



université PARIS-SACLAY

TD/TP5 de Calcul numérique Méthodes itératives de base

Penda THIAO

08/01/2025

1. Travail préliminaire : Etablissement d'un cas de test

Exercice 1

Résolution de l'équation de la chaleur

Discrétisation et équation discrète

Le domaine $[0, 1]$ est discrétisé selon $n + 2$ noeuds x_i pour $i = 0, 1, 2, \dots, n + 1$, espacés d'un pas h constant. L'équation de la chaleur discrétisée en chaque noeud est obtenue par une approximation centrée de la dérivée seconde :

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2}$$

Ainsi, l'équation discrète devient :

$$-k \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} = g_i$$

ou, en réarrangeant l'expression :

$$k \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} = g_i$$

Ce système peut être écrit sous la forme matricielle :

$$Au = f$$

où $A \in \mathbb{R}^{n \times n}$ est la matrice du système, u est le vecteur des températures discrètes T_i , et f est le vecteur des termes sources g_i .

Cas sans source de chaleur

Dans la suite du travail pratique, on considère le cas où il n'y a pas de source de chaleur, c'est-à-dire $g_i = 0$ pour tout i . Dans ce cas, l'équation discrétisée devient :

$$k \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} = 0$$

Ce qui peut être réarrangé pour donner une équation linéaire pour chaque T_i :

```

root@caa673dcflc9:/app# make run
bin/tp_testenv
----- Test environment of execution for Practical exercises of Numerical Algorithmics -----

The exponential 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

```

Figure 1: Test d'environnement

$$T_{i+1} - 2T_i + T_{i-1} = 0$$

Cela représente un système tridiagonal qui peut être résolu par des méthodes numériques appropriées.

Solution analytique

La solution analytique de l'équation de la chaleur sans terme source ($g = 0$) est donnée par la solution de l'équation différentielle :

$$-k \frac{\partial^2 T}{\partial x^2} = 0$$

La solution générale de cette équation est une fonction affine $T(x) = Ax + B$, où A et B sont des constantes à déterminer en utilisant les conditions aux bords. En appliquant les conditions $T(0) = T_0$ et $T(1) = T_1$, on obtient :

$$T(x) = T_0 + x(T_1 - T_0)$$

Exercice2

En configurant l'environnement de travail dans un conteneur Docker, les bibliothèques CBLAS et LAPACK ont été intégrées pour être reconnues lors de la compilation. Un fichier de test a été créé et exécuté, donnant le résultat suivant :

2. Méthode directe et stockage bande

Exercice3

1. Déclaration et Allocation d'une Matrice en C pour BLAS et LAPACK

Pour utiliser BLAS et LAPACK en C, les matrices doivent être déclarées comme des tableaux en mémoire contiguë (1D), car ces bibliothèques s'attendent à des données linéarisées.

Exemple de déclaration et allocation :

```
int rows = 3; // Nombre de lignes
int cols = 4; // Nombre de colonnes

// Allouer un tableau 1D pour une matrice 3x4
double* matrix = (double*)malloc(rows * cols * sizeof(double));
```

Accès aux éléments :

```
// En supposant un stockage row-major :
matrix[i * cols + j] = valeur; // i : ligne, j : colonne
```

Remarque : BLAS et LAPACK utilisent le format **colonne-major** par défaut, où les éléments des colonnes consécutives sont contigus en mémoire.

2. Signification de la Constante LAPACK_COL_MAJOR

La constante `LAPACK_COL_MAJOR` est utilisée dans l'interface C de LAPACK (`LAPACKE`) pour indiquer que les matrices sont stockées en **ordre colonne-major**, comme dans Fortran.

Stockage colonne-major :

- Les éléments d'une même colonne sont stockés consécutivement en mémoire.
- Exemple pour une matrice A 2×3 ($A[i][j]$) :

$$A = [1.0, 3.0, 2.0, 4.0, 5.0, 6.0]$$

Utilité : Cette constante garantit que la mémoire est organisée de manière compatible avec les routines LAPACK, qui utilisent par défaut le format colonne-major.

3. Dimension Principale (Leading Dimension, 1d)

La **dimension principale** (1d) spécifie la longueur physique (en mémoire) d'une matrice, même si sa taille logique est plus petite.

Définition

La dimension principale correspond au **nombre d'éléments alloués par colonne** pour une matrice en format colonne-major. Pour une matrice A de taille $m \times n$, $ld \geq m$.

Exemple

Soit une matrice logique A 3×3 , mais stockée dans un tableau 1D de taille 5×3 . Ici :

- $ld = 5$ (nombre d'éléments dans chaque colonne).
- Stockage en mémoire :

$$[a_{11}, a_{21}, a_{31}, -, -, a_{12}, a_{22}, a_{32}, -, -, a_{13}, a_{23}, a_{33}, -, -]$$

Utilisation dans BLAS/LAPACK

Lorsque vous appelez une routine, vous devez fournir 1d :

```
dgemm(LAPACK_COL_MAJOR, ..., A, ldA, ...);
```

4. Fonction gbmv

La fonction **dgbmv** effectue une multiplication matrice-vecteur pour une matrice générale bande (band matrix).

Description

Elle calcule :

$$y = \alpha \cdot A \cdot x + \beta \cdot y$$

Où :

- A est une matrice bande, spécifiée par ses diagonales stockées en mémoire contiguë.
- x et y sont des vecteurs.
- α et β sont des scalaires.

Méthode

La méthode utilisée est une multiplication optimisée en tenant compte de la structure bande de la matrice.

5. Fonction `dgbtrf`

La fonction `dgbtrf` réalise la factorisation LU d'une matrice bande générale.

Description

Elle décompose une matrice A sous la forme :

$$A = P \cdot L \cdot U$$

Où :

- P est une matrice de permutation.
- L est une matrice triangulaire inférieure avec des 1 sur la diagonale.
- U est une matrice triangulaire supérieure.

Méthode

La méthode utilisée est une adaptation de l'algorithme LU pour les matrices bande, en réduisant les opérations inutiles sur les zéros implicites.

6. Fonction `dgbtrs`

La fonction `dgbtrs` résout un système linéaire utilisant la factorisation LU obtenue avec `dgbtrf`.

Description

Elle résout :

$$A \cdot x = b$$

Où A est déjà factorisée sous la forme LU avec `dgbtrf`.

Méthode

Elle utilise une substitution directe pour résoudre les systèmes triangulaires inférieurs et supérieurs.

7. Fonction `dgbstv`

La fonction `dgbstv` combine `dgbtrf` et `dgbtrs` pour résoudre un système linéaire en une seule étape.

Description

Elle résout :

$$A \cdot x = b$$

Où A est une matrice bande.

Méthode

Elle effectue :

1. La factorisation LU de A (dgbtrf).
2. La résolution du système factorisé (dgbtrs).

8. Calcul de la Norme du Résidu Relatif avec BLAS

Pour calculer la norme du résidu relatif, on peut utiliser les fonctions BLAS comme suit :

Étapes

1. Calculer le résidu $r = b - A \cdot x$ avec `dgemv` pour le produit matrice-vecteur.
2. Calculer la norme de r et celle de b avec `dnrm2` :

$$\text{norme relative} = \frac{\|r\|_2}{\|b\|_2}$$

Exercice4. Stockage GB et appel à DGBMV

1. Question1

```
void set_GB_operator_colMajor_poisson1D(double *AB, int *lab, int *la, int *kv)
{
    int ii, jj, kk;
    // Boucle sur chaque colonne de la matrice AB
    for (jj = 0; jj < (*la); jj++)
    {
        kk = jj * (*lab); // Calcul de l'indice de début de la colonne jj dans AB
        // Si kv est supérieur ou égal à 0, initialiser les premiers éléments de la colonne à 0.0
        if (*kv >= 0)
        {
            for (ii = 0; ii < *kv; ii++)
            {
                AB[kk + ii] = 0.0;
            }
        }
        // Définir les valeurs pour la tridiagonale de la matrice du système de Poisson 1D
        AB[kk + *kv] = -1.0; // Élément de la sous-diagonale
        AB[kk + *kv + 1] = 2.0; // Élément de la diagonale principale
        AB[kk + *kv + 2] = -1.0; // Élément de la sur-diagonale
    }
    // Gérer les conditions aux frontières pour le premier et le dernier élément
    AB[0] = 0.0; // Premier élément de la première colonne
    // Si kv est égal à 1, définir le deuxième élément de la première colonne à 0
    if (*kv == 1)
    {
        AB[1] = 0.0;
    }
    // Dernier élément de la dernière colonne
    AB[(*lab) * (*la) - 1] = 0.0;
}
```

Figure 2: Stockage bande priorité colonne Poisson 1D

2. Question2

3.Question3

Pour valider notre stockage, on doit multiplier notre matrice AB par un vecteur unitaire $\mathbf{x} = (1, 1, 1, \dots, 1)^T$. La matrice AB en stockage bande s'écrit :

$$AB = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & -1 \\ 2 & 2 & 2 & 2 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

Le vecteur \mathbf{x} est donné par :

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

En effectuant le produit $\mathbf{y} = AB \cdot \mathbf{x}$, nous obtenons le vecteur attendu :

$$\mathbf{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Ce résultat valide que notre stockage en bande est correct.

Exercice5

1. Résolution du systeme linéaire avec lapack

```
/*Exercice 5*/
/* It can also be solved with dgbstv */
if (IMPLEM == SV) {
    // TODO : use dgbstv

    clock_t start, end;
    double time_used;
    start = clock();
    dgbstv_(&la,&kl,&ku,&NRHS,AB,&lab,ipiv,RHS,&la,&info);
    end = clock();

    time_used = ((double)(end-start))/CLOCKS_PER_SEC;
    printf("Time used = %f seconds\n",time_used);
}
```


Erreur Relative

```
Solution with LAPACK  
  
INFO = 30  
  
The relative forward error is relres = 6.454972e-01
```

2. Evaluation des performances

Pour évaluer les performances des fonctions **DGBTRF**, **DGBTRS**, et **DGBSV**, on peut mesurer le temps d'exécution à l'aide de fonctions de chronométrage telles que `clock()` ou `gettimeofday()` en C.

Complexité théorique

Pour une matrice bande de taille $n \times n$, avec :

- kl : nombre de diagonales inférieures,
 - ku : nombre de diagonales supérieures,
- la complexité des opérations est la suivante :

1. **DGBTRF** (Factorisation LU pour matrice bande) :

$$\mathcal{O}(n \cdot kl \cdot ku)$$

Cette méthode tire profit de la structure bande pour réduire la complexité par rapport à une matrice dense.

2. **DGBTRS** (Résolution après factorisation LU) :

$$\mathcal{O}(n \cdot kl \cdot ku)$$

3. **DGBSV** (Factorisation et résolution combinées) :

$$\mathcal{O}(n \cdot kl \cdot ku)$$

C'est essentiellement une combinaison de **DGBTRF** et **DGBTRS**, donc la complexité reste identique.

Temps d'exécution

```
----- END -----  
bin/tpPoisson1D_direct 2  
----- Poisson 1D -----  
  
Solution with LAPACK  
Time used = 0.000248 seconds
```

Exercice 6

1. Implémentation de la méthode de factorisation LU

Soit une matrice tridiagonale A de dimension $n \times n$ avec une bande de largeur 3 (diagonale principale, diagonale supérieure et diagonale inférieure). La factorisation LU consiste à décomposer cette matrice en un produit de deux matrices triangulaires : une matrice triangulaire inférieure L et une matrice triangulaire supérieure U , telles que $A = LU$.

La matrice A est stockée dans le format GB (General Band), ce qui permet de représenter efficacement les matrices avec une bande de non-zéros.

Algorithme de la factorisation LU

L'algorithme de factorisation LU pour une matrice tridiagonale peut être implémenté de manière suivante :

- On parcourt chaque ligne de la matrice et on effectue les étapes suivantes pour chaque élément i de la diagonale principale :
 1. Si l'élément $AB[i + LAB]$ (pivot) est nul, on renvoie un code d'erreur.
 2. On calcule le facteur de pivot $\text{facteur} = \frac{AB[i+LAB+1]}{AB[i+LAB]}$.
 3. On met à jour la diagonale supérieure en effectuant l'élimination avec la relation $AB[i + LAB + 2] = AB[i + LAB + 2] - \text{facteur} \times AB[i + LAB + 1]$.
 4. La diagonale inférieure est modifiée pour stocker le facteur calculé.
- À la fin, la matrice A est transformée en une matrice triangulaire inférieure L et une matrice triangulaire supérieure U .

3 Méthode de validation

La validation de la factorisation LU peut être effectuée de deux manières principales :

Reconstruction de la matrice

La première méthode consiste à reconstruire la matrice A en multipliant les matrices L et U , et vérifier si le produit correspond à la matrice originale.

- On effectue la multiplication $A = L \times U$.
- On compare les éléments de la matrice reconstruite avec ceux de la matrice initiale et on vérifie si les écarts sont suffisamment faibles (en utilisant une tolérance).

Vérification des résultats

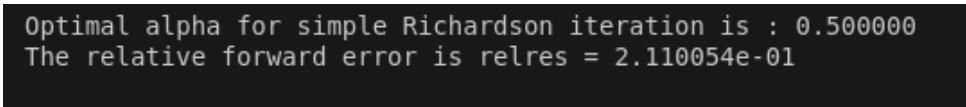
Une autre méthode consiste à comparer les résultats obtenus avec une bibliothèque éprouvée, comme LAPACK, pour effectuer la factorisation LU d'une matrice tridiagonale.

3. Méthode de résolution itérative

Exercice 7. Implémentation C - Richardson

Dans cet exercice, nous avons implémenté l'algorithme de Richardson en langage C, en utilisant des matrices au format GB (General Banded). Les étapes principales de l'implémentation sont les suivantes :

1. **Implémentation de l'algorithme de Richardson:** Nous avons développé une fonction en C qui exécute l'algorithme de Richardson pour résoudre un système linéaire. L'algorithme utilise des matrices stockées au format GB (General Banded), ce qui permet de gérer efficacement les matrices creuses avec des bandes diagonales. Dans notre implémentation, la fonction `richardson_MB()` initialise la solution à zéro, puis itère en ajustant cette solution en fonction du vecteur résidu. À chaque itération, le résidu est calculé et sauvegardé dans un vecteur pour une analyse ultérieure.
2. **Calcul de l'erreur:** Pour évaluer la précision de notre solution, nous avons calculé l'erreur par rapport à la solution analytique du système.



```
Optimal alpha for simple Richardson iteration is : 0.500000
The relative forward error is relres = 2.110054e-01
```

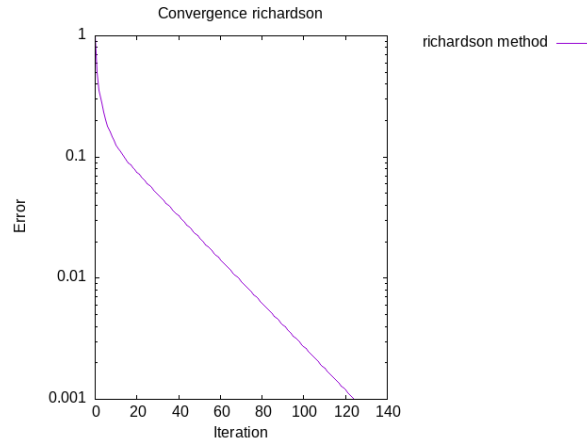
Figure 3: rreur par rapport à la solution analytique

3. **Analyse de la convergence:** Nous avons analysé la convergence de l'algorithme en traçant l'historique de convergence. Cela implique de suivre l'évolution du résidu à chaque itération et de le représenter graphiquement.

Exercice 8. Implémentation C - Jacobi

Dans cet exercice, nous avons implémenté la méthode de Jacobi en langage C pour résoudre des systèmes linéaires avec des matrices tridiagonales au format GB (General Banded). Les étapes principales de l'implémentation sont les suivantes :

1. **Implémentation de la méthode de Jacobi:** Nous avons développé une fonction en C qui implémente la méthode de Jacobi pour une matrice tridiagonale. La fonction `extract_MB_jacobi_tridiag()` initialise la matrice M utilisée dans l'algorithme de Jacobi. Cette fonction remplit la matrice M avec les inverses des



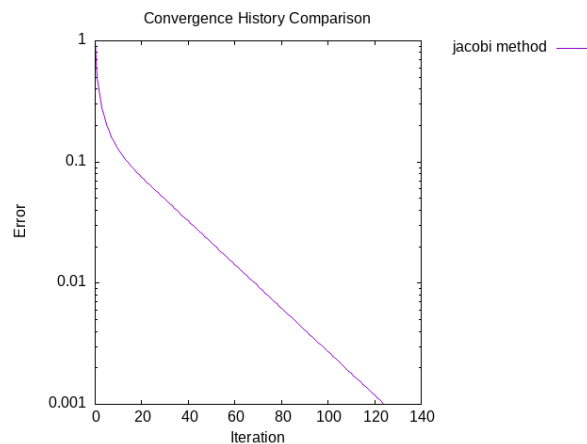
éléments diagonaux de la matrice tridiagonale AB , en laissant les autres éléments à zéro. Cette approche permet de gérer efficacement les matrices tridiagonales et de simplifier les calculs dans l'algorithme de Jacobi.

2. **Calcul de l'erreur:** Pour évaluer la précision de notre solution, nous avons calculé l'erreur par rapport à la solution analytique du système.

```
Optimal alpha for simple Richardson iteration is : 0.500000
The relative forward error is relres = 2.110054e-01
```

Figure 4: Erreur par rapport à la solution analytique

3. **Analyse de la convergence:** Nous avons analysé la convergence de l'algorithme en traçant l'historique de convergence.



Exercice 9. Implémentation C -Gauss-Seidel

1. **Implémentation de la méthode de Gauss-Seidel:** Nous avons développé une fonction en C, `extract_MB_gauss_seidel_tridiag()`, qui implémente la méthode de Gauss-Seidel pour une matrice tridiagonale. La fonction convertit la matrice tridiagonale AB en une matrice triangulaire inférieure $(D - E)$. Ensuite, elle calcule l'inverse de cette matrice triangulaire inférieure .

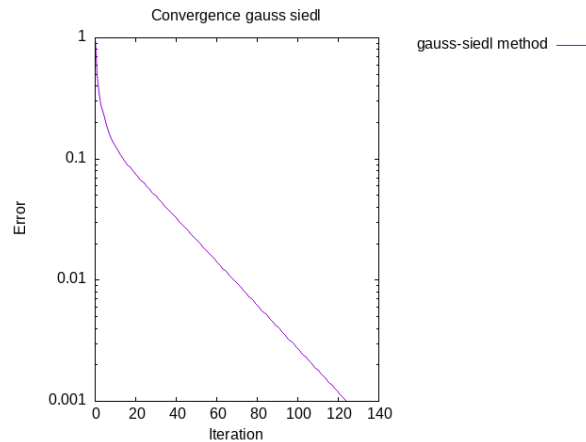
Choix pour le calcul de l'application de $(D-E)^{-1}$ au résidu r_k : Pour calculer l'application de $(D-E)^{-1}$ au résidu r_k , nous avons choisi de transformer la matrice tridiagonale en une matrice triangulaire inférieure $(D-E)$ et de calculer son inverse. Cette méthode permet de simplifier les calculs en utilisant la structure particulière des matrices tridiagonales, et assure que les opérations matricielles restent efficaces et stables. En inversant la matrice triangulaire inférieure, nous pouvons appliquer cette inverse au résidu r_k de manière itérative pour obtenir la solution mise à jour.

2. **Calcul de l'erreur:** Pour évaluer la précision de notre solution, nous avons calculé l'erreur par rapport à la solution analytique du système. Cette erreur est mesurée en utilisant la norme euclidienne entre la solution numérique obtenue et la solution analytique connue.

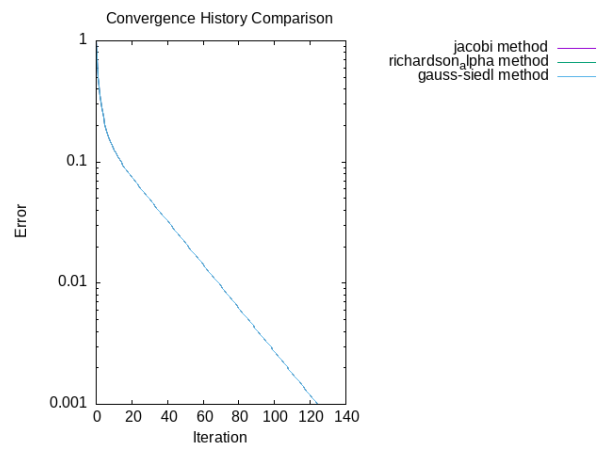
```
Optimal alpha for simple Richardson iteration is : 0.500000
The relative forward error is relres = 2.110054e-01
```

Figure 5: Erreur par rapport à la solution analytique

3. **Analyse de la convergence:** Nous avons analysé la convergence de l'algorithme en traçant l'historique de convergence. Cela implique de suivre l'évolution du résidu à chaque itération et de le représenter graphiquement.



Comparaison de différentes méthodes



4. Autres formats de stockage

Exercice 10. Format CSR / CSC

Les formats CSR (Compressed Sparse Row) et CSC (Compressed Sparse Column) sont des méthodes efficaces de stockage et de manipulation des matrices creuses, particulièrement utiles en calcul numérique et en algèbre linéaire. Une matrice creuse est une matrice principalement constituée de zéros, avec relativement peu d'éléments non nuls. Représenter ces matrices de manière dense (en stockant tous les zéros) serait inefficace en termes de mémoire et de calcul.

Format CSR (Compressed Sparse Row)

Le format CSR stocke uniquement les éléments non nuls d'une matrice et les indices de leurs colonnes. Il utilise trois tableaux principaux :

- **valeurs** : un tableau contenant les valeurs non nulles de la matrice.
- **indices_colonnes** : un tableau contenant les indices de colonne correspondants à chaque valeur non nulle.
- **pointeurs_lignes** : un tableau contenant les indices dans les tableaux **valeurs** et **indices_colonnes** où commence chaque ligne de la matrice.

Cette méthode permet un accès rapide aux éléments et une efficacité mémoire accrue.

Format CSC (Compressed Sparse Column)

Le format CSC est similaire au format CSR, mais il stocke les éléments par colonnes plutôt que par lignes. Il utilise également trois tableaux :

- **valeurs** : un tableau contenant les valeurs non nulles de la matrice.
- **indices_lignes** : un tableau contenant les indices de ligne correspondants à chaque valeur non nulle.
- **pointeurs_colonnes** : un tableau contenant les indices dans les tableaux **valeurs** et **indices_lignes** où commence chaque colonne de la matrice.

Le format CSC est particulièrement utile pour les opérations qui se déroulent naturellement par colonnes, comme certains algorithmes de multiplication matrice-vecteur.

Utilité des formats CSR et CSC

L'utilisation des formats CSR et CSC permet de réduire significativement la quantité de mémoire nécessaire pour stocker des matrices creuses et d'accélérer les opérations algébriques. Par exemple, ces formats sont couramment utilisés pour résoudre des systèmes linéaires issus de la discrétisation des équations aux dérivées partielles, comme le problème de Poisson en 1D. Ils permettent une manipulation efficace des matrices dans les applications de calcul scientifique, y compris la simulation numérique, l'optimisation, et la modélisation des réseaux.

En conclusion, les formats CSR et CSC sont des outils puissants pour le traitement des matrices creuses, offrant à la fois des gains en mémoire et des améliorations en performance computationnelle.