeurs et vecteurs propres

But :

Calcul des valeurs/vecteurs propres associés à $A\mathbf{u} = \lambda\mathbf{u}$ dans $\mathbb{C}^N$

Le projet s'appuie sur les Travaux Dirigés précédents, en particulier celui correspondant aux systèmes linéaires, celui sur la manipulation des fichiers, et celui concernant la gestion de la mémoire *via* les pointeurs.

1 Travail demandé

Le programme final devra[1] gérer les différentes fonctionnalités suivantes via un menu :

- `calculer` la valeur propre de plus grand module (voir A) d'une matrice donnée (à coefficents compexes) de taille arbitraire. Cette fonctionnalité sera implémentée pour les matrices au format matrice pleine (voir B.1) et matrice creuse 2 uniquement.
- une fois le calcul fait, `sauvegarder` l'historique de convergence dans un fichier texte (ou binaire). L'historique de convergence (valeur propre, vecteur propre, "qualité", `...`, à l'itération k) sera géré par une liste chaînée.
- `lire/écrire` une matrice au format matrice creuse 1 (voir B.2) depuis/dans un fichier texte.
- `lire/écrire` une matrice au format matrice creuse 2 (voir B.2) depuis/dans un fichier binaire.
- `convertir` une matrice au format matrice pleine ou matrice creuse 1 en une matrice au format matrice creuse 2.

1. au moins

1.1 Étape 1

Dans un premier temps, nous travaillerons dans $\mathbb{R}^N$ au lieu de $\mathbb{C}^N$, et une précision de `float` sera suffisante.

L'algorithme de recherche de la valeur propre de plus grand module sera implémenté d'abord pour la représentation matrice pleine dans une version d'allocation *statique*. Puis ceci sera étendu à l'autre représentation matricielle.

Les fonctions de conversion et de sauvegarde seront faites une fois qu'une version de l'algorithme donnera des résultats cohérents.

1.2 Étape 2

Dans un second temps, le programme sera modifié pour gérer les matrices et les vecteurs de manière *dynamique*.

Si le temps le permet, le programme pourra travailler dans $\mathbb{C}^N$.

2 Travail à rendre

Un rapport de 5 pages [2] ***maximum*** sera rendu à la fin. Il détaillera les différents éléments du programme et donnera quelques résultats numériques pour illustrer les "performances" de l'implémentation.

Vous devez spécifier les fonctionnalités, ce qui fonctionne et ce qui ne fonctionne pas. Un organigramme présentant la structure du programme sera le bienvenu. Le but d'un tel organigramme est de faciliter la compréhension de la structure du programme. Nous accorderons de l'importance à la présentation, la lisibilité et la facilité à comprendre le code [3]. En ce qui concerne l'organisation du programme, nous attacherons de l'importance à l'utilisation de structures, de fichier entête (`*.h`) et d'une utilisation raisonnable de fonctions. Nous regarderons aussi la modularité du programme. L'élégance des solutions proposées, i.e la programmation, sera prise en compte mais ne sera pas déterminante.

Suggestion de Plan pour le rapport

1. Introduction - Présentation
2. Organigramme

2. tout compris sauf le listing du code qui sera mis en annexe
3. nous conseillons de relire les conseils donnés lors du TD numéro 1.

3. Explications des *solutions techniques* s'il ya lieu. Pourquoi avoir fait tel choix ou tel choix.
4. Listing (dans cet ordre au risque d'être pénalisé)
   (a) header (structures, déclaration des fonctions, ...)
   (b) la programme principal `main`
   (c) fonction (dans l'ordre où elles apparaissent dans le main)

A Algorithme : calculer *eigenpair* dominant

Pour trouver la valeur propre de plus grand module et son vecteur propre associé, nous utilisons la méthode itérative suivante :

$$\begin{array}{rcl} \text{Choisir} & & \mathbf{q}_0 \in \mathbb{C}^N \quad \text{tel que} \quad ||\mathbf{q}_0|| = 1 \\ \text{Puis} & & \text{faire pour } k = 1, 2, \ldots \\ \mathbf{x}_k & = & A\mathbf{q}_{k-1} \\ \mathbf{q}_k & = & \dfrac{\mathbf{x}_k}{||\mathbf{x}_k||} \end{array}$$

et nous avons le résultat suivant :

$$\begin{array}{rcl} \lim\limits_{k\to\infty} ||A\mathbf{q}_k|| & = & |\lambda_1| \\ \lim\limits_{k\to\infty} \dfrac{\mathbf{x}_{k+1}(j)}{\mathbf{q}_k(j)} & = & \lambda_1 \quad \text{pour } 1 \leq j \leq N \quad \text{si} \quad \mathbf{q}_k(j) \neq 0 \\ \lim\limits_{k\to\infty} \left(\dfrac{\overline{\lambda_1}}{|\lambda_1|}\right)^k \mathbf{q}_k & = & \mathbf{u}_1 \end{array}$$

Ainsi,

$$A\mathbf{u}_1 = \lambda_1 \mathbf{u}_1 \tag{1}$$

où λ_1 est la valeur propre de plus grand module et $\mathbf{u}_1$ son vecteur propre (à droite) associé.

B Représentation des matrices

Les matrices peuvent être *informatiquement* représentées de plusieurs façons. La représentation dépend souvent de l'application visée, c'est à dire de la structure des matrices. Par exemple, pour optimiser le coût mémoire, nous aurions intérêt à ne stocker que la moitié qu'une matrice symmétrique.

B.1 Représentation Pleine (*full matrix*)

La représentation pleine est la représentation la plus *classique*. La matrice est stockée en mémoire sous la forme d'un tableau à deux dimensions.

Une solution pour représenter *statiquement* la matrice en `C/C++` est d'utiliser une structure comme le Listing 1.

Listing 1 – Illustration matrice pleine statique

```
1 typedef struct
2 {
3 int n;
4 float coeff[100][100];
5 }matricepl;
```

Le problème rédhibitoire de cette représentation est l'allocation statique. De la mémoire est réservée même si elle n'est pas utilisée, i.e., que dans ce cas un tableau de taille 10^4 est alloué en mémoire alors que par exemple nous ne travaillons qu'avec une matrice de taille $N = 5$ et nous utilisons alors qu'un tableau de taille 25. Une gestion *dynamique* de la mémoire permet de lever ce problème.

Toutefois, si la matrice comporte beaucoup d'éléments nuls (comme c'est souvent le cas dans les méthodes numériques type Différences Finies, Éléments Finis, ...), ces zéros sont stockés inutilement. Ils engendrent un coût numérique (mémoire et opératoire) non-négligeable.

B.2 Représentation Creuse (*sparse matrix*)

Dans la représentation creuse, seuls les coefficients non-nuls sont stockés. On notera N_z (ou `NZ`) le nombre d'éléments non-nuls ($N_z \leq N^2$). Deux représentations peuvent être imaginées.

Représentation 1 : dans ce cas, 3 tableaux de taille N_z sont créés : dans le premier est stocké l'indice de ligne i, dans le second est stocké l'indice de colonne j et dans le troisième est stocké la valeur A_{ij}.

Un exemple pour représenter *statiquement* la matrice serait le Listing 2.

Listing 2 – Illustration matrice creuse 1 statique

```
1  typedef struct
2  {
3  int i , j ;
4  float coeff;
5  }coeffsp1;
6
7  typedef struct
8  {
9  int n;
10 int NZ;
11 coeffsp1 coef[100];
12 }matricesp1;
```

Bien évidemment, une gestion dynamique de la mémoire semble beaucoup plus judicieuse.

Bien entendu, on peut imaginer des choses très similaires pour représenter les vecteurs, mais il faut toutefois remarquer que le stockage en mémoire ainsi que leur manipulation n'est pas le facteur limitant.

Représentation 2 : dans ce cas, 3 tableaux sont aussi crées. Ils sont construits comme suit :

1. un premier tableau contenant les valeurs A_{ij} stockées ligne par ligne, de la ligne 1 à la ligne N. La taille de tableau est bien évidemment N_z.
2. un second tableau contenant les indices de colonnes correspondant aux éléments A_{ij}, rangés comme dans le tableau précédent
3. un troisième tableau de taille $N + 1$ noté `II` contenant la position du début de chaque ligne dans les deux tableaux précédents (`II[i]` donne la position du premier éléments non-nul de la $i$ième ligne). Le dernier éléments `II[n+1]`de ce tableau contient le nombre `II[1]+NZ`.

Un exemple pour représenter *statiquement* la matrice serait le Listing 3.

TABLE 1 – Exemple matrice Creuse ($N = 5$, $N_z = 12$)

$$A = \begin{pmatrix} 1.1 & 0. & 0. & 4. & 0. \\ 5. & 2.2 & 0. & 7. & 0. \\ 6. & 0. & 3.3 & 8. & 9. \\ 0. & 0. & 11. & 10.1 & 0. \\ 0. & 0. & 0. & 0. & 12.7 \end{pmatrix}$$

i	1	2	1	2	2	4	4	5	3	3	3	3
j	1	1	4	2	4	3	4	5	1	4	3	5
coeff	1.1	5.	4.	2.2	7.	11.	10.	12.7	6.	8.	3.3	9.

vals	1.1	4.	5.	2.2	7.	6.	3.3	8.	9.	11.	10.1	12.7
j	1	4	1	2	4	1	3	4	5	3	4	5
II	1	3	6	10	12	13						

Listing 3 – Illustration matrice creuse 2 statique

```
1 typedef struct
2 {
3 int n;
4 int NZ;
5 float vals[100];
6 int j[100];
7 int II[11];
8 }matricesp2;
```

Les deux paramètres `n` sont *pratiques* mais ils ne sont pas nécessaires. Á partir des autres paramètres, la taille pourrait se déduire [4].

L'exemple présenté à la TABLE 1 illustre ces deux représentations.

4. ceci vaut pour les deux représentations creuses

B.3 Produit Matrice-vecteur

Dans cette partie, nous donnons à titre d'exemple illustratif quelques portions de code pour calculer le produit matrice-vecteur dans la représentation creuse 2.

Listing 4 – Illustration produit matrice-vecteur creux 2 statique

```
1  vecteur operator*(matricesp2 A , vecteur x)
2  {
3  vecteur y;
4  y.n=x.n;
5    for(int i=0;i<A.n;i++)
6    {
7      y.coeff[i]= 0.0;
8      for (int j=A.II[i];j<A.II[i+1]-1;j++)
9      {
10       y.coeff[i] += A.vals[j] * x.coeff[ A.j[j] ] ;
11   }
12 return y;
13 }
```

Si on note N_z le nombre d'éléments non-nuls de la matrice A, il est clair que le nombre d'opétrations pour calculer le produit matrice-vecteur en représentation creuse 2 est de N_z (au lieu de N^2 pour le produit matrice-vecteur en représentation pleinne) [5].

5. pour une matrice ne possédant aucun élément nul, le coût calculatoire est identique, i.e., N^2 opérations. Toutefois, le stockage coûtera beaucoup plus cher.