Table des matières

1 Matrices pleines (*full matrices*)

2 Matrices creuses (*sparse matrices*)

En résolvant numériquement des EDPs par des méthodes de type différences finies, éléments finis et/ou volumes finis, on abouti souvent à la résolution de grands systèmes linéaires creux (i.e. les matrices des systèmes linéaires contiennent un très grand nombre de zéros). Au niveau mémoire (de l'ordinateur), il est alors recommandé (voir nécessaire) d'utiliser un stockage particulier pour ces matrices : CSC, CSR, COO, LIL, DOK, ... Ces différentes formats permettent de ne stocker en mémoire que les éléments non nuls des matrices.

Sous Matlab/Octave, le format utilisé est le format CSC (Compressed Sparse Column). Il est très efficace pour les operations arithmétiques et les produits matrices vecteurs. Nous donnons deux utilisations (parmis d'autres) de la fonction **sparse** de Matlab/Octave :

- `M = sparse(m,n);`

  Cette commande crée la matrice creuse nulle `M` de dimension `m`×`n`
- `M = sparse(I,J,K,m,n);`

  Cette commande crée la matrice creuse `M` de dimension `m`×`n` telle que

  $$\texttt{M(Ig(k),Jg(k))} \leftarrow \texttt{M(Ig(k),Jg(k))} + \texttt{Kg(k)}.$$

  Les vecteurs `Ig`, `Jg` et `Kg` ont la même longueur. Les éléments nuls de `Kg` ne sont pas pris en compte et les éléments de `Kg` ayant les mêmes indices dans `Ig` et `Jg` sont sommés.

On donne maintenant dans plusieurs langages interprétés l'analogue de ces deux commandes :

- Python (*scipy.sparse* module) :
  — `M=sparse.<format>_matrix((m,n))`
  — `M=sparse.<format>_matrix((Kg,(Ig,Jg)),shape=(m,n))`
  où `<format>` est le format de stockage de la matrice creuse (i.e. `csc`, `csr`, `lil`, ...),
- R : format `CSC` (seulement ?),
  — `M=sparse(i=,j=,dims=c(m,n))`
  — `M=sparse(i=Ig,j=Jg,x=Kg,dims=c(m,n))`
- Scilab : format `row-by-row`(?)
  — `M=sparse([],[],[m,n])`
  — `M=sparse([Ig,Jg],Kg,[m,n])`

Dans tous ces langages, de très nombreuses fonctions existent permettant d'utiliser les matrices creuses comme des matrices *classiques*. Il est donc possible d'effectuer des opérations aussi élémentaires que des sommes, produits de matrices et vecteurs mais aussi des opérations plus complexes comme des factorisations (LU, Cholesky,QR, ...), résolution de systèmes linéaires (méthodes directes ou itératives), calcul de valeurs propres et vecteurs propres, ...

Pour les langages de programmation usuels (Fortan, C, C++, ...) des libraires existent permettant de réaliser ces mêmes opérations sur les matrices creuses :

- C/C++ avec les librairies *SuiteSparse* de T. Davis [**?**] ainsi que *BLAS* (Basic Linear Algebra Subroutines), *LAPACK* (Linear Algebra PACKage) et leurs dérivées.
- CUDA avec la librairie *cuSparse* [**?**] de Nvidia.

Nous avons commencé à utiliser les matrices creuses (*sparse matrix*) dans les TPs précédents. Par exemple,

pour générer la matrice $\mathbb{K} \in \mathcal{M}_d(\mathbb{R})$ définie par

$$\mathbb{K} = \begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \tag{2.1}$$

nous avons écrit une fonction `Lap1D` pouvant ressembler à ceci :

Listing 1 – Fonction `Lap1D`

```
function K=Lap1D(d)
  K=sparse(d,d);
  K(1,1)=-2;K(1,2)=1;
  for i=2:d-1
    K(i,i)=-2;
    K(i,i-1)=1;
    K(i,i+1)=1;
  end
  K(d,d)=-2;K(d,d-1)=1;
end
```

Toutefois cette manière de construire une matrice creuse est très loin d'être la plus efficace.

Dans un premier temps, une rapide explication du pourquoi vous est proposée et ensuite nous verrons comment améliorer celà en utilisant judicieusement la fonction **sparse** . Il est aussi possible d'utiliser conjointement les fonctions **speye** et **spdiags** pour créer la matrice $\mathbb{K}$ mais ces commandes ne seront plus utilisables pour assembler les matrices provenant des méthodes d'éléments finis et volumes finis, c'est pourquoi nous nous focaliserons sur la fonction **sparse** .

2.1 Stockage

Sous Matlab et Octave, une matrice creuse (ou *sparse matrix*), $\mathbb{A} \in \mathcal{M}_{M,N}(\mathbb{R})$, est stockée sous le format CSC (Compressed Sparse Column), utilisant trois tableaux :

$$ia(1 : nnz),\ ja(1 : N + 1) \text{ et } aa(1 : nnz),$$

où nnz est le nombre d'éléments non nuls de la matrice $\mathbb{A}$. Ces tableaux (cachés mais contenus dans l'object **sparse** Matlab/Octave) sont définis par

- aa : contient l'ensemble des nnz éléments non nuls de $\mathbb{A}$ stockés par colonnes.
- ia : contient les numéros de ligne des éléments stockés dans le tableau aa.
- ja : permet de retrouver les éléments d'une colonne de $\mathbb{A}$, sachant que le premier élément non nul de la colonne k de $\mathbb{A}$ est en position $ja(k)$ dans le tableau aa. On a $ja(1) = 1$ et $ja(N + 1) = nnz + 1$.

Pour illustrer, voici un exemple simple avec la matrice

$$\mathbb{A} = \begin{pmatrix} 1. & 0. & 0. & 6. \\ 0. & 5. & 0. & 4. \\ 0. & 1. & 2. & 0. \end{pmatrix}$$

on a $M = 3$, $N = 4$, $nnz = 6$ et

aa	1.	5.	1.	2.	6.	4.
ia	1	2	3	3	1	2
ja	1	2	4	5	7	

Le premier élément non nul de la colonne $k = 3$ de $\mathbb{A}$ est 2, ce nombre est en position 4 dans aa, donc $ja(3) = 4$.

2.2 Occupation mémoire

Pour illustrer le gain mémoire, nous allons calculer la taille mémoire occupée par la matrice tridiagonale $\mathbb{K}$ définie en (2.1) dans le cas d'un stockage plein (*full matrix*) ou creux (*sparse matrix*). Cette matrice contient $d + 2(d-1) = 3d - 2$ éléments non nuls.

Sous Matlab/Octave le stockage creux de cette matrice va utiliser $(3d-2)$ `double` (tableau aa), $(3d-2)$ `int` (tableau ia) et $(d+1)$ `int` (tableau ja) pour un total de

$$(3d-2)\ \texttt{double} \quad \text{et} \quad (4d-1)\ \texttt{int}.$$

Or un `double` occupe 8 `Octets` (en anglais *bytes*) et un `int` 4 `Octets`. La place mémoire occupée par la matrice stockée sous forme `sparse` est alors de

$$(3d-2) \times 8 + (4d-1) \times 4 = (40\,d - 20)\ \texttt{Octets}.$$

Par contre, si la matrice était stockée classiquement (`full`), alors il y a d^2 `double` qui occupe en mémoire :

$$8d^2\ \texttt{Octets}.$$

On rappelle que 1 `Mo` (Méga-octet) correspond à 10^6 `Octets`, 1 `Go` (Giga-octet) correspond à 10^9 `Octets` et 1 `To` (Tera-octet) correspond à 10^{12} `Octets`. On donne maintenant la place mémoire occupée par la matrice suivant sa dimension et son stockage (`full` ou `sparse`) :

d	**full** matrix	**sparse** matrix
10^3	8 Mo	40 Ko
10^4	800 Mo	400 Ko
10^5	80 Go	4 Mo
10^6	8000 Go	40 Mo
10^7	800 To	400 Mo

2.3 Insertion d'éléments

Regardons les opérations à effectuer sur les tableaux aa, ia et ja si l'on modifie la matrice $\mathbb{A}$ par la commande

```
A(1,2)=8;
```

La matrice devient alors

$$\mathbb{A} = \begin{pmatrix} 1. & 8. & 0. & 6. \\ 0. & 5. & 0. & 4. \\ 0. & 1. & 2. & 0. \end{pmatrix}.$$

Dans cette opération un élément nul de $\mathbb{A}$ a été remplacé par une valeur non nulle 8 : il va donc falloir la stocker dans les tableaux sachant qu'aucune place n'est prévue. On suppose que les tableaux sont suffisamment grands (pas de souci mémoire), il faut alors décaler d'une case l'ensemble des valeurs des tableaux aa et ia à partir de la 3ème position puis copier la valeur 8 en $aa(2)$ et le numéro de ligne 1 en $ia(2)$:

aa	1.	8.	5.	1.	2.	6.	4.

ia	1	1	2	3	3	1	2

Pour le tableau ja, il faut à partir du numéro de colonne 2 plus un, incrémenter de +1 :

ja	1	2	5	6	8

La répétition de ces opérations peut devenir très coûteuse en temps CPU lors de l'assemblage de matrices utilisant des méthodes similaires à celle utilisée en Listing 1 (et on ne parle pas ici des problèmes de réallocation dynamique qui peuvent arriver !).

2.4 Construction efficace de matrices creuses

Pour générer efficacement une matrice creuse, il faut donc éviter d'insérer de manière répétitive des éléments dans celle-ci. Pour celà, on va utiliser l'appel suivant de la fonction `sparse` :

```
M = sparse(I,J,K,m,n);
```

Cette commande génère une matrice creuse `m` par `n` telle que `M(I(k),J(k)) = K(k)` . Les vecteurs `I` , `J` et `K` ont la même longueur. Il faut noter que tous les éléments nuls de `K` sont ignorés et que tous les éléments de `K` ayant les mêmes indices dans `I` et `J` sont sommés. Cette sommation est très utile lors de l'assemblage de matrices obtenues par une méthode de type éléments finis.

Par exemple, voici deux commandes permettant de générer la matrice $\mathbb{A}$ précédente :

```
A = sparse([1 1 2 3 3 1  2],[1  2 2 2 3 4  4],[1  8 5 1 2 6  4],3,4) ;
A = sparse([1 2 1 1 2 3  3],[4  4 1 2 2 2  3],[6  4 1 8 5 1  2],3,4) ;
```

Il existe aussi la fonction `A = spalloc(m,n,nnz)` qui préalloue une matrice creuse de dimension `m` par `n` avec au plus `nz` éléments non nuls (*nonzero elements*).

Nous allons maintenant utiliser ceci dans un cadre un peu général : les matrices tridiagonales.

2.5 Matrices diagonales et tridiagonales

Q. 1 (Matlab) *Ecrire la fonction* `spMatDiag` *permettant à partir d'un vecteur* $\boldsymbol{v} \in \mathbb{R}^d$ *de retourner la matrice diagonale (creuse)* $\mathbb{D} \in \mathcal{M}_d(\mathbb{R})$ *de diagonale* $\boldsymbol{v}$ *(i.e.* $\mathbb{D}(i,i) = \boldsymbol{v}(i)$, $\forall i \in [\![1,d]\!]$*) en créant (sans boucle) les tableaux* `I`*,* `J` *et* `K` *puis en générant la matrice* $\mathbb{D}$ *à l'aide de la commande* `D = sparse(I,J,K,d,d);` ▪

Soit $\mathbb{A} \in \mathcal{M}_d(\mathbb{R})$ la matrice tridiagonale définit par

$$\mathbb{A} = \begin{pmatrix} v_1 & w_1 & 0 & \cdots & \cdots & 0 \\ u_1 & v_2 & w_2 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & v_{d-1} & w_{d-1} \\ 0 & \cdots & \cdots & 0 & u_{d-1} & v_d \end{pmatrix} \tag{2.2}$$

avec $\boldsymbol{u} \in \mathbb{R}^{d-1}$, $\boldsymbol{v} \in \mathbb{R}^d$ et $\boldsymbol{w} \in \mathbb{R}^{d-1}$.

Q. 2 (Matlab)

1. *Ecrire la fonction* `spMatTriDiagSca` *permettant à partir des vecteurs* $\boldsymbol{u}$, $\boldsymbol{v}$ *et* $\boldsymbol{w}$ *de retourner la matrice creuse* $\mathbb{A}$ *définie en* (2.2)*. Pour celà on va tout d'abord créer une matrice creuse de* $\mathcal{M}_d(\mathbb{R})$ *puis on va la «remplir» à l'aide d'une boucle* **for** *sur le principe du Listing 1.*
2. *Ecrire la fonction* `spMatTriDiagVec` *permettant à partir des vecteurs* $\boldsymbol{u}$, $\boldsymbol{v}$ *et* $\boldsymbol{w}$ *de retourner la matrice creuse* $\mathbb{A}$ *en créant (sans boucle) les tableaux* `I`*,* `J` *et* `K` *puis en générant la matrice* $\mathbb{A}$ *à l'aide de la commande* `A = sparse(I,J,K,d,d);`
3. *Ecrire un programme permettant de mesurer (tic-toc) puis de représenter graphiquement les performances de* `spMatTriDiagSca` *et* `spMatTriDiagVec` *en fonction de* d*.*

2.6 Produit de Kronecker de deux matrices

Pour générer efficacement et simplement les matrices creuses obtenus par des méthodes de différences finies en dimension $\geqslant 2$ nous pouvons utiliser le produit de Kronecker de deux matrices.

Soient $\mathbb{A} \in \mathcal{M}_{m,n}(\mathbb{R})$ et $\mathbb{B} \in \mathcal{M}_{p,q}(\mathbb{R})$. Le produit tensoriel de Kronecker de $\mathbb{A}$ par $\mathbb{B}$, noté $\mathbb{A} \otimes \mathbb{B}$, est la matrice de $\mathcal{M}_{mp,nq}(\mathbb{R})$ définie avec des blocs de dimension $p \times q$ par

$$\mathbb{A} \otimes \mathbb{B} = \begin{pmatrix} \mathrm{A}_{1,1}\mathbb{B} & \cdots & \mathrm{A}_{1,n}\mathbb{B} \\ \vdots & \ddots & \vdots \\ \mathrm{A}_{m,1}\mathbb{B} & \cdots & \mathrm{A}_{m,n}\mathbb{B} \end{pmatrix} \tag{2.3}$$

Le produit de Kronecker est bilinéaire et associatif : Si les dimensions des matrices $\mathbb{A}$, $\mathbb{B}$ et $\mathbb{C}$ sont compatibles on a $\forall \lambda \in \mathbb{K}$

$$\begin{aligned}
\mathbb{A} \otimes (\mathbb{B} + \lambda \ \cdot \mathbb{C}) &= (\mathbb{A} \otimes \mathbb{B}) + \lambda(\mathbb{A} \otimes \mathbb{C}) \\
(\mathbb{A} + \lambda \ \cdot \mathbb{B}) \otimes \mathbb{C} &= (\mathbb{A} \otimes \mathbb{C}) + \lambda(\mathbb{B} \otimes \mathbb{C}) \\
\mathbb{A} \otimes (\mathbb{B} \otimes \mathbb{C}) &= (\mathbb{A} \otimes \mathbb{B}) \otimes \mathbb{C}
\end{aligned}$$

Par contre, il n'est pas commutatif.

Nous allons écrire une fonction permettant d'effectuer ce produit. Cette fonction existe déjà nativement mais il est instructif de savoir l'implémenter. Mais auparavant, quelques petits rappels (ou non) des possibilités offertes par le langage Matlab.

Il est facile sous Malab/Octave de modifier une matrice (creuse ou non). Par exemple,

- `A(I,J)=M`

  Les tableaux d'indices `I` et `J` sont de dimension respective n et m. La matrice `M` est de dimension n-par-m. La matrice `A` est modifiée de telle sorte que

  $$\texttt{A(I(i),J(j))=M(i,j)},\ \forall \texttt{i} \in [\![1,n]\!],\ \forall \texttt{j} \in [\![1,m]\!]$$

- `A(I,J)=A(I,J)+M`

  Même chose en sommant.
- `A(i,:)=a`

  Ici `a` est un scalaire. Tous les éléments de la ligne `i` de la matrice `A` valent `a`.
- `A(:,j)=a`

  Ici `a` est un scalaire. Tous les éléments de la colonne `j` de la matrice `A` valent `a`.
- ...

Sous Matlab/Octave, il est possible de récupérer tous les éléments non nuls d'une matrice creuse `A` ainsi que leurs indices de ligne et de colonne à l'aide de la commande

```
[I,J,K]=find(A);
```

Ici les trois tableaux `I`, `J` et `K` ont même longueur et on a `A(I(k),J(k)) == K(k)` pour tout `k` inférieur ou égal à `length(K)`.

Q. 3 (Matlab) *Ecrire la fonction* `spMatKron` *permettant à partir deux matrices creuses carrées* $\mathbb{A} \in \mathcal{M}_n(\mathbb{R})$ *et* $\mathbb{B} \in \mathcal{M}_m(\mathbb{R})$ *de retourner la matrice creuse* $\mathbb{C} = \mathbb{A} \otimes \mathbb{B}$. *Pour celà, on initialisera la matrice retournée* `C` *à l'aide de la commande* `C=sparse(n*m,n*m)` *puis on utilisera uniquement une boucle sur les éléments non-nuls de la matrice* $\mathbb{A}$ *pour «remplir» la matrice* `C`. ▪

On pourra comparer cette fonction à la fonction `kron` de Matlab/Octave.