Projet de MPNA : Méthode des itérations simultanées

Matthias Beaupère, Pierre Granger

Rapport MPNA - CHPS - 19 février 2019

Table des matières

1 Introduction et position du problème

De nombreux problèmes modernes impliquent le calcul de valeurs propres et de vecteurs propres de grandes matrices creuses. Souvent, on ne désire que les quelques valeurs propres dominantes de la matrice et on souhaite utiliser une méthode robuste et efficace. Par exemple, ce problème peut être rencontré lors de l'étude de certains systèmes physiques ou bien par exemple dans les moteurs de recherche qui reposent sur ce type de technique. Des algorithmes spécifiques doivent être utilisés afin de résoudre ces problèmes de valeur propre en un temps raisonnable. De plus les algorithmes utilisés doivent pouvoir être facilement utilisés sur les architectures massivement parallèles actuelles ce qui représente un grand défi.

2 Approche utilisée

Afin d'effectuer le calcul d'un petit nombre de vecteurs propres dominants d'une grande matrice, on se propose d'implémenter en langage **C** l'algorithme des itérations simultanées aussi appelé algorithme d'itération du sous-espace. L'algorithme des itérations simultanées est basé sur la méthode de la puissance qui consiste à extraire la valeur propre principale d'une matrice en multipliant un grand nombre N de fois un vecteur initial par la matrice d'intérêt A. On s'attend alors à voir converger le vecteur vers le vecteur propre dominant de A à la vitesse $\left(\frac{\lambda_1}{\lambda_2}\right)^N$ avec λ_1 la valeur propre dominante et λ_2 la seconde valeur propre dominante. La méthode des itérations simultanées permet de calculer un espace invariant par A de dimension $k > 1$ et ainsi d'extraire les k valeurs propres dominantes plutôt qu'un seule. Cette méthode est une généralisation de la méthode de la puissance dans laquelle on orthonormalise à chaque étape le sous-espace que l'on souhaite faire converger avant de le multiplier par A et de continuer les itérations successives. On choisit dans notre implémentation d'utiliser un sous espace de dimension $m \times m\,, m > k$ afin d'obtenir une meilleure convergence des valeurs propres. On effectue une projection dans le sous-espace de Krylov de dimension $m \times m$ dans lequel on utilise la librairie **LAPACKE** afin d'extraire les valeurs et vecteurs propres dans le sous-espace de Krylov avant de revenir dans l'espace d'origine et de calculer les vecteurs propres et valeurs propres qui correspondent par un changement de base. L'algorithme détaillé est présenté ci-après.

3 Cas séquentiel

3.1 Description de l'algorithme général

Données du problèmes :
— m : taille du sous-espace de Krylov
— k : nombre de vecteurs propres demandé
— p : précision demandé
— A : matrice de taille $n \times n$ donnée en entrée
— N : nombre d'itérations

L'algorithme implémenté est l'algorithme 1. Dans les paragraphes suivant sont détaillés chaque étape de l'algorithme.

3.1.1 Procédé de Gram-Schmidt

On utilise une décomposition QR avec le procédé de Gram-Schmidt pour orthonormalisé la matrice Q. L'ortogonalisation consiste chaque vecteur de la matrice Z dans un vecteur temporaire tout en lui soustrayant son projeté sur chaque vecteur déjà ajouté. On normalise ensuite en divisant chaque vecteur par sa norme. L'algorithme correspondant est l'algorithme 2.

Algorithm 1 Algorithme général

$Q \leftarrow rand()$
while $i = 0..N-1$ OU max(precisions) $<$ p **do**
    $Z = AQ$
    Gram-Schmidt Q
    Projection $B = Z^t AZ$
    Décomposition de Schur $B = Y^t RY$
    Retour dans l'espace d'origine $Q = ZY$
    Sélection des k vecteur propres
    Calcul de la précision
end while

Algorithm 2 Algorithme de Gram-Schmidt

for $i = 0..m-1$ **do**
    $q_i^{temp} \leftarrow q_i$
    for $k = 0..i$ **do**
        $q_i^{temp} \leftarrow q_i^{temp} - q_k(q_k.q_i)$
    end for
end for
for $i = 0..m-1$ **do**
    $q_i \leftarrow \frac{q_i^{temp}}{\|q_i^{temp}\|}$
end for

3.1.2 Décomposition de Schur

La décomposition de Schur permet de calculer les valeurs et vecteurs propre de l'espace de Krylov, aussi appelés valeurs et vecteurs de Ritz. Pour ce calcul a été utilisé la bibliothèque `lapacke`.

3.1.3 Sélection de k vecteurs propres

En entrée du programme est précisé le nombre k de vecteurs propres désirés. La précision est calculé uniquement sur les k vecteurs de plus grande valeur propre associée. Pour sélectionner ces vecteur, on range les vecteurs par valeur propre associée puis on ne garde que les k premiers.

3.1.4 Calcul de la précision

Pour calculer la précision de chaque vecteur propre, on compare le vecteur propre avec son produit par la matrice A donnée en entrée. On donne ci-dessous la formule pour le vecteur propre i.

$$p_i = \|Av - \lambda v\|$$

On se propose aussi de tester une métrique qui permet d'étudier la correction d'un vecteur propre va sa colinéarité avec son image par A. Plus sa valeur est proche de 1 plus le vecteur trouvé est une bonne estimation d'un vecteur propre de A.

$$\underset{\sim}{p_i} = \frac{< Av, v >}{\|Av\| \, \|v\|}$$

3.2 Étude de performances théorique

3.2.1 Objectif en précision ou en itération

La terminaison du programme est donnée par deux critères suivant les paramètres entrée par l'utilisateur :

— Un objectif en précision : pour un exposant p donné en argument, le programme termine si les k vecteurs propres ont une précision inférieure à 10^{-p}.

— Un objectif en itération : pour un entier i donné en entrée, le programme termine si on atteint l'itération i.

Pour les formules de complexité, on supposera que l'objectif est de N_{iter} itérations, même si en pratique le calcul s'arrête dans le cas où l'objectif en précision est spécifié et atteint avant l'objectif en itération.

3.2.2 Calcul de complexité

Exprimons la complexité totale de l'algorithme des itérations simultanées sous le forme de la somme des complexités des opérations qui le composent.

$$C^{tot} = N_{iter}(C^{AQ} + C^{GM} + C^{Proj} + C^{Schur} + C^{mm} + C^{select} + C^{preci})$$

Détaillons maintenant chaque terme de cette somme.

Produit AQ : On effectue le produit de $A \in \mathbb{R}^{N\times N}$ par $Q \in \mathbb{R}^{N\times M}$. On a donc la complexité

$$C^{AQ} = O(N^2M)$$

Procédé de Gram-Schmidt : L'algorithme de Gram-Schmidt se décompose en deux étape indépendantes :

— l'othogonalisation consiste à un produit scalaire entre de vecteur de taille N dans une double boucle couvrant l'intervalle $\{(i,j)|i \in [0,M-1]etj \in [O,i]\}$. Le cardinal de cet intervalle est $(M)log(M)$, ce qui donne une complexité $O(NMlog(M))$ pour l'orthogonalisation

— la normalisation consiste en $M-1$ produits scalaires de vecteurs de taille N, ce qui donne une complexité $O(NM)$ pour la normalisation.

La complexité pour le procédé de Gram-Schmidt est donc

$$C^{GM} = O(NMlog(M))$$

Projection : Il s'agit d'un produit entre la matrice $Z^t \in \mathbb{R}^{M\times N}$ et la matrice $A \in \mathbb{R}^{N\times N}$ puis un produit entre la matrice résultante et la matrice Z. Ces produits matriciels on tous les deux la complexité $O(MN^2)$. Ce qui donne une complexité pour l'étape de projection

$$C^{Proj} = C^{mm} = O(N^2M)$$

Décomposition de Schur : La décomposition de Schur est effectué sur l'espace de taille $M \times N$. Comme on a $N \gg M$ on peut négliger la complexité introduit par cette étape du calcul. Donc

$$C^{Schur} \approx 0$$

Sélection $C^{select} = O(N)$

Précision $C^{preci} = O(N)$

3.3 Étude de performances pratique

3.4 Ajout d'une méthode de déflation "Locking"

Toutes les valeurs propres de la matrice possèdent une vitesse de convergence différente par la méthode des itérations simultanées. Cela nous amène à affiner et recalculer constamment des vecteurs déjà connus avec la bonne précision ce qui entraine une perte de temps de calcul. De plus, des vecteurs propres qui ont déjà convergés jusqu'à la bonne précision peuvent perdre en précision à cause d'instabilités numériques au fil des calculs : la précision de certains vecteurs propres peut osciler au cours des itérations.

Afin de résoudre ces deux problèmes nous avons décidé de mettre en place une méthode de déflation appelée "locking". Le principe est simple : lorsqu'un vecteur propre a convergé jusqu'à la précision désirée, on le verrouille de sorte à ne plus le remultiplier par la matrice A et on diminue la taille du sous-espace de Krylov d'une unité. Néanmoins, on utilise toujours ce vecteur pour l'orthonormalisation afin qu'il guide la convergence correcte des veteurs restants.

3.5 Conclusions

4 Cas parallèle

4.1 Approches utilisées

Maintenant que nous avons correctement pu tester notre code séquentiel et valider les résultats que nous obtenons avec, nous nous attachons à paralléliser notre implémentation. Tout d'abord, il faut remarquer que nous effectuons des calculs dans deux espaces vectoriels différents : dans l'espace d'origine $\mathbb{R}^{N\times N}$ et dans le sous-espace de Krylov $\mathbb{R}^{m\times m}$. Les calculs les plus couteux sont donc bien évidemment ceux effectués dans l'espace d'origine qui est une dimension largement supérieure à celle du sous-espace de Krylov. Les calculs qu'il est donc nécessaire de paralléliser sont ceux effectués dans l'espace d'origine ainsi que les changements de base pour passer de l'espace d'origine au sous-espace de Krylov. Les calculs à paralléliser sont donc les produits matrice/matrice dans l'espace d'origine.

Afin de paralléliser les produits matriciels, on commence par l'implémentation la plus simple et la plus efficace qui consiste à effectuer une parallélisation en mémoire partagée grâce à **OpenMP** car la plupart des calculs sont indépendants lors du produit matriciel et que l'on peut gérer efficacement les réductions. De plus, puisque la mémoire est partagée, il n'y a pas de pénalités en temps de communication. On réalise donc cet ajout d'OpenMP à l'aide de l'ajout de pragmas devant les boucles de notre code qui peuvent être parallélisés. Les gains en performance apportés par OpenMP sont représentées sur la figure 1. On peut observer que l'on gagne un facteur d'accélération de 7 en utilisant 8 coeurs physiques ce qui est très important. Cela montre que notre code est fortement parallélisable et que peu de temps de calcul est utilisé lors des parties séquentielles. On peut ensuite observer sur la figures que lorsque l'on utilise des coeurs logiques hyperthreadés supplémentaires les performances sont dégradées avant de s'aéliorer lorsque leur nombre augmente. Ceci est logique car les coeurs hyperthreadés ne possèdent pas leurs propres modules pour effectuer les opérations et ne proposent donc pas une bonne accélération lorsque le code est fortement parallélisable.

4.2 Étude de performances théorique

4.3 Étude de performances pratique

4.4 Conclusions

5 Conclusion générale

Nous avons pu implémenter la méthode des itérations simultanées afin de calculer les valeurs et vecteurs propres d'une grande matrice creuse. Nous avons ensuite tenté d'améliorer l'algorithme initialement proposé en introduisant le "locking" permettant de modifier les paramètres de redémarrage afin de gagner

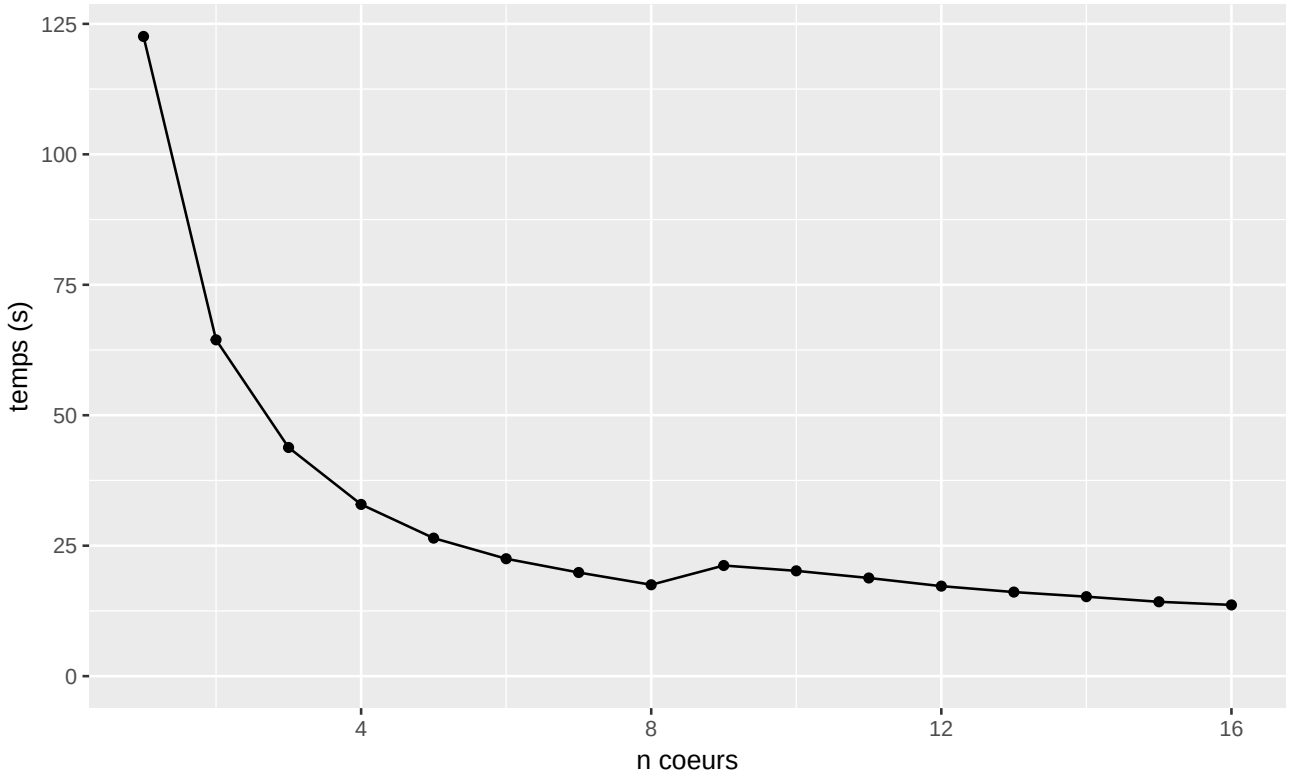


FIGURE 1 – Evolution de l'efficacité de l'implémentation en fonction du nombre de threads OpenMP utilisés sur un noeud avec 8 coeurs hyperthreadés pour un calcul de 4 valeurs propres avec $m = 8$ à $p = 10^{-8}$

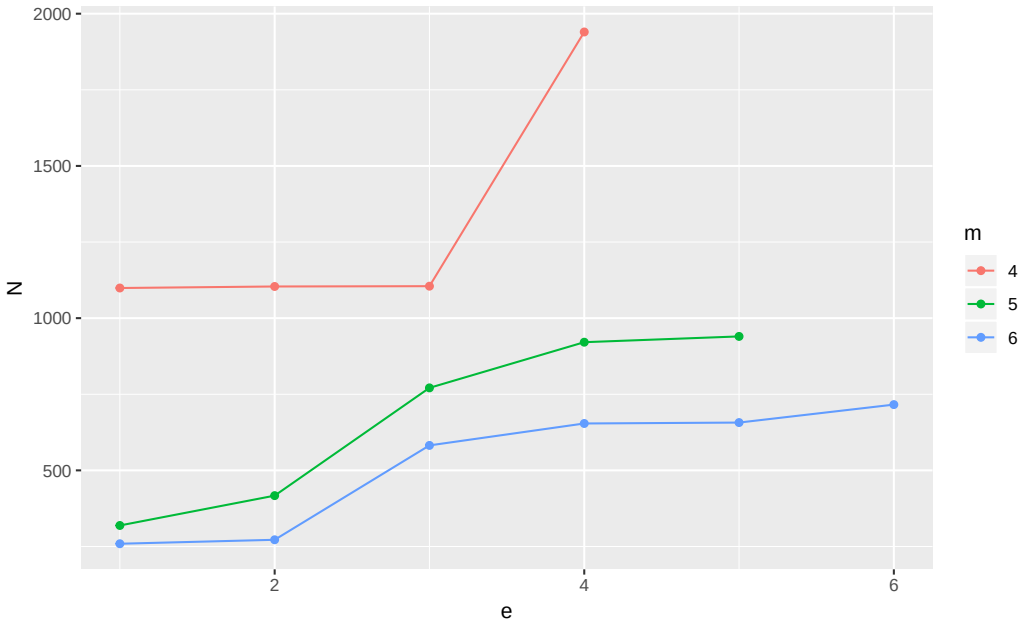


FIGURE 2 – Nombre d'itérations N nécessaires pour faire converger e valeurs propres pour différentes tailles de sous-espace de Krylov m et une précision $p = 10^{-6}$

à la fois en temps de calcul et en stabilité numérique de la convergence. Nous avons pu constater que l'ajout du locking a effectivement permis d'augmenter l'efficacité et la précision de notre implémentation. Nous avons aussi implémenté un conditionnement simpliste des matrices en entrée afin de résoudre certains problèmes de stabilité numérique dans le cas de matrices mal conditionnées en entrée.

Nous nous sommes ensuite proposés de paralléliser notre programme à l'aide de deux outils : OpenMP et MPI. Nous avons tout d'abord effectué la parallélisation des boucles avec OpenMP puis nous avons implémenté les produits de matrices couteux en MPI. Nous avons pu tester notre code parallèle sur le cluster poincare à la maison de la simulation. Les résultats furent assez décevants car notre parallélisation MPI a augmenté les temps de calcul à cause des coups trop élevés des transferts de données. Nous en concluons que la parallélisation efficace de ce type d'algoithme doit plutôt reposer sur l'implémentation de méthodes hybrides.

On représente sur la figure 7 la précision au cours des itérations pour e et m fixés dans le cas de la méthode initiale et de la méthode avec locking. On peut tout d'abord observer que la convergence est bien plus rapide lorsque le locking est utilisé (environ 2000 contre 3800 itérations). On peut en outre constater sur la figure 7a que la convergence avec locking semble moins instable.

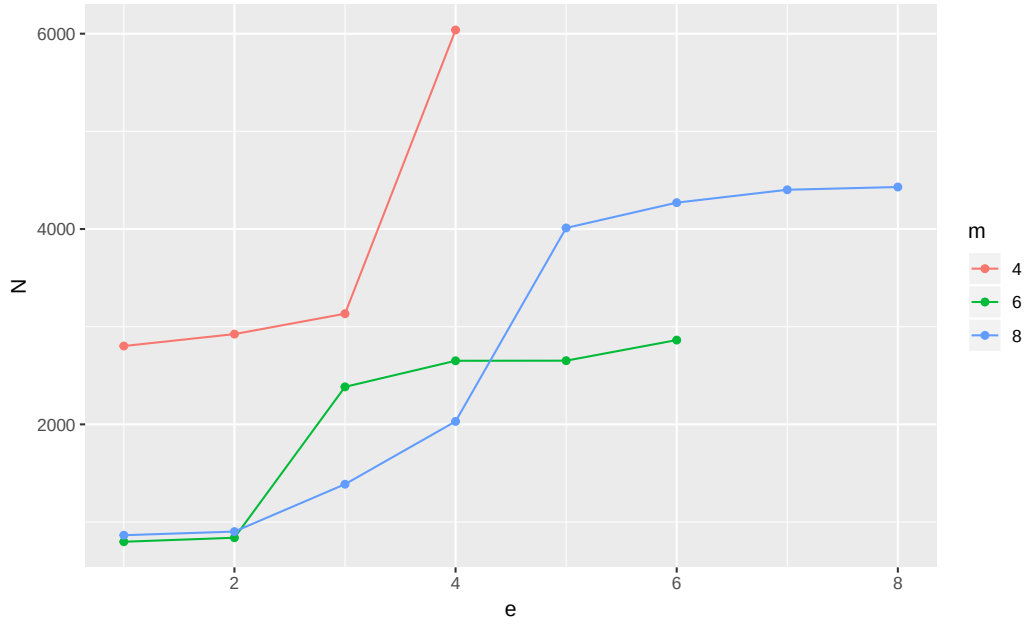


FIGURE 3 – Nombre d'itérations N nécessaires pour faire converger e valeurs propres pour différentes tailles de sous-espace de Krylov m et une précision $p = 10^{-8}$

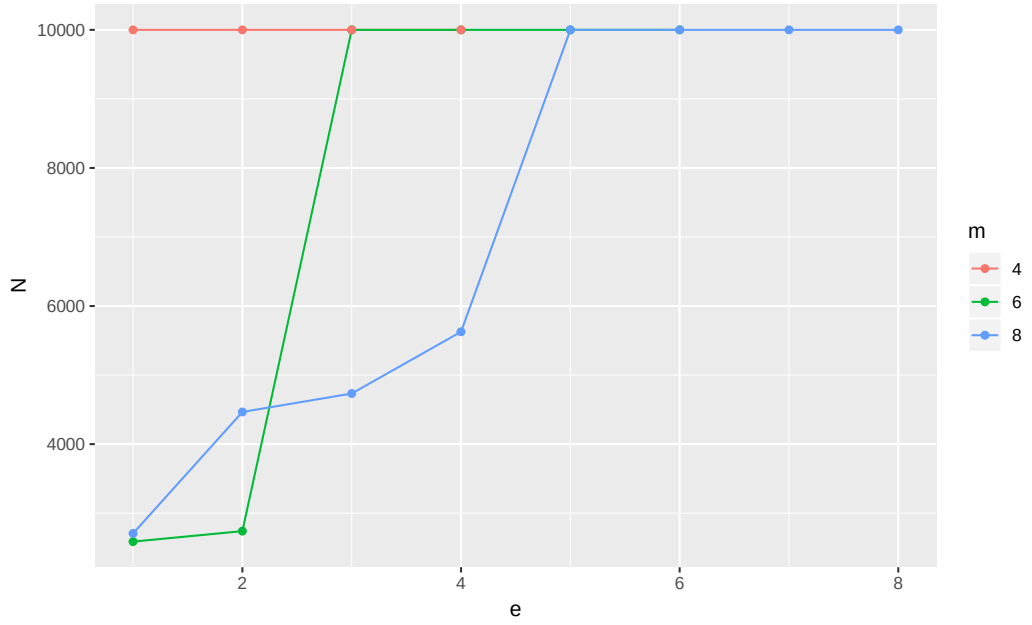


FIGURE 4 – Nombre d'itérations N nécessaires pour faire converger e valeurs propres pour différentes tailles de sous-espace de Krylov m et une précision $p = 10^{-10}$

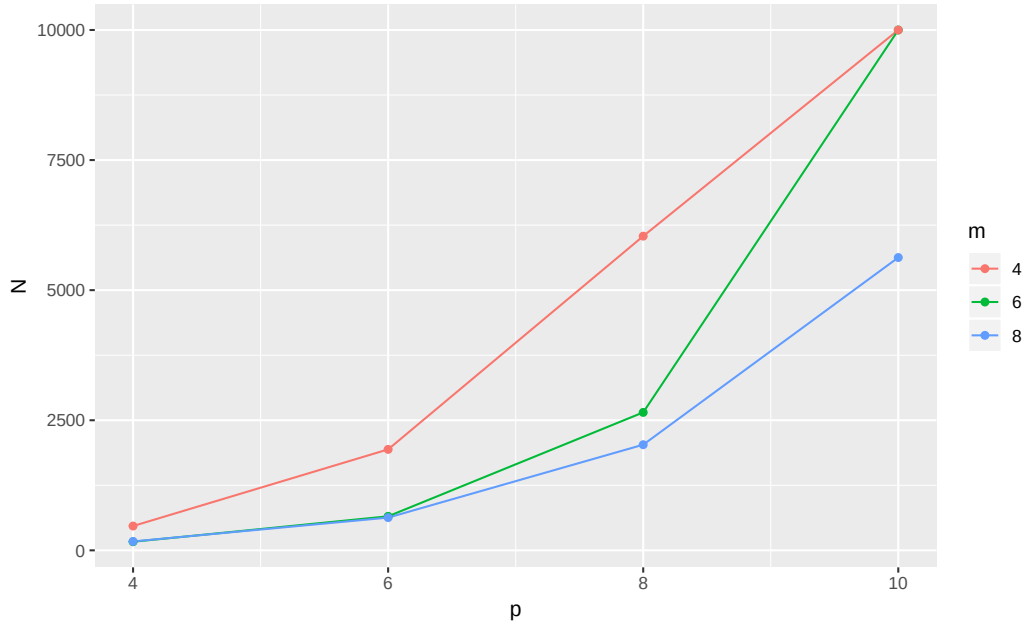


FIGURE 5 – Nombre d'itérations N nécessaires pour faire converger $e = 4$ valeurs propres pour différentes tailles de sous-espace de Krylov m et une précision p

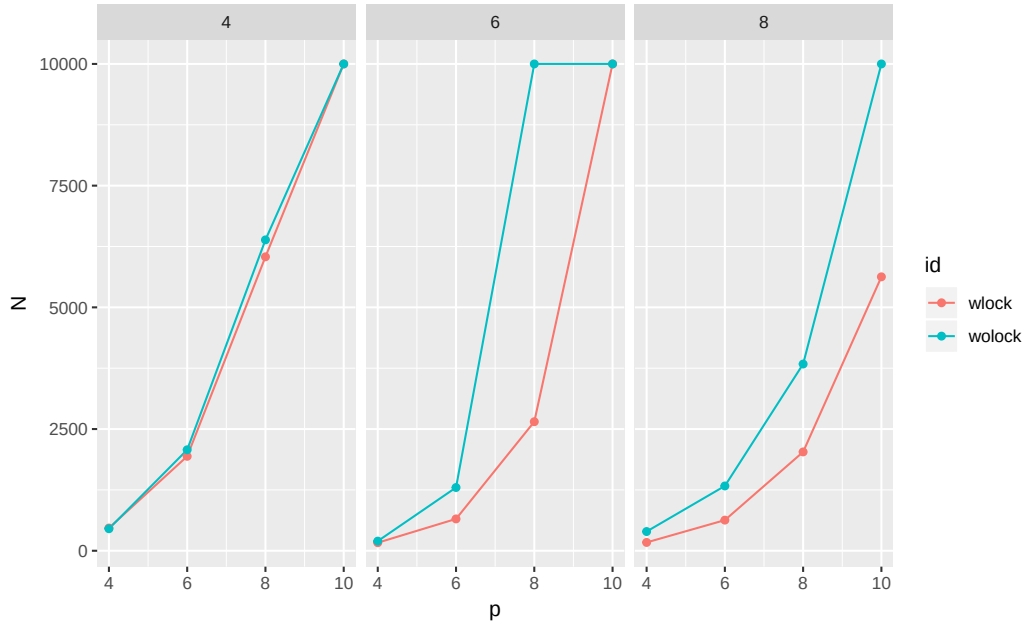
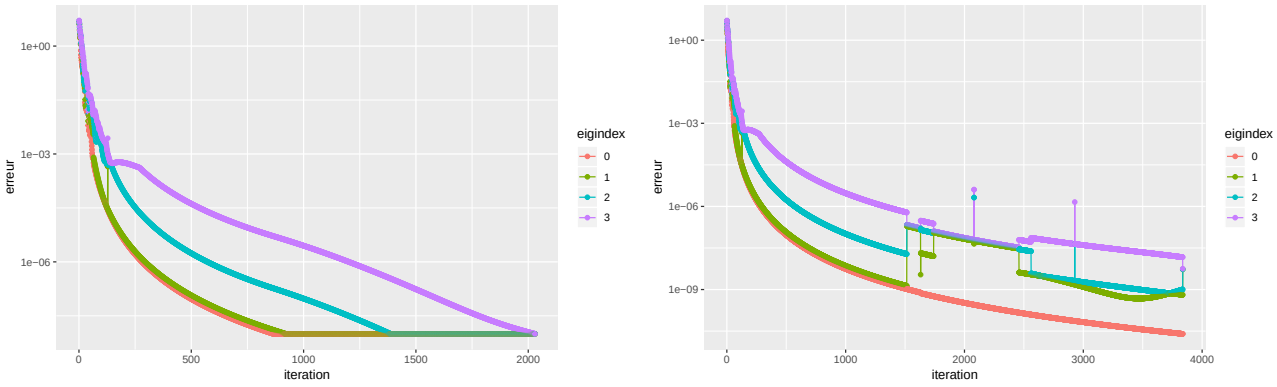


FIGURE 6 – Nombre d'itérations N nécessaires pour faire converger $e = 4$ valeurs propres pour différentes tailles de sous-espace de Krylov m et une précision p avec et sans utilisation du locking



(a) Avec locking

(b) Sans locking

FIGURE 7 – Précision au cours des itérations N pour $e = 4$ valeurs propres pour une taille de sous-espace de Krylov $m = 8$