MPNA - Rendu des TPs

Yingqin SU

Les Méthodes et Programmation Numériques Avancées proposent de compléter – voire de remplacer – les approches analytiques traditionnelles par des techniques numériques innovantes, permettant d'obtenir des solutions approximatives mais remarquablement précises pour des équations ou systèmes dont la résolution exacte reste inatteignable.

March 12, 2025

Contents

0.1. Introduction

Ce rapport vise à compléter et illustrer les codes développés lors des différents travaux pratiques. Il s'articule autour de trois parties principales :

- **Première partie** : Implémentation d'une multiplication matrice-matrice en utilisant le stockage classique.
- **Deuxième partie** : Mise en œuvre du stockage de matrices au format CSR, accompagnée de plusieurs méthodes itératives de résolution (Jacobi, Gauss-Seidel, gradient conjugué et GMRES). L'objectif ici est d'évaluer le nombre d'itérations et le temps d'exécution pour chaque méthode. Pour vérifier la justesse des résolutions, une méthode de multiplication matrice-vecteur a également été implémentée, en se basant sur des données issues du site Matrix Market. [2]
- **Troisième partie** : Résolution d'un problème physique classique, celui de la diffusion dans une flamme, en exploitant la bibliothèque de calcul Hypre.

Les travaux pratiques permet ainsi d'illustrer l'efficacité et la pertinence des différentes techniques numériques pour la résolution de problèmes complexes.

0.2. Première partie : Multiplication Matrix-Matrix en dense

En général, nous stockons les matrices de manière dense, ce qui signifie qu'une matrice de dimensions n x n occupe n × n unités de stockage. Ainsi, la multiplication classique de deux matrices de taille n x n se réalise à l'aide d'une triple boucle imbriquée (i, j, k), comme illustré ci-dessous en C :

```c
void matrix_multiply(int n, double A[n][n], double B[n][n], double C[n][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0.0;  // Initialisation de C[i][j]
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];  // Accumulation du produit
            }
        }
    }
}
```

Il existe par ailleurs plusieurs techniques d'optimisation pour ce type de multiplication, telles que la permutation de l'ordre des boucles, l'utilisation d'instructions FMA, les opérations SIMD ou encore la transposition des matrices. Toutefois, ces optimisations ne seront pas abordées dans ce TP.

1. Deuxième partie : Format CSR

Dans cette section, nous explorons le stockage des matrices au format CSR, qui permet d'optimiser l'utilisation mémoire pour des matrices creuses. L'objectif est d'implémenter différentes méthodes de résolution pour des systèmes linéaires issus de ces matrices, en évaluant leur convergence et leur performance.

- Mise en place des matrices en format CSR
  - Conversion d'une matrice dense en format CSR pour exploiter efficacement les espaces mémoire et accélérer les calculs.
- Méthodes itératives : Jacobi et Gauss-Seidel
  - Déploiement des algorithmes itératifs classiques, tels que Jacobi et Gauss-Seidel, permettant d'obtenir des solutions approximatives tout en contrôlant les erreurs.
- Méthodes gradient : Gradient conjugué et GMRES
  - Mise en œuvre de méthodes basées sur les gradients (gradient conjugué et GMRES) pour résoudre les systèmes linéaires avec une approche plus robuste en termes de convergence.
- Matrix Vector Multiply
  - Développement d'une fonction de multiplication matrice-vecteur afin de vérifier la validité et la précision des résultats obtenus par les différentes méthodes.

Pour vérifier la validité de notre code, nous utiliserons les données publiques disponibles sur Matrix Market.[2]

1.1. Mise en place des matrices en format CSR

Pour la mise en place, nous utilisons le langage C++. La démarche est relativement simple : il suffit de définir une structure en C++ qui encapsule les informations nécessaires. Dans cette structure :

- **n** représente le nombre de lignes de la matrice.
- **row_ptr** est un tableau d'indices pointant vers le début de chaque ligne, de longueur (n + 1).
- **col_idx** et **values** sont deux tableaux, de taille égale au nombre de valeurs non nulles, représentant respectivement l'indice de la colonne et la valeur correspondante de chaque élément non nul de la matrice.

```
template <typename IdType, typename ScalarType>
struct CSRMatrix
{
    IdType n;
    std::vector<IdType> row_ptr;
    std::vector<IdType> col_idx;
    std::vector<ScalarType> values;
};
```

Cela nous offre une base solide pour déployer diverses techniques numériques.

1.2. Méthodes itératives : Jacobi et Gauss-Seidel

À partir de l'instance définie précédemment, nous implémentons ensuite deux méthodes itératives de résolution de matrices en format CSR, à savoir Jacobi et Gauss-Seidel.

Implémentation de Jacobi :

```
template<typename IdType, typename ScalarType>
int jacobi(int n, const CSRMatrix<IdType, ScalarType>& A, const
std::vector<ScalarType>& b, std::vector<ScalarType>& x, ScalarType tol, int
max_iter)
{
    std::vector<ScalarType> x_old(n, 0.0);
    for (int iter = 0; iter < max_iter; ++iter)
    {
        for (int i = 0; i < n; ++i)
        {
            ScalarType sum = 0.0;
            ScalarType diag = 0.0;
            for (int j = A.row_ptr[i]; j < A.row_ptr[i + 1]; ++j)
            {
                if (A.col_idx[j] == i)
                {
                    diag = A.values[j];
                }
                else
                {
                    sum += A.values[j] * x_old[A.col_idx[j]];
                }
            }
            x[i] = (b[i] - sum) / diag;
        }

        // Simplest stopping criterion: check the norm of the difference
        ScalarType norm = 0.0;
        for (int i = 0; i < n; ++i)
        {
            norm += std::pow(x[i] - x_old[i], 2);
        }
        norm = std::sqrt(norm);

        if (norm < tol)
        {
            return iter;
        }

        x_old = x;
    }
    return max_iter;
}

#endif //JACOBI_H
```

Implémentation de Gauss-Seidel :

```
// Code pour l'implémentation de Gauss-Seidel
template<typename IdType, typename ScalarType>
int gaussSeidel(int n, const CSRMatrix<IdType, ScalarType>& A, const
std::vector<ScalarType>& b, std::vector<ScalarType>& x, ScalarType tol, int
max_iter)
{
    for (int iter = 0; iter < max_iter; ++iter)
    {
        ScalarType norm = 0.0;
        for (int i = 0; i < n; ++i)
        {
            ScalarType sum = 0.0;
            ScalarType diag = 0.0;
            for (int j = A.row_ptr[i]; j < A.row_ptr[i + 1]; ++j)
            {
                if (A.col_idx[j] == i)
                {
                    diag = A.values[j];
                }
                else
                {
                    sum += A.values[j] * x[A.col_idx[j]];
                }
            }
            ScalarType x_new = (b[i] - sum) / diag;
            norm += std::pow(x_new - x[i], 2);
            x[i] = x_new;
        }

        norm = std::sqrt(norm);
        if (norm < tol)
        {
            return iter;
        }
    }
    return max_iter;
}

#endif //GAUSSSEIDEL_H
```

Ces implémentations nous permettent d'obtenir le tableau suivant en appliquant sur la matrix laplacien 2D de taille 9x9 :

Méthode	Nombre d'itérations pour convergence (float)	Nombre d'itérations pour convergence (double)
Jacobi	304	290
Gauss-Seidel	155	152

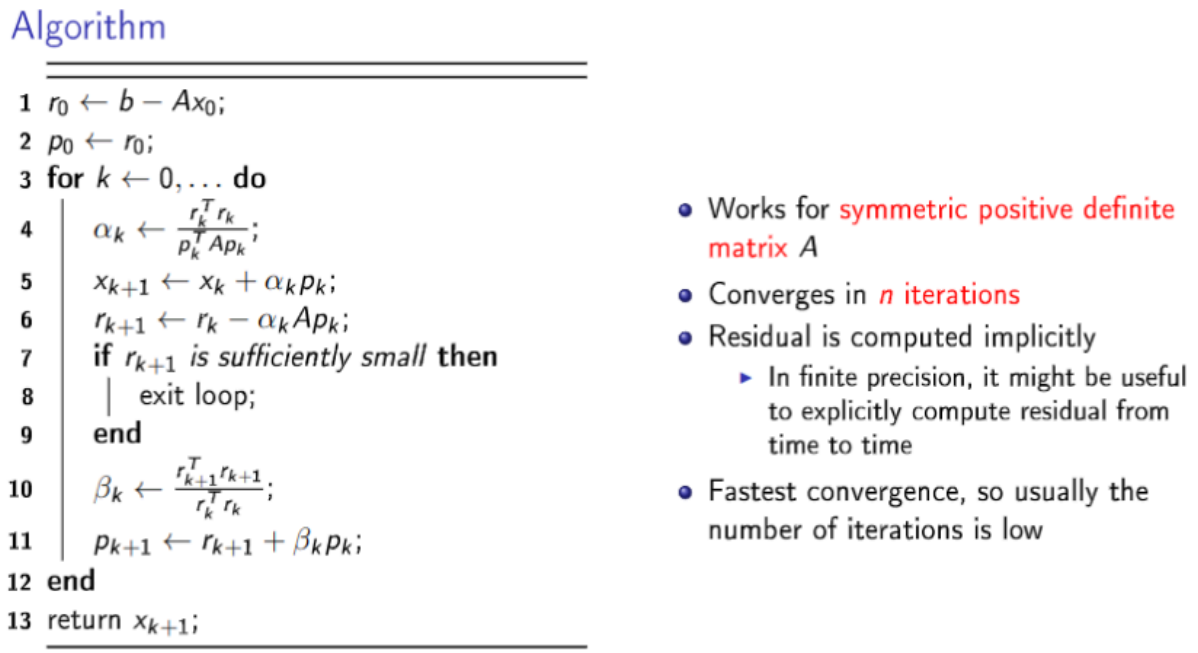
D'après les principes théoriques, la méthode de Jacobi demande environ deux fois plus d'itérations que Gauss-Seidel pour atteindre la convergence. En conséquence, même si Gauss-Seidel converge en une itération de moins, le temps d'exécution théorique reste pratiquement équivalent, bien que nous n'ayons pas testé cet aspect ici.

1.3. Méthodes gradient : Gradient conjugué et GMRES

Les méthodes de Jacobi et Gauss-Seidel imposent des exigences strictes sur la matrice (par exemple, une domination diagonale ou d'autres conditions spécifiques), ce qui limite leur convergence aux seuls cas satisfaisant ces contraintes. Pour les matrices symétriques et définies positives, la méthode CG se révèle particulièrement efficace et constitue un choix optimal. À l'inverse, GMRES offre l'avantage de pouvoir traiter aussi bien des matrices non symétriques que non définies positives, bien que cette souplesse se traduise généralement par une complexité algorithmique accrue et des coûts de calcul plus élevés.

Au cours de ce TP, nous allons également implémenter ces deux méthodes en utilisant le format CSR auquel se retrouve dans le repo du code associé.

La méthode de descente par gradient s'appuie sur l'exemple présenté en cours :



Quelques fonctions ont été implémentées pour simplifier le processus : `matVec(A, x)` réalise la multiplication Ax, `dot(u, v)` calcule le produit scalaire, `norm(u)` détermine la norme euclidienne. Il est à noter que x, u et v sont représentés sous forme dense. Ces fonctions sont également utilisées dans l'implémentation de la méthode GMRES.

```
// Multiplication d'une matrice creuse A par un vecteur x : y = A * x
template <typename IdType, typename ScalarType>
std::vector<ScalarType> matVec(const CSRMatrix<IdType, ScalarType>& A, const
std::vector<ScalarType>& x){
    std::vector<ScalarType> y(A.n, ScalarType(0));
    for (IdType i = 0; i < A.n; ++i){
        ScalarType sum = 0;
        for (IdType j = A.row_ptr[i]; j < A.row_ptr[i + 1]; ++j){
            sum += A.values[j] * x[A.col_idx[j]];
        }
        y[i] = sum;
    }
    return y;
}
```

```cpp
// Produit scalaire dot(u, v) = Σ(u[i] * v[i])
template <typename ScalarType>
ScalarType dot(const std::vector<ScalarType>& u, const std::vector<ScalarType>& v)
{
    ScalarType result = 0;
    for (size_t i = 0; i < u.size(); ++i)
    {
        result += u[i] * v[i];
    }
    return result;
}
```

Quant à l'algorithme de GMRES : [3]

GMRES

$Q \leftarrow \text{empty}(\text{size}(b, k+1))$;
$H \leftarrow \text{zeros}(k+1, k)$;
$r_0 \leftarrow b - A * x_0$;
$Q[:, 0] \leftarrow \frac{r_0}{||r_0||}$;
for $n \leftarrow 1 \dots k$ **do**
  Set entries of Q and H as an Arnoldi iteration;
  Compute the residual *res* and the least squares solution y_n for the part of H so far created;
  if *res* $<$ *tol* **then**
    break;
  end
end
return $Q[:, n+1]$, *res*;

- No more 3-terms recurrence: memory consumption is high!
- Orthogonalization is very sensitive to rounding errors

In practice, we choose a maximum size for the basis and we *restart* the algorithm from the current solution

Arnoldi iteration

Skeleton

With an arbitrary vector q_1 with norm 1 ;
for $k \leftarrow 2 \dots$ **do**
  $q_k \leftarrow A q_{k-1}$;
  for $k \leftarrow 1 \dots k-1$ **do**
    $h_{j,k-1} \leftarrow q_j^* q_k$;
    $q_k \leftarrow q_k - h_{j,k-1} q_j$;
  end
  $h_{k,k-1} \leftarrow ||q_k||$;
  $q_k \leftarrow \frac{q_k}{h_{k,k-1}}$;
end

Dans les diapositives, le pseudo-code expose principalement la logique fondamentale de GMRES, c'est-à-dire la construction de l'espace de Krylov, l'orthogonalisation via Arnoldi et la minimisation du résidu.

Dans mon implémentation, j'utilise des `std::vector<ScalarType>` pour représenter les vecteurs intermédiaires ainsi que des `std::vector<std::vector<ScalarType>>` pour stocker la base de Krylov (la matrice V) et la matrice de Hessenberg (H) en format dense.

J'ai apporté quelques compléments au pseudo-code afin de rendre l'algorithme directement exécutable :

- **Arnoldi et Givens** :
  - Pour éliminer les termes sous-diagonaux, j'effectue d'abord l'orthogonalisation (à l'aide d'un Gram-Schmidt modifié) pour mettre à jour les coefficients de H. Ensuite, j'applique successivement des rotations de Givens, dont les paramètres sont stockés dans les vecteurs cs et sn, afin de transformer H en une forme quasi-triangulaire et mettre à jour le vecteur des résidus. Ce procédé simplifie considérablement la résolution du problème de moindres carrés.

- **Boucle externe de redémarrage** :
  - Les diapositives mentionnent qu'il faut « redémarrer » après un certain nombre d'itérations sans en détailler l'implémentation. Dans mon code, j'ai intégré une boucle externe qui limite le nombre d'itérations entre deux redémarrages (via le paramètre restart). Ainsi, dès que le nombre d'itérations atteint restart ou que le résidu devient suffisamment petit, je mets à jour la solution et relance un nouveau cycle Arnoldi à partir de ce point, jusqu'à ce que la tolérance soit atteinte ou que le nombre maximal d'itérations (max_iter) soit écoulé.

Bien que théoriquement ces deux méthodes devraient aboutir à des résultats quasi identiques, notre implémentation démontre que, dans certains cas, la méthode de descente par gradient converge alors que GMRES échoue. Par exemple, en simple précision avec une tolérance de 1e-6 appliquée à un laplacien 2D de dimension 9x9, la descente par gradient converge alors que GMRES ne converge pas. Par ailleurs, dans certaines configurations, GMRES nécessite davantage d'itérations que la méthode du gradient (43 contre 13 dans le cas du laplacien 2D 9x9 avec une tolérance de 1e-5). En revanche, en double précision, les deux méthodes convergent vers la même solution avec un nombre d'itérations comparable.

1.4. Multiplication Matrice-Vecteur

La mise au point d'une fonction de multiplication matrice-vecteur ne sert pas uniquement à valider la solution, mais représente également une étape intermédiaire essentielle dans les méthodes utilisant le gradient résiduel.

Dans ce TD, nous avons implémenté la multiplication matrice-vecteur en version séquentielle et parallèle. Le programme a été validé à l'aide de données issues du site MatrixMarket.

La démarche consiste d'abord à charger les données au format COO via les outils de Matrix-Market, puis à les convertir en format CSR pour le calcul, puisque les données ne sont pas initialement stockées sous ce format. L'implémentation de la version séquentielle est relative-

ment simple. La version parallèle repose sur la division des données à traiter, en répartissant notamment les tableaux col_id et value selon le nombre de nœuds MPI pour effectuer le calcul, tandis que le tableau row_ptr présente des difficultés de répartition. Le vecteur à multiplier est stocké sous forme dense.

Les résultats ont été validés en observant la convergence lors des calculs avec les méthodes de Jacobi et Gauss-Seidel. Néanmoins, les performances ainsi que la scalabilité du programme n'ont pas encore été évaluées, ce qui pourrait être envisagé ultérieurement.

2. Résolution du Problème de Diffusion Non Linéaire

2.1. Introduction au Problème

Ce projet a pour objectif de déterminer la répartition de la température dans une flamme. Concrètement, lorsque la flamme génère de la chaleur, cette énergie se diffuse vers l'extérieur par deux mécanismes principaux : la conduction et le rayonnement. Pour obtenir la distribution de température, nous discrétisons la région de la flamme en de nombreux petits segments et utilisons des méthodes numériques itératives pour calculer la température dans chaque segment.

2.1.1. Modalités de Mise en Œuvre

- **Discrétisation :** La région continue de la flamme est divisée en de petites cellules, chacune associée à une valeur de température. Cela permet de transformer une équation continue en un ensemble d'équations discrètes plus simples.
- **Méthodes Itératives :** Deux approches principales sont utilisées :
  - La méthode de point fixe (schéma implicite linéarisé) qui met à jour la température de chaque cellule jusqu'à ce que la solution converge.
  - La méthode de Newton qui, grâce à un ajustement basé sur l'évaluation de l'erreur, accélère la convergence vers la solution.
- **Critère de Convergence :** Lors de chaque itération, le programme vérifie que la variation de température dans toutes les cellules est inférieure à un seuil prédéfini. Lorsque ce critère est satisfait, la solution est considérée comme convergée et le résultat final est déterminé.

2.2. Approche de l'Implémentation

2.2.1. Initialisation des Paramètres et Génération de la Grille

Le code commence par définir les paramètres physiques essentiels (tels que la conductivité thermique, le coefficient de rayonnement et l'intensité de la source de chaleur). Ensuite, la flamme est discrétisée en N+1 points (selon le nombre de segments que nous allons choisi), ce qui permet de déterminer la position de chaque cellule ainsi que la taille du pas dx.

2.2.2. Assemblage du Système d'Équations Discrétisées

Sur la base des relations de transfert de chaleur entre cellules voisines, un système d'équations non linéaires est construit. Sans entrer dans les détails mathématiques, ce processus consiste à « décomposer » le problème global en de nombreux petits problèmes locaux décrivant l'interaction entre une cellule et ses voisines.

2.2.3. Méthodes de Résolution

- **Itération par Point Fixe :** À partir d'une estimation initiale, la solution est mise à jour itérativement jusqu'à ce que la distribution de température devienne stable.
- **Méthode de Newton :** Cette méthode corrige rapidement la solution en évaluant et en réduisant l'erreur de manière optimale, ce qui permet d'atteindre la convergence plus efficacement.

2.2.4. Résolution du Système Linéaire

À chaque itération, il est nécessaire de résoudre un système linéaire creux. Pour ce faire, le code fait appel à l'interface IJ de la bibliothèque hypre, qui permet de traiter efficacement ces matrices de grande taille et d'améliorer la rapidité de calcul.

2.2.5. Vérification de la Convergence et Sortie des Résultats

Le programme surveille en continu la variation de température de chaque cellule. Lorsque ces variations deviennent négligeables (inférieures à la tolérance définie), la solution est considérée comme convergée. Le résultat final, correspondant à la distribution de température dans la flamme, est alors affiché et peut être visualisé à l'aide d'outils graphiques.

2.3. Explication Détaillée du Code

2.3.1. Configuration des Paramètres et Initialisation de la Grille

- **Configuration des Paramètres :** Au début du code, nous définissons plusieurs paramètres physiques essentiels, tels que :
  - **kappa_0 (conductivité thermique) :** Contrôle la vitesse de diffusion de la chaleur dans la flamme.
  - **sigma (coefficient de rayonnement) :** Détermine la quantité d'énergie diffusée par rayonnement.
  - **beta (intensité de la source de chaleur) :** Indique la puissance de la réaction chimique génératrice de chaleur.
  - **Fonction de dépendance à la température :** Par exemple, kappa(u) peut varier selon différentes lois, influençant ainsi la manière dont la température affecte la diffusion. Ici nous prenons les 2 configurations suivant pour le test :

```
1. kappa_0=0,01, sigma=0.1, beta=1, kappa(u)=kappa_0 \sqrt u,
2. kappa_{0}=0,01, sigma=1, beta = 300, kappa(u)=kappa_0 u^2.
```

- **Génération de la Grille :** Le code calcule le pas dx en fonction du nombre de points N et crée un tableau pour stocker la position de chaque point ainsi que la température initiale correspondante.

2.3.2. Assemblage du Système d'Équations Discrétisées

- **Processus de Discrétisation :** Pour chaque nœud interne, une équation en différences finies est établie à partir des températures des nœuds voisins. Pour les nœuds de frontière, des conditions particulières sont appliquées : une condition de Neumann (avec une symétrie ou « condition miroir ») à gauche et une condition de Dirichlet (valeur imposée) à droite.

- **Matrice Creuse et Vecteur du Second Membre :** Le système complet est représenté par une matrice creuse (tridiagonale, en raison de la dépendance uniquement aux voisins immédiats) et un vecteur, ce qui facilite l'utilisation de la bibliothèque hypre pour la résolution du système.

2.3.3. Processus Itératif de Résolution

- **Module d'Itération par Point Fixe :** Une fonction du code met à jour la distribution de température en appliquant l'algorithme du point fixe. À chaque itération, la nouvelle répar-

tition est calculée, et sa différence avec l'itération précédente est évaluée. Si cette différence est inférieure au seuil de tolérance, l'itération est arrêtée.

- **Module de la Méthode de Newton :** Pour accélérer la convergence, la méthode de Newton est également implémentée. Le processus consiste à :
  - Calculer le résidu non linéaire de la solution courante.
  - Construire (de manière simplifiée grâce à la structure tridiagonale) la matrice jacobienne.
  - Utiliser hypre pour résoudre le système linéaire et obtenir la correction à appliquer à la solution.
  - Répéter ces étapes jusqu'à ce que le résidu soit suffisamment faible.

2.3.4. Résolution du Système Linéaire avec hypre [1]

- **Stockage Optimisé de la Matrice :** Grâce à l'interface IJ de hypre, le système est stocké sous forme de matrice creuse, ce qui réduit l'utilisation de la mémoire et améliore l'efficacité des calculs.

- **Paramétrage du Solveur :** Le code configure, si nécessaire, un préconditionneur multigrille (BoomerAMG) pour accélérer davantage la résolution du système linéaire. Les détails de cette configuration sont conformes aux recommandations de la documentation hypre.

2.3.5. Sortie des Résultats et Visualisation

- **Contrôle de la Convergence :** La solution est vérifiée en permanence. Une fois que la variation de température dans chaque cellule est inférieure à la tolérance définie, la solution est considérée comme convergée.

- **Affichage des Résultats :** La distribution finale des températures est alors affichée, soit dans la console, soit écrite dans un fichier. De plus, le code peut générer des graphiques (par exemple, avec matplotlib) pour illustrer la variation de température en fonction de la position dans la flamme.

2.4. Détermination du Résultat Final

La solution finale est obtenue grâce à un processus itératif. À chaque itération, le programme calcule la variation de température pour chaque nœud. Lorsque ces variations deviennent infimes (inférieures à un seuil prédéfini), la solution est considérée comme convergée. À ce stade, la distribution complète de la température dans la flamme est validée et affichée comme résultat final.

2.5. Conclusion

Ce rapport présente la méthode employée pour calculer numériquement la répartition de température dans une flamme. Les résultats démontrent que la méthode de Newton converge nettement plus rapidement que l'approche linéaire implicite, avec des performances pouvant être jusqu'à dix fois supérieures.

Bibliography

[1] “HYPRE.” [Online]. Available: https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods°

[2] “MatrixMarket.” [Online]. Available: http://math.nist.gov/MatrixMarket°

[3] “GMRES_reference.” [Online]. Available: https://web.stanford.edu/class/cme324/saad-schultz.pdf°