Numéro d'ordre : 2428 Année : 1998

THÈSE

présentée à

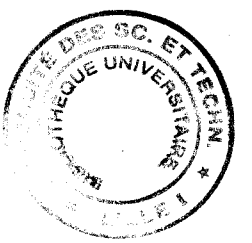
L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Caroline LE CALVEZ



Accélération de méthodes de Krylov pour la résolution de systèmes linéaires creux sur machines parallèles

le 16 Décembre 1998, devant la commission d'examen :

Président :	Jean-Marc GEIB	Professeur, USTL, Lille
Rapporteurs :	Bernard PHILIPPE	Directeur de Recherche, IRISA, Rennes
	Yousef SAAD	Professeur, University of Minnesota, Minneapolis
Examinateurs :	Claude BREZINSKI	Professeur, USTL, Lille
	Michel DAYDE	Maître de Conférence, ENSEEIHT, Toulouse
	Serge PETITON	Professeur, USTL, Lille

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A Bât M3. 59655 Villeneuve d'Ascq CEDEX

Mis en page avec la classe TheseCRIN.

Remerciements

Je remercie le Professeur **Serge PETITON** d'avoir accepté d'être mon directeur de thèse pendant ces trois années.

Je souhaite exprimer ma gratitude envers le Professeur **Claude BREZINSKI**, co-directeur de ma thèse, pour son soutien et sa confiance en mes travaux. Par ce biais, je remercie également l'ensemble des personnes du laboratoire **ANO**, qui m'ont souvent fait des remarques judicieuses sur mon travail.

Je tiens à exprimer ma reconnaissance envers le Professeur **Yousef SAAD**, qui m'a accueillie au sein de son laboratoire pendant trois mois à l'Université du Minnesota. J'ai ainsi pu bénéficier de sa grande compétence et de sa maîtrise dans les techniques des méthodes de Krylov. Je le remercie également d'avoir accepté d'être rapporteur de ma thèse.

Je remercie **Bernard PHILIPPE**, Directeur de Recherche à l'IRISA, de m'avoir fait l'honneur de rapporter ma thèse.

Je remercie également le Professeur **Jean-Marc GEIB**, de m'avoir accueillie dans le laboratoire LIFL, lorsqu'il en était le directeur, mais aussi d'avoir accepté d'être président de mon jury.

Je tiens à préciser que j'ai été particulièrement ravie de la présence en tant qu'examinateur de **Michel DAYDE**, Maître de Conférence à l'ENSEEIHT et surtout l'un de mes anciens professeurs.

Je remercie **Brígida MOLINA** d'avoir accepté que nous travaillions ensemble. Je la remercie également pour son soutien moral et son caractère jovial.

Je tiens à remercier **Richard LEHOUCQ** de m'avoir gentiment envoyé les codes en matlab de la méthode **IRA**. Cela m'a permis de gagner un temps considérable dans le développement des méthodes exposées au chapitre 4.

Je remercie également l'**E**tablissement **T**echnique **C**entral de l'**A**rmement de m'avoir accueillie à l'occasion pendant ma première année de thèse. J'ai ansi pu faire quelques expériences sur la CM5 de l'ex-**SEH**, sous les conseils éclairés de Roch de la Bourbonnais.

Je remercie l'**I**nstitut du **D**éveloppement et des **R**essources en **I**nformatique **S**cientifique de m'avoir permis de développer les codes et d'effectuer les tests qui sont décrits dans cette thèse. Je remercie également les personnes de l'assistance pour leur disponibilité et leur gentillesse.

Je remercie l'**E**dinburgh **P**arallel **C**omputing **C**enter de m'avoir accueillie pendant 2 mois à Edinburgh. Le séjour y a été non seulement très enrichissant mais aussi agréable.

Je suis également reconnaissante envers Monsieur **Pierre LALLEMAND** pour m'avoir accueillie au sein de son laboratoire l'**ASCI**, d'Avril à Septembre 1998. L'accueil y a été très chaleureux et l'environnement très propice à un travail de fin de thèse.

Je remercie également l'équipe **ANP** du **LIP6** dirigée par le Professeur **René Alt**, qui m'ont tout d'abord accueillie au sein de leur équipe et permis de finir ma thèse dans d'excellentes conditions.

Je remercie toutes les personnes des laboratoires **LIFL**, **ANO** et **ASCI**, qui ont contribué à la réalisation de cette thèse. Je remercie notamment Nathalie pour sa gentille relecture de ma thèse, Azeddine et Guy, Thomas et Martha.

Je remercie enfin toute ma famille et particulièrement Eric pour son soutien de tous les jours.

A Eric, Juliette et Raphaël,

Table des matières

Introduction générale

Nous nous intéressons à la résolution des systèmes linéaires réels

$$Ax = b \tag{1}$$

où A est une matrice carrée à coefficients dans $\mathbb{R}$, de dimension n et b est un vecteur réel de dimension n. La matrice A est creuse, ce qui signifie que son nombre d'éléments non nuls est très faible par rapport à n^2. Nous dirons qu'une matrice est creuse si son nombre d'éléments non nuls est inférieur à 3%. A est choisie la plus générale qui soit, ni symétrique, ni normale.

Les matrices creuses apparaissent dans différents domaines, notamment dans celui de la physique. Elles sont fréquemment, mais pas exclusivement, issues d'équations aux dérivées partielles servant à modéliser un problème physique particulier comme en électromagnétisme ou en mécanique des fluides. Dans le cas particulier des équations aux dérivées partielles, celles-ci sont discrétisées ainsi que le domaine sur lequel elles portent, donnant lieu à la résolution d'un système linéaire dont la matrice est creuse. La solution du système linéaire est alors une approximation de la solution physique : la solution des équations aux dérivées partielles. Nous nous restreindrons aux matrices réelles.

Par la suite, nous ne tiendrons pas compte du phénomène physique ni des équations dont est issue la matrice. En tenir compte nous apporterait des informations supplémentaires et nous aiderait, par exemple, dans le choix de la méthode de résolution du système et d'un préconditionneur adapté. Nous nous intéressons ici à résoudre un système linéaire creux et ne tenons compte que des caractéristiques algébriques de la matrice.

De ce point de vue, il existe deux grandes classes de méthodes de résolution de systèmes linéaires : les méthodes directes et les méthodes itératives. Les premières consistent à transformer la matrice en un produit de matrices facilement inversibles. La plus répandue est la factorisation de la matrice A sous la forme LU, produit d'une matrice L triangulaire inférieure et d'une matrice U triangulaire supérieure. Une telle décomposition peut être facilement réutilisée pour résoudre d'autres systèmes linéaires avec la même matrice A et des seconds membres b différents. Un des principaux inconvénients de cette technique, lorsqu'elle est appliquée à une matrice A qui est creuse, est que les matrices L et U sont plus pleines que A et nécessitent un stockage bien plus important que celui de A. Des techniques très élaborées, notamment de réordonnancement, qui tire partie de la théorie des graphes, permettent de pallier à cette difficulté et de rendre les matrices L et U relativement creuses. Cela permet en outre de diminuer la complexité de l'algorithme, qui, sinon, s'avèrerait fort coûteux.

Nous nous intéressons dans cette thèse à la seconde classe des méthodes de résolution des systèmes linéaires, à savoir les méthodes itératives. À partir d'une solution initiale

approchée, elles consistent à se rapprocher de la solution du système linéaire, itérativement, c'est-à-dire en mettant à jour cette solution approchée au moyen d'un même procédé à chaque pas. Ces méthodes se scindent en deux sous-classes principales: les méthodes stationnaires et les méthodes de Krylov. Ces deux sous-classes sont des méthodes de projection: les premières résolvent le système linéaire à chaque pas dans un sous-espace de dimension 1, tandis que les secondes le résolvent sur un sous-espace appelé sous-espace de Krylov, dont la taille croît avec le nombre d'itérations. Les premières sont moins performantes et ne sont adaptées qu'à un certain type de matrices. Les secondes sont d'implantation plus compliquée, mais plus efficaces. Une classification des méthodes de Krylov fait l'objet du chapitre 1.

Afin d'obtenir une précision de la solution calculée toujours plus grande, la discrétisation des équations dont est issue la matrice A, par différences ou éléments finis, se fait plus fine. La conséquence immédiate est que la dimension de la matrice s'accroît, rendant difficile sa résolution en un temps raisonnable.

Les architectures parallèles apparaissent alors comme une façon naturelle de réduire le temps total d'exécution du calcul de la solution. Elles permettent également de stocker une matrice qui, parfois trop volumineuse même lorsqu'elle est creuse, ne peut être contenue dans la mémoire d'un seul processeur. Se posent alors les questions de la répartition de la matrice sur les processeurs de la machine parallèle, de la gestion des communications entres les processeurs, du mode de programmation à utiliser, tout ceci en vue de résoudre le système linéaire le plus efficacement possible sur la machine parallèle utilisée.

Les premières méthodes de Krylov n'ont pas été inventées avec pour préoccupation première d'être parallèles et les premières expériences sur machines parallèles ont consisté à paralléliser les méthodes de Krylov existantes. Or certaines, très robustes, présentent des zones de calcul, qui sont intrinsèquement séquentielles. Depuis, d'autres approches ont émergé, pour améliorer la résolution des systèmes linéaires sur machines parallèles à l'aide des méthodes de Krylov. Nous pouvons les classer de la façon suivante:

- optimisation de l'implantation des méthodes de Krylov existantes d'un point de vue logiciel,
- modification algorithmique de leurs zones de calcul intrinsèquement séquentielles,
- amélioration de leur convergence en diminuant le nombre d'itérations et ainsi le nombre des opérations coûteuses.

Toutes ces approches visent naturellement à réduire le temps total d'exécution du calcul de la solution.

Nous présentons dans cette thèse deux approches parmi les trois existantes permettant d'accélérer les méthodes de Krylov **FOM**(m) et **GMRES**(m) sur architectures parallèles. La première, qui sera vue dans le chapitre 3, transforme les algorithmes initiaux afin de supprimer les zones de calcul intrinsèquement séquentielles. La seconde approche consiste à réduire le nombre total d'itérations pour converger, tout en diminuant la complexité d'une itération. Cela permet de diminuer les temps d'exécution sur machines parallèles.

La thèse est divisée en quatre chapitres. Le premier est consacré à la classification

des méthodes de Krylov suivant les critères du produit scalaire et du sous-espace de projection. Nous énumérons ensuite les opérations les plus coûteuses dans les méthodes de Krylov sur architectures parallèles et dont nous aimerions diminuer le nombre. Nous mettons également en évidence les zones de calcul intrinsèquement séquentielles de ces mêmes méthodes.

Nous présentons dans le chapitre 2 une liste non exhaustive de techniques d'accélération existantes suivant les trois approches développées ci-dessus. Ces techniques sont très nombreuses, certaines très générales s'adaptant à l'ensemble des méthodes de Krylov, comme les techniques de préconditionnement ou les méthodes par blocs, d'autres spécifiques à une méthode de Krylov particulière. Ces dernières donnent naissance à un ensemble de techniques très variées et pas toujours uniformes.

Nous développons dans les chapitres 3 et 4 des techniques accélératrices des méthodes **FOM**(m) et **GMRES**(m). Ces méthodes présentent des zones de calcul intrinsèquement séquentielles auxquelles nous essayons de remédier : les produits scalaires induits se font en séquence, ce qui diminue les performances de la méthode. Nous proposons dans le chapitre 3 de changer de produit scalaire en se plaçant d'un point de vue polynomial et de le choisir discret. Les algorithmes résultants appelés **ADOP** sont alors de même coût que les méthodes **FOM**(m) et **GMRES**(m), avec les produits scalaires transformés en un produit matrice matrice. Sur machines parallèles, une itération de ces nouveaux algorithmes est plus rapide qu'une itération de **FOM**(m) ou **GMRES**(m).

Dans le chapitre 4, nous tentons de diminuer le nombre total d'itérations de la méthode. Les méthodes **GMRES**(m) et **FOM**(m) sont des méthodes redémarrées. Au bout d'un nombre fixe d'itérations, le procédé est arrêté et redémarré. Mais une importante partie de l'information que nous avions emmagasinée est totalement perdue et doit être reconstruite. Cela induit donc un ralentissement de la convergence. Nous proposons de redémarrer les méthodes **FOM**(m) et **GMRES**(m) en gardant une certaine partie de l'information, notamment spectrale, et une complexité égale. Cela est possible grâce à la méthode **IRA** [58] qui nous permet de redémarrer «implicitement». Les méthodes résultantes dénommées **IDFOM** et **IDGMRES** permettent ainsi de converger en un nombre d'itérations inférieur par rapport aux méthodes **FOM** et **GMRES** avec un coût de stockage identique.

Le dernier chapitre résume l'ensemble des résultats obtenus et s'ouvre vers de nouvelles perspectives.

Chapitre 1

Mises en œuvre des méthodes de Krylov en environnements parallèles

Sommaire

Nous nous intéressons exclusivement aux méthodes itératives et plus particulièrement à la classe dite des méthodes de Krylov.

Dans le cas d'une matrice symétrique définie positive, la méthode de Krylov la plus connue et la plus étudiée est la méthode du Conjugate Gradient (**CG**) développée par Hestenes et Stiefel [23] en 1952. Tout d'abord considérée comme une méthode directe, parce qu'elle permet de calculer la solution de $Ax = b$ en un maximum de n itérations en arithmétique exacte, lorsque n est la dimension de la matrice, elle a été redécouverte par Reid [40] en 1971 en tant que méthode itérative et n'a pris toute son ampleur qu'avec l'utilisation plus massive des ordinateurs. La méthode du **CG** est une méthode performante qui allie quatre qualités : un coût de stockage peu élevé, un nombre d'opérations

par itération faible, une condition de minimisation de l'erreur au sens de la norme de A à chaque pas et un algorithme très parallèle.

L'enjeu de ces deux dernières décennies a été la recherche de méthodes alliant les trois premières qualités du **CG** pour la résolution de systèmes linéaires non symétriques. De nombreuses méthodes ont vu le jour comme **FOM** [41], **GMRES** [52], **Bi-CG** [27, 15], **CGS** [57], **QMR** [20], **Bi-CGSTAB** [59], **TFQMR** [18], **CMRH** [56], etc. Bien que les machines parallèles soient apparues dans les années 70 et qu'elles aient été de plus en plus utilisées à partir des années 80, la contrainte supplémentaire que la méthode recherchée soit également facilement parallélisable, comme c'est le cas du **CG**, n'a réellement préoccupé les chercheurs qu'à partir de la fin des années 80.

Dans ce chapitre, nous présentons tout d'abord une classification mathématique possible des méthodes de Krylov, classification inspirée de Ernst et Eiermann [12] et Heyouni [24]. Nous différencions ensuite les diverses implantations auxquelles elles donnent lieu. Forts de cette classification, nous nous penchons sur leur mise en œuvre en environnements parallèles. Nous mettons en évidence les opérations récurrentes, soulignons celles qui sont les plus coûteuses et celles qui posent problème.

1.1 Classification des méthodes de Krylov

1.1.1 Principe général

Les méthodes de Krylov constituent un sous-ensemble des méthodes de projection. Le principe de ces dernières consiste à projeter le problème $Ax = b$ sur un sous-espace K_m de dimension m inférieure à n, orthogonalement à un deuxième sous-espace L_m, également de dimension m, appelé sous-espace des contraintes. Le système linéaire résultant est alors de taille plus petite et est plus facile à résoudre.

Soient x^* le vecteur solution de $Ax = b$ et x_0 un vecteur initial. Les méthodes de projection consistent alors à corriger le vecteur x_0 à l'aide du vecteur y_m appartenant à K_m tel que l'équation $A(x_0 + y_m) = b$ soit vraie dans $\mathrm{L}_m^{\perp}$. Le vecteur $x_0 + y_m$ ainsi calculé est noté x_m. Lorsque $\mathrm{K}_m = \mathrm{L}_m$, la solution approchée est exacte sur K_m. On dit que la projection est orthogonale. Dans le cas contraire, lorsque $\mathrm{K}_m \neq \mathrm{L}_m$, la projection est dite oblique.

Les méthodes de Krylov se restreignent aux sous-espaces K_m de la forme

$$\mathrm{K}_m(A, u) = \mathrm{Vect}\{u, Au, \ldots, A^{m-1}u\}$$

où $\mathrm{K}_m(A, u)$ est le sous-espace de Krylov de dimension m associé à la matrice A et au vecteur u. Ce sous-espace $\mathrm{K}_m(A, u)$ est formé de l'ensemble des combinaisons linéaires des vecteurs $u, Au, \ldots, A^{m-1}u$, ce qui, d'un point de vue polynomial, s'exprime par :

Proposition 1.1 *Le sous-espace de Krylov $\mathrm{K}_m(A, u)$ est le sous-espace formé de tous les vecteurs de la forme $p(A)u$ où p est un polynôme de degré strictement inférieur à m.*

D'une façon générale, une méthode de Krylov calcule le vecteur x_m approchant la solution x^* à partir du sous-espace de Krylov $\mathrm{K}_m(A, r_0)$ (qui sera souvent abrégé par K_m lorsqu'aucune confusion ne sera possible) associé au résidu initial $r_0 = b - Ax_0$, après avoir

calculé ou mis en place tous les éléments permettant de calculer x_k pour $k = 1, \ldots, m-1$. La dimension du sous-espace de Krylov grandit avec le nombre d'itérations du procédé. D'un point de vue polynomial, nous avons $x_m = x_0 + p_m(A)r_0$ où p_m est un polynôme à coefficients dans $\mathbb{R}$ de degré strictement inférieur à m et $r_m = (I_n - Ap_m(A))r_0$. La convergence a lieu lorsque $x = x^*$, ce qui est équivalent à avoir le résidu correspondant $b - Ax$ égal au vecteur nul. Nous sommes dans ce cas lorsque le polynôme associé au résidu $(1 - xp_m(x))$, dit polynôme résiduel, est égal au polynôme minimal de A associé à r_0 (défini ci-dessous), multiplié par une constante dépendante de ce polynôme minimal.

Définition 1.1 *On appelle polynôme minimal de la matrice non singulière A par rapport au vecteur r_0, le polynôme p non nul, unitaire, de degré minimal et vérifiant $p(A)r_0 = 0$. Notons L le degré de ce polynôme.*

Si $(-\alpha_0, -\alpha_1, \ldots, -\alpha_{L-1}, 1)$ sont les coefficients du polynôme minimal dans la base canonique $(x^k)_{0 \leq k \leq L}$, nous avons alors :

$$A^L r_0 = \sum_{i=0}^{L-1} \alpha_i A^i r_0$$

avec $\alpha_0 \neq 0$, sinon le polynôme minimal aurait été de degré strictement inférieur à L, ce qui est faux par hypothèse. Nous avons finalement :

$$A\left(x_0 + \frac{1}{\alpha_0}(A^{L-1} - \sum_{i=0}^{L-2} \alpha_{i+1} A^i) r_0\right) = b,$$

et le vecteur correction $y_L = \frac{1}{\alpha_0}(A^{L-1} - \sum_{i=0}^{L-2} \alpha_{i+1} A^i) r_0$ permet de résoudre de façon exacte le système linéaire. Le polynôme résiduel est égal à $(-1/\alpha_0)p(x)$ où p est le polynôme minimal. Il n'est donc plus nécessaire de poursuivre le procédé. De plus, nous avons :

Proposition 1.2 *1.* $\forall\ 1 \leq k \leq L \quad \dim(\mathrm{K}_k(A, r_0)) = k.$

2. $\forall\ k \geq 0 \quad \dim(\mathrm{K}_{L+k}(A, r_0)) = L \quad \text{et} \quad \mathrm{K}_{L+k}(A, r_0) = \mathrm{K}_L(A, r_0).$

La proposition (2) signifie que les sous-espaces de Krylov $\mathrm{K}_k(A, r_0)$ sont A-invariants pour $k \geq L$: $A\mathrm{K}_k(A, r_0) = \mathrm{K}_k(A, r_0)\ \forall k \geq L$.

1.1.2 Critères de classification

Nous considérons à présent deux critères qui vont nous permettre de classifier les méthodes de Krylov :

- le produit scalaire noté $\langle \cdot, \cdot \rangle_{K,V}$ (et $\| \ \|_{K,V}$ la norme associée), lié à la base générée lors de la construction du sous-espace de Krylov $\mathrm{K}_m(A, r_0)$. Nous parlons donc toujours de base orthonormale par rapport à un produit scalaire donné. Cependant, lorsqu'il n'est pas spécifié, la base est orthonormale pour le produit scalaire euclidien,

- l'espace L_m lui-même. Nous considérons deux cas :

  1. soit $\mathrm{L}_m = \mathrm{K}_m(A, r_0)$. Lorsque le produit scalaire utilisé est euclidien, nous parlons de méthode **RO** pour **R**ésidu issu d'une projection **O**rthogonale. Dans le cas contraire, nous parlons de méthode **RQO** pour **R**ésidu issu d'une projection **Q**uasi-**O**rthogonale,
  2. soit $\mathrm{L}_m = A\mathrm{K}_m(A, r_0)$. Lorsque le produit scalaire utilisé est euclidien, nous parlons de méthode **RM** pour **R**ésidu **M**inimal. Dans le cas contraire, nous parlons de méthode **RQM** pour **R**ésidu **Q**uasi-**M**inimal.

Une méthode de Krylov peut donc se décrire ainsi :

$$\begin{cases} x_m = x_0 + y_m \text{ avec } y_m \in K_m(A, r_0) \\ r_m = r_0 - Ay_m \perp_{K,V} \mathrm{L}_m \text{ avec } \mathrm{L}_m = \mathrm{K}_m(A, r_0) \text{ ou } A\mathrm{K}_m(A, r_0). \end{cases}$$

Afin de calculer y_m, nous construisons une base $v_1, \ldots, v_m$ de $\mathrm{K}_m(A, r_0)$, appelée base de Krylov, telle que :

$$\mathrm{K}_k(A, r_0) = \mathrm{Span}\{v_1, \ldots, v_k\} \quad \forall k = 1, \ldots, L. \tag{1.1}$$

Nous avons alors

$$v_{m+1} \in \mathrm{Span}\{A^m r_0, v_1, \ldots, v_m\} = \mathrm{Span}\{Av_m, v_1, \ldots, v_m\}$$

et

$$\begin{aligned} AV_m &= V_{m+1}\bar{H}_m & (1.2) \\ &= V_m H_m + h_{m+1,m} e_m^{(m)T} & (1.3) \end{aligned}$$

où $V_m = [v_1, \ldots, v_m]$ est la matrice formée par les vecteurs de la base de Krylov v_i, $\bar{H}_m$ est une matrice de Hessenberg supérieure de dimension $(m+1) \times m$ et $e_i^{(j)}$ est le ième vecteur colonne de la matrice identité $j \times j$. Nous montrons ci-dessous l'exemple d'une matrice de Hessenberg supérieure de dimension 9×8, où les $*$ représentent les éléments non nuls de la matrice.

$$\bar{H}_8 = \begin{pmatrix} * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ & * & * & * & * & * & * & * \\ & & * & * & * & * & * & * \\ & & & * & * & * & * & * \\ & & & & * & * & * & * \\ & & & & & * & * & * \\ & & & & & & * & * \\ & & & & & & & * \end{pmatrix}.$$

H_m est la matrice $\bar{H}_m$ dont nous avons supprimé la dernière ligne, soit encore

$$\bar{H}_m = \begin{pmatrix} H_m \\ 0 \ldots 0 \; h_{m+1,m} \end{pmatrix} = \begin{pmatrix} H_m \\ h_{m+1,m} e_m^{(m)T} \end{pmatrix}.$$

Le vecteur correction y_m s'exprime à présent dans la base $v_1, \ldots, v_m$ sous la forme $y_m = V_m z_m$ et

$$\begin{aligned} x_m &= x_0 + V_m z_m \\ r_m &= b - Ax_m \\ &= r_0 - AV_m z_m. \end{aligned} \tag{1.4}$$

D'après (1.1) appliquée à $k = 1$, nous avons $r_0 = \beta v_1$ avec $\beta \neq 0$, puis en utilisant (1.2), nous obtenons :

$$r_m = V_{m+1}(\beta e_1^{(m+1)} - \bar{H}_m z_m). \tag{1.5}$$

1.1.3 Méthodes RO et RM

Nous différencions à présent les vecteurs z_m, y_m, x_m et r_m des différentes méthodes en leur ajoutant un exposant correspondant au nom de la méthode.

Dans le cas d'une projection **RO**, nous avons

$$V_m^T r_m^{RO} = 0 \tag{1.6}$$

tandis que pour une projection **RM**

$$(AV_m)^T r_m^{RM} = 0. \tag{1.7}$$

Cette dernière relation est équivalente à

$$r_m^{RM} = \min_{v \in \mathrm{K}_m(A, r_0)} \|r_0 - Av\|_2 \tag{1.8}$$

d'où le nom donné de méthode de projection à résidu minimal. Il serait plus juste de parler de résidu de norme euclidienne minimale.

Lorsque les vecteurs $v_1, \ldots, v_m, v_{m+1}$ forment une base orthonormale, V_{m+1} est une matrice orthonormale et la condition (1.6) valide pour les méthodes **RO** devient :

$$I_{m \times (m+1)}(\beta e_1^{(m+1)} - \bar{H}_m z_m^{RO}) = 0 \tag{1.9}$$

avec $I_{m \times (m+1)}$ la matrice identité de dimension $m \times (m+1)$. Cette relation se simplifie en

$$\beta e_1^{(m)} - H_m z_m^{RO} = 0. \tag{1.10}$$

Lorsque H_m n'est pas singulière, la solution approchée x_m^{RO} existe et s'écrit :

$$x_m^{RO} = x_0 + V_m H_m^{-1}(\beta e_1^{(m)}). \tag{1.11}$$

Si nous nous intéressons maintenant au cas des projections **RM**, la condition (1.7) s'écrit :

$$\bar{H}_m^T(\beta e_1^{(m+1)} - \bar{H}_m z_m^{RM}) = 0 \tag{1.12}$$

qui sont exactement les équations normales du problème

$$\begin{aligned} z_m^{RM} &= \arg \min_{z \in \mathbb{R}^m} \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2 \\ r_m^{RM} &= \arg \min_{v \in \mathrm{K}_m(A, r_0)} \|r_0 - Av\|_2. \end{aligned} \tag{1.13}$$

L'équation précédente est exactement l'équation (1.8). Le calcul de z_m^{RM} nécessite de résoudre un problème de moindres carrés linéaire de dimension $(m+1) \times m$, ce qui peut se faire à l'aide de la factorisation **QR** de la matrice $\bar{H}_m$. Ces calculs seront détaillés dans les paragraphes suivants. Nous avons en outre :

$$x_m^{RM} = x_0 + V_m \bar{H}_m^{\dagger}(\beta e_1^{(m+1)})$$

avec $\bar{H}_m^{\dagger} = (\bar{H}_m^T \bar{H}_m)^{-1} \bar{H}_m^T$, la matrice pseudo-inverse de $\bar{H}_m$.

1.1.4 Méthodes RQO et RQM

Lorsque la matrice V_{m+1} n'est plus orthonormale, nous ne pouvons plus simplifier les équations (1.6) et(1.7), qui s'écrivent respectivement :

$$V_m^T V_{m+1} \bar{H}_m z_m^{RO} = V_m^T r_0$$

et

$$\bar{H}_m^T V_{m+1}^T V_{m+1} \bar{H}_m z_m^{RM} = \bar{H}_m^T V_{m+1}^T r_0.$$

La connaissance de la matrice de Gram $M_{m+1} = V_{m+1}^T V_{m+1}$ est nécessaire dans le calcul des vecteurs z_m^{RO} et z_m^{RM}. Or les méthodes qui n'utilisent pas de bases orthonormales souhaitent réduire le volume de stockage en faisant en sorte que les vecteurs v_i de la base de Krylov puissent se déduire les uns des autres à l'aide d'une récurrence courte (en général à trois termes). Seuls les vecteurs nécessaires à la construction du prochain vecteur de la base de Krylov v_{m+1} sont conservés, ce qui représente un nombre faible et fixe de vecteurs à stocker à chaque pas de l'algorithme. Il devient alors impossible de former la matrice M_{m+1}. Une alternative consiste à supposer que la matrice V_{m+1} est quasi-orthonormale et à considérer que la matrice $V_{m+1}^T V_{m+1}$ est proche de la matrice identité. Les solutions **RQO** et **RQM** se calculent alors comme suit :

$$x_m^{RQO} = x_0 + V_m z_m^{RQO}$$

tel que

$$\beta e_1^{(m)} = H_m z_m^{RQO}$$

et

$$x_m^{RQM} = x_0 + V_m z_m^{RQM}$$

tel que

$$z_m^{RQM} = \arg \min_{z \in \mathbb{R}^m} \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2.$$

En fait, ces projections sont respectivement orthogonale et minimale pour le produit scalaire suivant :

$$\begin{aligned} \langle \cdot, \cdot \rangle_{K_L,V_L} : \quad \mathrm{K}_L(A, r_0) \times \mathrm{K}_L(A, r_0) &\rightarrow \mathbb{R} \\ \langle u, v \rangle_{K_L,V_L} &= \langle V_L a, V_L b \rangle_{K_L,V_L} \\ &= (a, b)_2 \end{aligned}$$

avec a et b les uniques vecteurs appartenant à $\mathbb{R}^L$ tels que $u = V_L a$ et $v = V_L b$. Nous avons finalement :

$$r_m^{RQO} \perp_{K_L,V_L} \mathrm{K}_m(A, r_0)$$

et

$$r_m^{RQM} \perp_{K_L,V_L} A\mathrm{K}_m(A, r_0).$$

Le produit scalaire utilisé est directement lié à la base de Krylov construite et plus exactement à V_L.

1.1.5 Méthodes PO et PM

Nous pouvons donc regrouper les méthodes **RO** et **RQO** sous le terme de méthodes **PO** pour **P**rojection **O**rthogonale. Elles se résument ainsi :

1. Construction d'une base $v_1, \ldots v_{m+1}$ de K_m avec $r_0 = \beta v_1$, ainsi que de la matrice $\bar{H}_m$ vérifiant $AV_m = V_{m+1}\bar{H}_m$,
2. Résolution de $H_m z_m^{PO} = \beta e_1^{(m)}$ et calcul de $x_m^{PO} = x_0 + V_m z_m^{PO}$.

De la même façon, les méthodes **RM** et **RQM** se regroupent sous la dénomination **PM** pour **P**rojection avec condition de **M**inimisation. La première étape (1) qui consiste à construire V_{m+1} et $\bar{H}_m$ est identique à celle des méthodes **PO**. L'étape (2) est remplacée par la résolution de $z_m^{PM} = \arg\min_z \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2$ et le calcul de $x_m^{PM} = x_0 + V_m z_m^{PM}$.

Dans le paragraphe suivant, nous décrivons plus en détail les méthodes **PO** et **PM**, ainsi que les différents algorithmes mathématiquement équivalents qui en découlent.

1.2 Une méthode pour plusieurs algorithmes

Nous avons vu dans le paragraphe précédent, que, outre la condition de projection orthogonale **PO** ou oblique **PM**, les méthodes se différencient aussi par le choix de la base de K_m. Nous allons caractériser celle-ci à l'aide des vecteurs $u_1, \ldots, u_m$ de $\mathbb{R}^n$, qui peuvent être quelconques, mais qui doivent être linéairement indépendants. Les différents choix de ces vecteurs seront vus un peu plus loin. Les vecteurs u_j permettent de caractériser les vecteurs v_j du sous-espace K_m, pour $j = 1, \ldots, m$, ainsi :

$$v_j \in \mathrm{K}_j(A, r_0) \quad \text{et} \quad (v_j, u_i) = u_i^T v_j = \eta_j \delta_{i,j} \quad \text{pour} \quad i = 1, \ldots, j.$$

Nous avons alors la relation matricielle suivante

$$U_m^T V_m = L_m \tag{1.14}$$

où L_m est une matrice triangulaire inférieure de dimension $m \times m$ et de diagonale principale égale à $\eta_1, \ldots, \eta_m$.

Les vecteurs u_i peuvent être soit fixés à l'avance, comme c'est le cas des méthodes de Hessenberg [24] et **CMRH** [56, 24], où les vecteurs u_i sont choisis égaux aux vecteurs de la base canonique, soit élaborés au fur et à mesure du calcul des v_i, comme c'est le cas pour les méthodes **Bi-CG** [27, 15] et **QMR** [20]. La construction de la base $v_1, \ldots, v_{m+1}$ se fait au travers du processus de Hessenberg généralisé décrit ci-dessous, qui calcule simultanément les matrices V_{m+1} et $\bar{H}_m$ à partir de la matrice U_m, dont les colonnes sont formées par les vecteurs $u_1, \ldots, u_m$.

Algorithme 1.1 $[V_{m+1}, \bar{H}_m]$ = *Hessenberg-généralisé* (A, r_0, m, U_m)

1. *Initialisation :*

   (a) *Choix de* $\beta \neq 0$ *coefficient normalisateur de* r_0

   (b) *Calcul de* $v_1 = r_0/\beta$ *et* $\eta_1 = (v_1, u_1)$

2. *Boucle principale*
   Pour $j = 1, \ldots, m$ *Faire*

   (a) $w = Av_j$

   (b) *Pour* $i = 1, \ldots, j$ *Faire*
   $h_{i,j} = (w, u_i)/\eta_i$
   $w = w - h_{i,j} v_i$
   Fin Pour

   (c) *Choix de* $h_{j+1,j}$ *facteur normalisateur de* v_{j+1}

   (d) $v_{j+1} = w/h_{j+1,j}$ *et* $\eta_{j+1} = (v_{j+1}, u_{j+1})$

   Fin Pour

3. *Définir* $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $\bar{H}_m = \{h_{i,j}\}$

Lorsque $\eta_{j+1} = (v_{j+1}, u_{j+1}) = 0$, on dit qu'il y a une panne du procédé (*breakdown*) : le processus est arrêté. Si cela se produit, soit $v_{j+1} = 0$ et dans ce cas la panne est propice (*lucky breakdown*), puisqu'un sous-espace invariant a été trouvé et nous verrons ci-dessous que la solution exacte est calculée par les méthodes **PO** ou **PM**, soit $v_{j+1} \perp u_{j+1}$: dans ce cas, si les u_j sont fixés à l'avance nous calculons la solution x_j et le résidu correspondant r_j et redémarrons le processus avec $x_0 = x_j$. Si les vecteurs u_j ne sont pas connus à l'avance, nous choisissons un autre vecteur u_{j+1} qui n'est pas orthogonal à v_{j+1}. Par la suite, nous supposons que $\eta_j \neq 0 \ \forall j = 1, \ldots, m$ *i.e.* que la matrice L_m est inversible.

Dans le cas d'une projection **PO**, nous résolvons $H_m z_m^{PO} = \beta e_1^{(m)}$.

Proposition 1.3 *Résoudre* $H_m z_m = \beta e_1^{(m)}$ *est équivalent à résoudre* $U_m^T r_m = 0$.

Preuve: Nous avons $r_m = r_0 - AV_m z_m$, donc $U_m^T r_m = 0$ s'écrit:

$$\begin{aligned} & U_m^T(r_0 - AV_m z_m) = 0 \\ \Leftrightarrow \quad & U_m^T r_0 - U_m^T V_{m+1} \bar{H}_m z_m = 0 \\ \Leftrightarrow \quad & U_m^T V_m(\beta e_1^{(m)}) - (L_m, 0)\bar{H}_m z_m = 0 \\ \Leftrightarrow \quad & L_m(\beta e_1^{(m)}) - L_m H_m z_m = 0. \end{aligned}$$

Comme L_m est non-singulière, cette dernière équation est exactement équivalente à:

$$H_m z_m = \beta e_1^{(m)}. \quad \blacksquare$$

Le résidu correspondant est alors égal:

$$\begin{aligned} r_m^{PO} &= V_{m+1}(\beta e_1^{(m+1)} - \bar{H}_m z_m^{PO}) \\ &= V_{m+1}(\beta \begin{pmatrix} e_1^{(m)} \\ 0 \end{pmatrix} - \begin{pmatrix} H_m \\ h_{m+1,m} e_m^{(m)T} \end{pmatrix} H_m^{-1}(\beta e_1^{(m)})) \\ &= -h_{m+1,m}(e_m^{(m)T} z_m^{PO}) v_{m+1} \\ &= -h_{m+1,m}(z_m^{PO})_m \, v_{m+1} \end{aligned} \tag{1.15}$$

avec $(z_m^{PO})_m$ la dernière composante de z_m^{PO}. Lorsque $h_{m+1,m} = 0$, nous avons alors trouvé la solution x^* du système linéaire.

Lorsque L_m est inversible, la matrice $L_m^{-1}U_m^T = (U_m^T V_m)^{-1}U_m^T$ est définie: c'est une matrice inverse généralisée à gauche de V_m, notée V_m^{Left}, car elle vérifie:

$$V_m^{Left} V_m = I_m. \tag{1.16}$$

Proposition 1.4 *La matrice $(V_m^{Left})^T V_m^{Left}$ est symétrique définie positive sur K_m.*

Preuve: Il est évident que la matrice $(V_m^{Left})^T V_m^{Left}$ est symétrique. Montrons qu'elle est définie positive sur K_m:

$$v^T(V_m^{Left})^T V_m^{Left} v = (V_m^{Left} v)^T (V_m^{Left} v).$$

Or

$$\forall v \in \mathrm{K}_m, \forall m \leq L, \ \exists! z \in \mathbb{R}^m \ / \ v = V_m z$$

donc

$$v^T(V_m^{Left})^T V_m^{Left} v = \|z\|_2 \geq 0.$$

Pour $v \neq 0, z \neq 0$ et $\|z\|_2 \neq 0$. La matrice $(V_m^{Left})^T V_m^{Left}$ est bien symétrique définie positive sur K_m. $\blacksquare$

Nous pouvons donc définir le produit scalaire sur K_m associé à cette matrice, qui s'écrit:

$$\begin{aligned} (\cdot,\cdot)_{(V_m^{Left})^T V_m^{Left}}: \ \mathrm{K}_m(A, r_0) \times \mathrm{K}_m(A, r_0) &\rightarrow \mathbb{R} \\ (u, v)_{(V_m^{Left})^T V_m^{Left}} &= (u, (V_m^{Left})^T V_m^{Left} v). \end{aligned}$$

Nous avons alors :

$$(u,v)_{(V_m^{Left})^T V_m^{Left}} = (V_m^{Left}u, V_m^{Left}v) = (a,b)_2 = (\tilde{a},\tilde{b})_2 = \langle u,v\rangle_{K_L,V_L} \tag{1.17}$$

avec a et b les uniques vecteurs appartenant à $\mathbb{R}^m$ tels que $u = V_m a$ et $v = V_m b$. Les vecteurs $\tilde{a}$ et $\tilde{b}$ appartiennent à $\mathbb{R}^L$ et sont définis comme suit :

$$\tilde{a} = \begin{pmatrix} a \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{et} \quad \tilde{b} = \begin{pmatrix} b \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Proposition 1.5 *Résoudre $H_m z_m = \beta e_1^{(m)}$ est équivalent à caculer le résidu r_m tel qu 'il vérifie $r_m \perp_{(V_{m+1}^{Left})^T V_{m+1}^{Left}} \mathrm{K}_m(A, r_0)$.*

Preuve : $r_m \perp_{(V_{m+1}^{Left})^T V_{m+1}^{Left}} \mathrm{K}_m(A, r_0)$ peut s'exprimer ainsi grâce à l'équation (1.5) :

$$\left(V_{m+1}(\beta e_1^{(m+1)} - \bar{H}_m z_m), V_{m+1} \begin{pmatrix} y \\ 0 \end{pmatrix} \right)_{(V_{m+1}^{Left})^T V_{m+1}^{Left}} = 0 \qquad \forall y \in \mathbb{R}^m$$

soit encore $(\beta e_1^{(m)} - H_m z_m, y) = 0$ pour tout vecteur y dans $\mathbb{R}^m$, ce qui est équivalent à avoir $\beta e_1^{(m)} - H_m z_m = 0$.■

Après avoir construit V_{m+1} et $\bar{H}_m$ à l'aide du processus généralisé de Hessenberg, les méthodes **PM** résolvent $z_m^{PM} = \arg\min_z \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2$. Pour les mêmes raisons que celles invoquées dans la proposition 1.5, nous avons :

Proposition 1.6 *Résoudre $z_m = \arg\min_z \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2$ est équivalent à résoudre $z_m = \arg\min_z \|r_m\|_{(V_{m+1}^L)^T V_{m+1}^L}$.*

Lorsque $h_{m+1,m} = 0$, le problème de minimisation $z_m^{PM} = \arg\min_z \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2$ revient à résoudre le système linéaire $H_m z_m^{PO} = \beta e_1^{(m)}$. Nous avons alors $x_m^{PM} = x_m^{PO} = x^*$.

Nous venons de voir que résoudre $r_m \perp_{(V_{m+1}^{Left})^T V_{m+1}^{Left}} \mathrm{K}_m(A, r_0)$ est équivalent à résoudre $r_m \perp_{K_L,V_L} \mathrm{K}_m(A, r_0)$. De même, résoudre $r_m \perp_{(V_{m+1}^{Left})^T V_{m+1}^{Left}} A\mathrm{K}_m(A, r_0)$ est équivalent à résoudre $r_m \perp_{K_L,V_L} A\mathrm{K}_m(A, r_0)$. Or pour déterminer r_m, il faut que V_{m+1}^{Left} existe et donc que L_{m+1} (et non L_m) soit inversible.

Dans le prochain paragraphe, nous étudions en détail les méthodes **RO** et **RM**, car elles sont utilisées dans la suite de la thèse, puis nous décrivons de manière plus succinte plusieurs méthodes de type **RQO** et **RQM**.

1.2.1 Méthodes issues du produit scalaire euclidien

Nous nous attachons dans ce paragraphe aux méthodes qui utilisent comme produit scalaire $\langle\cdot,\cdot\rangle_{K_L,V_L}$ le produit scalaire euclidien. Les méthodes se subdivisent en deux catégories :

- celles qui construisent de façon explicite la base $v_1, \ldots, v_m$ et projettent la solution sur celle-ci. Ces méthodes sont dites à récurrence longue,

- celles qui le font de façon implicite. L'ensemble des vecteurs $v_1, \ldots, v_m$ n'est pas conservé, voire même calculé, mais remplacé par des vecteurs dits de direction. La projection est quant à elle bien effectuée. Ces méthodes sont dites à récurrence courte.

Nous abordons en premier lieu les méthodes explicites.

Méthodes explicites

Les méthodes **RO** et **RM** construisent une base orthonormale du sous-espace de Krylov au travers du processus de Hessenberg généralisé, alors simplifié et renommé le processus d'Arnoldi. Différentes versions de ce processus existent, notamment celles utilisant l'algorithme de Gram-Schmidt classique (**CGS**) et l'algorithme de Gram-Schmidt modifié (**MGS**), qui sont mathématiquement équivalents. L'algorithme **MGS** présente l'avantage d'être plus stable numériquement. En effet, lors de la construction de la base $v_1, \ldots, v_m$, au fur et à mesure que m grandit, les vecteurs v_i deviennent rapidement de moins en moins orthogonaux entre-eux. Utiliser l'algorithme **MGS** permet d'y remédier mais pas toujours. Une autre solution consiste à utiliser l'algorithme **CGS**, qui est plus parallèle que **MGS** et à réorthogonaliser les vecteurs v_i une seconde fois. Cette solution présente l'inconvénient de doubler le nombre d'opérations par rapport à **MGS** ou **CGS** seuls. Enfin, une autre possibilité consiste à utiliser l'algorithme de Householder [49], qui est très stable numériquement, mais malheureusement deux fois plus coûteux que les algorithmes **MGS** ou **CGS**. Nous n'utiliserons ni l'algorithme de Householder, ni **CGS** avec réorthogonalisation, mais **MGS**, qui représente un bon compromis, puisqu'il est de faible coût et relativement bon numériquement. Le processus d'Arnoldi utilisé avec l'algorithme **MGS** s'écrit :

Algorithme 1.2 $[V_{m+1}, \bar{H}_m]$ = *Arnoldi*(A, r_0, m)

1. *Initialisation : Calcul de* $v_1 = r_0/\|r_0\|_2$
2. *Algorithme de Gram-Schmidt modifié*
   Pour $j = 1, \ldots, m$ *Faire*
   (a) $w = Av_j$
   (b) *Pour* $i = 1, \ldots, j$ *Faire*
   $h_{i,j} = (w, v_i)$
   $w = w - h_{i,j} v_i$
   Fin Pour
   (c) $h_{j+1,j} = \|w\|_2$
   (d) *si* $h_{j+1,j} \neq 0$ *alors* $v_{j+1} = w/h_{j+1,j}$ *sinon stop*
   Fin Pour
3. *Définir* $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $\bar{H}_m = \{h_{i,j}\}$

Lorsque $h_{j+1,j}$ est égal à zéro, un sous-espace de Krylov invariant a été trouvé : $L = j$. Les deux méthodes **RO** et **RM** calculent alors la solution exacte. Nous supposons par la suite qu'aucun des éléments $h_{j+1,j}$ de la matrice $\bar{H}_m$ ainsi construite n'est nul. La matrice $\bar{H}_m$ est donc une matrice de Hessenberg supérieure irréductible, c'est-à-dire une matrice de Hessenberg supérieure dont tous les éléments de la sous-diagonale sont non nuls. Nous avons en outre $h_{j+1,j} > 0 \; \forall j = 1, \ldots, m$.

La méthode **RO** qui calcule la solution approchée x_m^{RO} à l'aide de la relation $x_m^{RO} = x_0 + V_m H_m^{-1}(\beta e_1^{(m)})$ s'appelle la méthode **F**ull **O**rthogonalization **M**ethod (**FOM**) et est due à Saad [41] :

Algorithme 1.3 (RO) $[x_m, r_m, V_{m+1}, \bar{H}_m] = FOM(A, x_0, m)$

1. *Initialisation :* $r_0 = b - Ax_0$
2. *Construction de la base :* $[V_{m+1}, \bar{H}_m] = Arnoldi(A, r_0, m)$
3. *Calcul de la solution approchée :*
   - *Calcul de* z_m *solution de* $H_m z_m = \beta e_1^{(m)}$
   - $x_m = x_0 + V_m z_m$
   - $r_m = r_0 - AV_m z_m$.

La méthode **RM** construisant la solution approchée x_m^{RM} à partir des formules (1.4) et (1.13) s'appelle la méthode **G**eneralized **M**inimal **RES**idual (**GMRES**). Elle est due à Saad et Schultz [52] :

Algorithme 1.4 (RM) $[x_m, r_m, V_{m+1}, \bar{H}_m] = GMRES(A, x_0, m)$

1. *Initialisation :* $r_0 = b - Ax_0$
2. *Construction de la base :* $[V_{m+1}, \bar{H}_m] = Arnoldi(A, r_0, m)$
3. *Calcul de la solution approchée :*
   - $z_m = \arg\min_z \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2$
   - $x_m = x_0 + V_m z_m$
   - $r_m = r_0 - AV_m z_m$.

Nous allons voir à présent comment se calculent les vecteurs z_m^{RO} et z_m^{RM}. Nous appliquons, sans la calculer, la matrice $Q_m = \Omega_m^{(m+1)} \ldots \Omega_1^{(m+1)}$, produit de matrices de rotation

de Givens de dimension $m+1$ où

$$\Omega_i^{(m+1)} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & c_i & s_i & & \\ & & & -s_i & c_i & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix} \leftarrow i^{\text{ème}}\text{ligne}$$

$$\uparrow$$
$$i^{\text{ème}}\text{colonne}$$

telle que la matrice Q_m de dimension $(m+1) \times (m+1)$ transforme la matrice $\bar{H}_m$ en une matrice triangulaire supérieure, soit encore :

$$Q_m \bar{H}_m = \bar{R}_m = \begin{pmatrix} R_m \\ 0 \end{pmatrix} \tag{1.18}$$

avec R_m une matrice $m \times m$ triangulaire supérieure, non singulière car H_m est supposée non singulière. Nous avons aussi :

$$Q_{m-1} H_m = Q_{m-1} \left(\bar{H}_{m-1} \mid h_{1:m,m} \right) = \left(\begin{array}{c|c} R_{m-1} & Q_{m-1} h_{1:m,m} \\ 0 \ldots 0 & \end{array} \right) = T_m$$

avec $h_{1:m,m}$ la dernière colonne de H_m. Le vecteur z_m^{RO} est solution de l'équation suivante :

$$Q_{m-1}(\beta e_1^{(m)} - H_m z_m^{RO}) = 0$$

qui s'écrit encore :

$$z_m^{RO} = T_m^{-1}(Q_{m-1} \beta e_1^{(m)}).$$

La matrice H_m étant supposée inversible, T_m n'est pas singulière et l'équation précédente admet une solution unique. D'après la relation (1.15) et comme la base de Krylov est orthonormale, les résidus successifs de la méthode **FOM** sont deux à deux orthogonaux. En notant $\tilde{\gamma}_1 = \beta$ et

$$\Gamma_{m+1}^{(i)} = Q_i \beta e_1^{(m+1)} = \begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_i \\ \tilde{\gamma}_{i+1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{pour } i = 1, \ldots, m, \tag{1.19}$$

nous obtenons $(z_m^{RO})_m = \tilde{\gamma}_m / \tilde{h}_{m,m}$, où $\tilde{h}_{m,m}$ est le coefficient (m,m) de $Q_{m-1} H_m = T_m$. La matrice T_m étant triangulaire supérieure et non singulière, il est clair que $\tilde{h}_{m,m}$ est non nul. À partir de la relation (1.15), nous exprimons le résidu :

$$r_m^{RO} = (-h_{m+1,m} \tilde{\gamma}_m) / \tilde{h}_{m,m} v_{m+1} \tag{1.20}$$

ainsi que sa norme

$$\|r_m^{RO}\| = |h_{m+1,m}\tilde{\gamma}_m/\tilde{h}_{m,m}|.$$

En remplaçant $\bar{H}_m$ par $Q_m^T\bar{R}_m$ dans l'équation (1.12) et sachant que la matrice Q_m est orthonormale, nous obtenons :

$$\bar{R}_m^T\bar{R}_m z_m^{RM} = R_m^T R_m z_m^{RM} = \bar{R}_m^T(Q_m\beta e_1^{(m+1)}).$$

À l'aide de la relation (1.19), z_m^{RM} s'écrit :

$$z_m^{RM} = R_m^{-1}(\gamma_1,\ldots,\gamma_m)^T.$$

La solution approchée x_m^{RM} s'exprime à présent :

$$x_m^{RM} = x_0 + V_m R_m^{-1}(\gamma_1,\ldots,\gamma_m)^T \tag{1.21}$$

et le résidu correspondant

$$\begin{aligned} r_m^{RM} &= V_{m+1}(\beta e_1^{(m+1)} - \bar{H}_m z_m^{RM}) \\ &= V_{m+1}Q_m^T(\Gamma_{m+1}^{(m)} - \bar{R}_m z_m^{RM}) \\ &= V_{m+1}Q_m^T(0,\ldots,0,\tilde{\gamma}_{m+1})^T \end{aligned}$$

ce qui finalement permet d'exprimer la norme du résidu assez simplement :

$$\|r_m^{RM}\|_2 = |\tilde{\gamma}_{m+1}|. \tag{1.22}$$

De par la définition même de r_m^{RM}, nous avons que les résidus successifs sont A-orthogonaux.

Les solutions approchées **RO** et **RM** sont reliées entre elles par les sinus et cosinus s_m et c_m des angles des rotations de Givens $\Omega_m^{(m+1)}$, de la façon suivante [49, 12] :

$$x_m^{RM} = s_m^2 x_{m-1}^{RM} + c_m^2 x_m^{RO}.$$

Il en va de même pour les résidus

$$r_m^{RM} = s_m^2 r_{m-1}^{RM} + c_m^2 r_m^{RO}. \tag{1.23}$$

D'après l'équation (1.19), nous avons également :

$$\begin{cases} \gamma_m = c_m\tilde{\gamma}_m \\ \tilde{\gamma}_{m+1} = -s_m\tilde{\gamma}_m \end{cases} \tag{1.24}$$

avec $\tilde{\gamma}_1 = \beta$, ce qui permet de déduire d'après (1.22) :

$$\|r_m^{RM}\|_2 = |s_m|\ \|r_{m-1}^{RM}\|_2.$$

Le vecteur r_m^{RO} étant orthogonal au vecteur r_m^{RM}, l'équation précédente et la formule (1.23), nous donnent

$$\|r_m^{RM}\|_2 = |c_m|\ \|r_m^{RO}\|_2.$$

Ainsi lorsque GMRES stagne, $|s_m|$ est très proche de 1 et la courbe de la norme du résidu de FOM présente une pente ascendante car $|c_m|$ devient très petit et $\|r_m^{RO}\|_2$ augmente. Au contraire, lorsque GMRES montre une accélération nette, la norme de son résidu décroît fortement et la norme du résidu de FOM devient très proche de celle de GMRES. Les convergences de FOM et GMRES sont donc très liées.

Du fait de sa condition de minimisation, la norme du résidu de GMRES décroît de façon monotone. Cette propriété fait que GMRES est souvent préféré à FOM, dont la norme du résidu présente des comportements oscillatoires.

Les valeurs s_m et c_m représentent en fait le sinus et le cosinus de l'angle suivant :

$$\angle(r_{m-1}^{RM}, A\mathrm{K}_m) = \angle(\mathrm{K}_m, A\mathrm{K}_m).$$

Pour plus de détails voir l'article de Ernst et Eiermann [12].

Nous étudions ci-dessous le cas particulier des systèmes symétriques.

Méthodes implicites

De nombreuses simplifications apparaissent lorsque la matrice A est symétrique. H_m est alors également symétrique. Étant une matrice de Hessenberg supérieure, cela implique qu'elle est tridiagonale, les diagonales supérieures et inférieures étant identiques. Cette propriété va nous permettre de construire la solution de façon implicite. Nous réécrivons la matrice H_m ainsi :

$$H_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_m \\ & & & \beta_m & \alpha_m \end{pmatrix}.$$

Le processus d'Arnoldi est simplifié car seuls les vecteurs v_m et v_{m-1} sont nécessaires à la construction de v_{m+1}. Néanmoins, le calcul de x_m^{RO} et x_m^{RM} nécessite le stockage de V_{m+1} d'après l'équation (1.4) et donc de tous les vecteurs $v_1, \ldots, v_{m+1}$. Nous allons voir comment y remédier.

Algorithme RO : SDFOM. Nous considérons d'abord la méthode **RO**. En supposant que la factorisation LU de la matrice H_m existe, que nous mettons sous la forme $A_m S_m$, nous obtenons :

$$H_m = \underbrace{\begin{pmatrix} 1 & & & \\ \lambda_2 & 1 & & \\ & \ddots & \ddots & \\ & & \lambda_m & 1 \end{pmatrix}}_{A_m} \times \underbrace{\begin{pmatrix} \eta_1 & \beta_2 & & \\ & \eta_2 & \beta_3 & \\ & & \ddots & \ddots \\ & & & \eta_m \end{pmatrix}}_{S_m}.$$

En fixant $\lambda_1 = 0$, $\beta_1 = 0$ et $\eta_1 = \alpha_1$, les coefficients λ_k et η_k suivants se calculent par récurrence:

$$\left\{ \begin{array}{lcl} \lambda_k & = & \beta_k/\eta_{k-1} \\ \eta_k & = & \alpha_k - \lambda_k\beta_k \end{array} \right. \quad \forall k = 2, \ldots, m. \tag{1.25}$$

Nous réécrivons la solution approchée x_m^{RO} sous la forme

$$\begin{aligned} x_m^{RO} &= x_0 + \underbrace{V_m S_m^{-1}}_{P_m} \underbrace{A_m^{-1}(\beta e_1^{(m)})}_{d_m} \\ &= x_0 + P_m d_m \end{aligned}$$

et définissons les vecteurs $p_1, \ldots, p_m$ comme les vecteurs colonnes de P_m. Ils vérifient:

$$p_m = \frac{1}{\eta_m}(v_m - \beta_m p_{m-1}).$$

Enfin, du fait de la structure de A_m, nous pouvons mettre d_m sous la forme:

$$d_m = \begin{pmatrix} d_{m-1} \\ \delta_m \end{pmatrix}$$

avec $\delta_m = -\lambda_m \delta_{m-1}$, ce qui permet d'exprimer x_m^{RO} en fonction de x_{m-1}^{RO},

$$x_m^{RO} = x_{m-1}^{RO} + \delta_m p_m$$

et de décrire complètement l'algorithme, que nous appelons **SDFOM** pour Symmetric Direct **FOM** et qui ne s'applique qu'aux matrices symétriques.

Algorithme 1.5 (RO) $[x_m, r_m]$ = *SDFOM(A,x_0,m)*

1. *Initialisation:* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $v_1 = r_0/\beta$
   $\lambda_1 = \beta_1 = 0$, $\delta_1 = \beta$ *et* $p_0 = 0$

2. *Boucle principale:*
   Pour $j = 1, \ldots, m$ *Faire*

   (a) $w = Av_j - \beta_j v_{j-1}$

   (b) $\alpha_j = (w, v_j)$ *et* $w = w - \alpha_j v_j$

   (c) $\beta_{j+1} = \|w\|_2$
   Si $\beta_{j+1} \neq 0$ *alors* $v_{j+1} = w/\beta_{j+1}$ *sinon stop*

   (d) Si $j > 1$ *alors* $\lambda_j = \beta_j/\eta_{j-1}$ *et* $\delta_j = -\lambda_j \delta_{j-1}$

   (e) $\eta_j = \alpha_j - \lambda_j\beta_j$ *et* $p_j = (v_j - \beta_j p_{j-1})/\eta_j$

   (f) Calcul de la solution approchée
   $x_j = x_{j-1} + \delta_j p_j$

   Fin Pour

Ce nouvel algorithme nécessite beaucoup moins de stockage que la version initiale, puisque seuls les vecteurs v_j, v_{j-1} et p_{j-1} doivent être stockés pour calculer la nouvelle direction p_j. La matrice $\bar{H}_m$ n'est pas stockée mais remplacée par le stockage d'un petit nombre de coefficients. Cet algorithme pose néanmoins des problèmes dans la factorisation LU de H_m car des pivots nuls peuvent survenir. Heureusement, une version avec pivotage partiel existe.

Cet algorithme est mathématiquement équivalent à celui de **FOM** lorsque les matrices considérées sont symétriques. Les résidus sont donc deux à deux orthogonaux. De plus, nous avons :

Proposition 1.7 *Les directions p_i sont deux à deux A-orthogonales.*

Preuve :

$$\begin{aligned} P_m^T A P_m &= S_m^{-T} V_m^T A V_m S_m^{-1} \\ &= S_m^{-T} H_m S_m^{-1} \\ &= S_m^{-T} A_m S_m S_m^{-1} \\ &= S_m^{-T} A_m. \end{aligned}$$

A étant symétrique, les matrices $P_m^T A P_m$ et $S_m^{-T} A_m$ le sont aussi. Or la matrice $S_m^{-T} A_m$ est une matrice triangulaire supérieure. Elle est donc diagonale et les directions p_i sont deux à deux A-orthogonales. ■

On dit également que les directions p_i sont conjuguées. Cette propriété nous permet de réécrire l'algorithme de **SDFOM** et d'en dériver l'algorithme du **CG**.

Algorithme RO : le CG. Les coefficients α_j et β_j ci-dessous n'ont plus la même signification que ceux définis dans H_m. Nous exprimons à nouveau x_{j+1} et le résidu correspondant :

$$x_{j+1} = x_j + \alpha_j p_j \quad \text{et} \quad r_{j+1} = r_j - \alpha_j A p_j$$

avec la nouvelle direction :

$$p_{j+1} = r_{j+1} + \beta_j p_j.$$

La condition d'orthogonalité des résidus r_{j+1} et r_j nous permet de caractériser le coefficient α_j qui vaut :

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, r_j)}.$$

Comme $(Ap_j, r_j) = (Ap_j, p_j - \beta_{j-1} p_{j-1}) = (Ap_j, p_j)$, α_j s'écrit :

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, p_j)}.$$

Le coefficient β_j se calcule à l'aide de la condition de A-orthogonalité entre les vecteurs p_{j+1} et p_j :

$$\beta_j = -\frac{(r_{j+1}, Ap_j)}{(p_j, Ap_j)} = -\frac{(r_{j+1}, \frac{1}{\alpha_j}(r_j - r_{j+1}))}{\frac{(r_j, r_j)}{\alpha_j}} = \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)}.$$

L'algorithme correspondant est l'algorithme du Gradient Conjugué de Hestenes et Stiefel [23], qui est décrit ci-dessous :

Algorithme 1.6 (RO) $[x_m, r_m] = CG(A, x_0, m)$

1. *Initialisation :* $r_0 = b - Ax_0$, $p_0 = r_0$
2. *Boucle principale*
   Pour $j = 1, \ldots, m-1$ *Faire*
   (a) $\alpha_j = (r_j, r_j)/(Ap_j, p_j)$
   (b) $x_{j+1} = x_j + \alpha_j p_j$
   (c) $r_{j+1} = r_j - \alpha_j A p_j$
   (d) $\beta_j = (r_{j+1}, r_{j+1})/(r_j, r_j)$
   (e) $p_{j+1} = r_{j+1} + \beta_j p_j$
   Fin Pour

L'algorithme du **CG** a été dérivé pour l'ensemble des matrices symétriques. Cependant, des divisions par zéro peuvent survenir dès que $(Ap_j, p_j) = 0$. La direction p_j ne peut être nulle que si le résidu r_j est lui-même nul et que la solution a été trouvée. En imposant que la matrice A soit symétrique définie positive, nous aurons que $(Ax, x) \neq 0$ pour tout vecteur x non nul et donc que l'algorithme du **CG** ne peut s'arrêter prématurément.

Nous pouvons procéder à présent de la même façon pour les méthodes **PM** et déduire **SDGMRES** de la méthode **GMRES**.

Algorithme RM : SDGMRES. La matrice $\bar{H}_m$ étant tridiagonale, la matrice $\bar{R}_m = Q_m \bar{H}_m$ est triangulaire supérieure à trois diagonales que nous pouvons écrire ainsi

$$\bar{R}_m = \begin{pmatrix} \rho_1 & \sigma_2 & \tau_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \tau_m \\ & & & \rho_{m-1} & \sigma_m \\ & & & & \rho_m \\ & & & & 0 \end{pmatrix}.$$

Comme cela a été fait pour l'algorithme de **FOM**, nous allons reformuler la mise à jour de la solution approchée x_m^{RM}. Elle s'écrit :

$$x_m^{RM} = x_0 + \underbrace{V_m R_m^{-1}}_{P_m} (\gamma_1, \ldots, \gamma_m)^T.$$

Si $p_1, \ldots, p_m$ sont les colonnes de la matrice P_m, du fait de la forme de la matrice R_m, nous avons :

$$p_m = (v_m - \sigma_m p_{m-1} - \tau_m p_{m-2})/\rho_m$$

ce qui nous permet d'exprimer x_m^{RM} en fonction de x_{m-1}^{RM} :

$$x_m^{RM} = x_{m-1}^{RM} + \gamma_m p_m.$$

Les équations (1.24), nous permettent de calculer γ_m récursivement.

Nous appelons ce nouvel algorithme **SDGMRES** pour Symmetric Direct **GMRES**.

Algorithme 1.7 (RM) $[x_m, r_m]$ = *SDGMRES(A,x_0,m)*

1. *Initialisation :* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $v_1 = r_0/\beta$, $\tilde{\gamma}_1 = \beta$
   $p_{-1} = p_0 = 0$ *et* $c_{-1} = c_0 = s_{-1} = s_0 = 0$

2. *Boucle principale*
   Pour $j = 1, \ldots, m$ *Faire*

   (a) *Calcul de la* $m^{ième}$ *colonne de* H_m :
   $w = Av_j - \beta_j v_{j-1}$
   $\alpha_j = (w, v_j)$ *et* $w = w - \alpha_j v_j$.
   $\beta_{j+1} = \|w\|$, *si* $\beta_{j+1} \neq 0$ *alors* $v_{j+1} = w/\beta_{j+1}$ *sinon stop*

   (b) *Calcul de la* $m^{ième}$ *colonne de* R_m :
   $\tau_j = s_{j-2}\beta_j$ *et* $\epsilon = c_{j-2}\beta_j$
   $\sigma_j = c_{j-1}\epsilon + s_{j-1}\alpha_j$ *et* $\epsilon = -s_{j-1}\epsilon + c_{j-1}\alpha_j$
   Calcul des coefficients c_j *et* s_j *correspondant à* ϵ *et* β_{j+1}
   $\rho_j = c_j\epsilon + s_j\beta_{j+1}$

   (c) *Mise à jour de* Γ :
   $\gamma_j = c_j\tilde{\gamma}_j$ *et* $\tilde{\gamma}_{j+1} = -s_j\tilde{\gamma}_j$

   (d) *Calcul de la solution approchée*
   $p_j = (v_j - \sigma_j p_{j-1} - \tau_j p_{j-2})/\rho_j$
   $x_j = x_{j-1} + \gamma_j p_j$

   Fin Pour

Comme pour l'algorithme de **SDFOM**, celui-ci est moins onéreux en coût de stockage que l'algorithme initial. Comme dans le cas de **GMRES**, les résidus sont A-orthogonaux. Nous avons en outre :

Proposition 1.8 *Les directions* Ap_i *sont orthogonales.*

Preuve : Nous avons

$$\begin{aligned} AP_m &= AV_m R_m^{-1} \\ &= V_{m+1}\bar{H}_m R_m^{-1} \\ &= V_{m+1}Q_m^T. \end{aligned}$$

Les matrices V_{m+1} et Q_m étant orthogonales, nous avons

$$(AP_m)^T(AP_m) = I_m$$

et les vecteurs Ap_j sont bien orthogonaux. ■

Algorithme RM: le CR. En procédant de la même façon que pour le calcul du **CG**, nous exprimons la solution approchée sous la forme $x_m^{RM} = x_0 + \alpha_j p_j$ avec $p_{j+1} = r_{j+1} + \beta_j p_j$. Après des calculs analogues à ceux effectués pour le **CG**, nous obtenons:

$$\alpha_j = \frac{(r_j, Ar_j)}{(Ap_j, Ap_j)} \tag{1.26}$$

et

$$\beta j = \frac{(r_{j+1}, Ar_{j+1})}{(r_j, Ar_j)}. \tag{1.27}$$

L'algorithme final se nomme l'algorithme Conjugate **R**esidual (**CR**) [49] et est décrit ci-dessous:

Algorithme 1.8 (RM) $[x_m, r_m] = CR(A, x_0, m)$

1. *Initialisation:* $r_0 = b - Ax_0$, $p_0 = r_0$
2. *Corps de l'algorithme*
   Pour $j = 1, \ldots, m-1$ *Faire*
   (a) $\alpha_j = (r_j, Ar_j)/(Ap_j, Ap_j)$
   (b) $x_{j+1} = x_j + \alpha_j p_j$
   (c) $r_{j+1} = r_j - \alpha_j Ap_j$
   (d) $\beta_j = (r_{j+1}, Ar_{j+1})/(r_j, Ar_j)$
   (e) $p_{j+1} = r_{j+1} + \beta_j p_j$
   Fin Pour

De la même façon que pour la méthode du **CG**, la méthode du **CR** a été dérivée parce que la matrice A est symétrique. Cependant des pannes peuvent survenir lorsque (Ap_j, Ap_j) ou (Ar_j, r_j) s'annulent. En choisissant A définie positive, de telles pannes ne peuvent se présenter.

Nous venons de voir en détail les méthodes **RO** et **RM**. Nous utiliserons principalement les méthodes **FOM** et **GMRES** dans la suite de la thèse, car elles s'appliquent à des systèmes non symétriques. Cependant le nombre d'opérations ainsi que le stockage croissent de façon au moins linéaire avec la dimension m du sous-espace. Une façon d'y remédier mais aussi de pallier à la perte d'orthogonalité des vecteurs de la base est de redémarrer le procédé. Ainsi, après un certain nombre d'itérations m fixé à l'avance, nous calculons la solution x_m et la prenons comme point de départ x_0 du procédé suivant. Les méthodes **FOM** et **GMRES** alors notées **FOM**(m) et **GMRES**(m) sont dites des méthodes redémarrées. Ce redémarrage induit une perte partielle des informations des itérations précédentes et s'accompagne d'un ralentissement de la convergence. Par ailleurs, cette convergence est plus complexe à analyser et moins bien comprise que celle des versions non redémarrées de ces méthodes. Ce sont précisément les méthodes redémarrées que nous tenterons d'accélérer dans la suite de la thèse.

Nous survolons dans le paragraphe suivant les méthodes **RQO** et **RQM**.

1.2.2 Méthodes issues d'un produit scalaire différent

Nous venons de voir qu'une façon de remédier au coût prohibitif de stockage et de calcul des méthodes **FOM** et **GMRES** étaient de les redémarrer. Nous allons voir une autre alternative.

Méthodes IOM et QGMRES

Une autre façon consiste à tronquer le processus d'Arnoldi et à ne rendre le vecteur v_m orthogonal qu'aux k vecteurs $(v_{m-j})_{j=1,\ldots,k}$ précédents. La matrice V_{m+1} n'est alors plus orthogonale et nous calculons une solution approchée de type **RQO** ou **RQM**. Seul le processus d'Arnoldi est simplifié et remplacé par le processus d'Arnoldi tronqué :

Algorithme 1.9 $[V_{m+1},\bar{H}_m]$ = *Arnoldi-tronqué*(A,r_0,m,k)

1. *Initialisation : Calcul de* $v_1 = r_0/\|r_0\|_2$
2. *Algorithme de Gram-Schmidt modifié*
   Pour $j = 1,\ldots,m$ *Faire*
   (a) $w = Av_j$
   (b) *Pour* $i = \max\{1, j-k+1\},\ldots,j$ *Faire*
   $h_{i,j} = (w, v_i)$
   $w = w - h_{i,j}v_i$
   Fin Pour
   (c) $h_{j+1,j} = \|w\|_2$
   (d) *si* $h_{j+1,j} \neq 0$ *alors* $v_{j+1} = w/h_{j+1,j}$ *sinon stop*
   Fin Pour
3. *Définir* $V_{m+1} = [v_1,\ldots,v_{m+1}]$, $\bar{H}_m = \{h_{i,j}\}$

Ce nouvel algorithme donne lieu à deux nouvelles versions de **FOM** et **GMRES** dénommées respectivement **I**ncomplete **O**rthogonal **M**ethod (**IOM**) due à Saad [42] et **Q**uasi **GMRES** (**QGMRES**) due à Saad et Wu [54]. Les matrices $\bar{H}_m$ et $\bar{R}_m$ sont à présent des matrices bande de largeur $k+1$, ce qui permet de réduire le nombre de produits scalaires (w, v_i) à k au lieu de i à chaque itération i. Néanmoins le calcul de la solution approchée (1.4) nécessite de garder l'ensemble des vecteurs de la base de Krylov, même si ceux-ci ne forment pas une base orthonormale.

De la même façon que nous avons dérivé la méthode **SDFOM** de **FOM** en factorisant la matrice H_m sous forme LU, nous dérivons la méthode **DIOM** pour **D**irect **IOM** de **IOM**. La nouvelle direction p_m se calcule alors à l'aide de v_m et des $k-2$ directions précédentes $(p_{m-j})_{j=1,\ldots,k-2}$. De même, en notant $P_m = V_m R_m^{-1}$ la matrice des directions, nous pouvons dériver la version directe de **QGMRES** notée **DQMGRES** pour **D**irect **QGMRES**.

Les nouveaux algorithmes **DIOM** et **DQGMRES** sont respectivement mathématiquement équivalents à **IOM** et **QGMRES**, mais bien sûr pas à **FOM** et **GMRES**, puisque la matrice V_{m+1} n'est pas orthonormale.

Méthodes de Hessenberg et CMRH

Nous présentons à présent deux autres méthodes de type **RQO** et **RQM**. Les vecteurs u_i du processus de Hessenberg généralisé sont connus à l'avance et pris égaux aux vecteurs $e_i^{(n)}$. Le processus de Hessenberg est réellement simplifié puisque tous les produits scalaires sont remplacés par le calcul d'une composante d'un vecteur. Le processus résultant est le processus de Hessenberg décrit ci-dessous :

Algorithme 1.10 $[V_{m+1},\bar{H}_m]$ = *Hessenberg(A,*r_0*,m,*U_m*)*

1. *Initialisation :* $v_1 = r_0$ *et* $\eta_1 = (v_1)_1$
2. *Boucle principale*
   Pour $j = 1, \ldots, m$ *Faire*
   (a) $w = Av_j$
   (b) *Pour* $i = 1, \ldots, j$ *Faire*
   $h_{i,j} = w_i/\eta_i$
   $w = w - h_{i,j}v_i$
   Fin Pour
   (c) *Choix de* $h_{j+1,j}$ *facteur normalisateur de* v_{j+1}
   (d) $v_{j+1} = w/h_{j+1,j}$ *et* $\eta_{j+1} = (v_{j+1})_{j+1}$
   Fin Pour
3. *Définir* $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $\bar{H}_m = \{h_{i,j}\}$

Du fait de la possibilité d'une panne (*breakdown*) ou d'une pseudo-panne (*near-breakdown*), c'est-à-dire η_j proche de zéro, une version avec pivotage existe, où η_j est prise comme la valeur de la composante de v_j qui est la plus grande en valeur absolue par rapport à toutes les autres composantes.

La méthode de type **RQO** issue de ce procédé de Hessenberg s'appelle la méthode de Hessenberg. Il existe également une version tronquée dénommée **THM** pour **T**runcated **H**essenberg **M**ethod ainsi que sa version directe **DTHM**. La méthode de type **RQM** quant à elle s'appelle **CMRH** et est due à Sadok [56]. De la même façon que précédemment, nous pouvons dériver la version tronquée **TCMRH** et sa version directe **DTCMRH**. Ces méthodes sont détaillées dans [24].

Algorithmes Bi-CG, CGS et QMR. Nous présentons un dernier groupe de méthodes de type **RQO** et **RQM**, qui construisent les vecteurs $(u_i)_{1\leq i\leq m}$ de façon progressive. Ces vecteurs constituent une base du sous-espace $\mathrm{K}_m(A^T, u_1)$ avec $(r_0, u_1) \neq 0$, tout en formant une base biorthogonale aux vecteurs $(v_i)_{1\leq i\leq m}$. Ils vérifient :

$$\mathrm{K}_k(A^T, u_1) = \mathrm{Span}\{u_1, \ldots, u_k\} \ \forall k = 1, \ldots, L$$

et

$$(v_j, u_i) = (u_j, v_i) = v_i^T u_j = \eta_j \delta_{i,j} \ \forall j = 1, \ldots, m \text{ et } i = 1, \ldots, j.$$

Ceci signifie que la matrice $L_m = U_m^T V_m$ (voir 1.14) est diagonale et $\bar{H}_m$ tridiagonale. Nous avons en outre la relation suivante :

$$A^T U_m = U_{m+1} \widetilde{\bar{H}}_m^T.$$

En choisissant $L_m = I_m$, nous obtenons $\tilde{\bar{H}}_m = \bar{H}_m$. Le processus de Hessenberg généralisé est remplacé par la procédure de biorthogonalisation de Lanczos décrite ci-dessous :

Algorithme 1.11 (RQO) $[V_{m+1}, \bar{T}_m]$ = *Lanczos-biorthogonalisation(A,r₀,m)*

1. *Initialisation :* $v_1 = r_0$. *Choisir* u_1 *tel que* $(v_1, u_1) = 1$
   $\beta_1 = \delta_1 = 0$ *et* $v_0 = u_0 = 0$

2. *Boucle principale :*
   Pour $j = 1, \ldots, m$ *Faire*

   (a) $\alpha_j = (Av_j, u_j)$

   (b) $w = Av_j - \alpha_j v_j - \beta_j v_{j-1}$

   (c) $\tilde{w} = A^T u_j - \alpha_j u_j - \beta_j u_{j-1}$

   (d) $\delta_{j+1} = |(w, \tilde{w})|^{1/2}$
   Si $\delta_{j+1} = 0$ *alors stop*

   (e) $\beta_{j+1} = (w, \tilde{w})/\delta_{j+1}$

   (f) $v_{j+1} = w/\beta_{j+1}$

   (g) $u_{j+1} = \tilde{w}/\delta_{j+1}$

   Fin Pour

3. *Définir* $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $\bar{T}_m = tridiag(\delta_i, \alpha_i, \beta_i)$

La méthode de type **RQO** utilisant le processus de biorthogonalisation de Lanczos donne lieu à la méthode de **Lanczos**. La méthode dérivée de la méthode de **Lanczos** et reformulée de la même façon que la méthode du **CG** est dérivée de la méthode **FOM**, s'appelle la méthode du **Bi-CG** [15]. Lorsque la matrice utilisée est symétrique, cette méthode est mathématiquement équivalente à la méthode **SDFOM**. De la même façon, la méthode **RQM** dérivée du processus de biorthogonalisation de Lanczos peut être dérivée comme **SDGMRES** est dérivé de **GMRES**. Réécrite comme la méthode du **CR** est réécrite à partir de **SDGMRES**, elle donne naissance à la méthode **QMR** signifiant **Q**uasi **M**inimal **R**esidual. Outre la possibilité d'une panne (*breakdown*) pouvant survenir lorsque le sous-espace $\mathrm{K}_m(A^T, u_1)$ est invariant ou bien lorsque $v_m \perp u_m$, les méthodes **Bi-CG** et **QMR** présentent l'inconvénient d'utiliser des produits matrice vecteur avec la matrice A^T. Un algorithme issu du **Bi-CG** a été développé qui n'utilise plus de produits matrice vecteur avec A^T. C'est le **C**onjugate **G**radient **S**quare ou **CGS** [57], qui n'est pas mathématiquement équivalent à la méthode du **Bi-CG**. Du fait du comportement oscillatoire de la norme du résidu de la méthode **CGS** dont le résidu ne vérifie pas de condition de minimisation, Van der Vorst a développé une méthode qui minimise de

façon locale la norme du résidu : la méthode du **Bi-CGSTAB** pour **Bi-CG Stab**ilized [59]. Enfin, il est également possible de mettre en œuvre la méthode du **Bi-CG** sans effectuer de produits matrice vecteur avec la matrice A^T [5], en utilisant simultanément les méthodes **CGS** et **Bi-CGSTAB**, qui permettent de retrouver les coefficients de la récurrence du **Bi-CG**. Une version sans produit matrice vecteur avec la matrice A^T a également été développée par Freund à partir de **QMR**, qui se nomme **T**ranspose **F**ree **QMR** ou **TFQMR** [18].

Avec ces divers algorithmes, il existe également différentes stratégies, comme celle du *look-ahead* pour **QMR** [39], qui pallient aux pannes et pseudo-pannes.

Les méthodes les plus utilisées parmi les méthodes de type **RQO** et **RQM** sont le **CGS**, le **Bi-CGSTAB** et **QMR**. Ces méthodes ont un grand intérêt qui est celui d'une implantation facile et plus aisée que pour les méthodes **FOM**(m) et **GMRES**(m). De plus, elles construisent un sous-espace de Krylov de dimension beaucoup plus grande que les méthodes **FOM**(m) et **GMRES**(m), qui ne construisent à chaque fois, qu'un sous-espace de dimension m.

Nous détaillons dans le paragraphe suivant l'environnement parallèle dans lequel nous nous plaçons, les opérations principales des méthodes de Krylov et regardons comment celles-ci peuvent être mises en œuvre.

1.3 Mises en œuvre des méthodes de Krylov dans un environnement parallèle

L'environnement parallèle sur lequel nous nous plaçons sont les machines à architecture **MIMD** à mémoire distribuée. Ces machines peuvent être une machine parallèle comme le T3E ou bien un réseau de stations de travail. Par la suite, nous appellerons indifféremment processeur aussi bien les processeurs d'une machine multi-processeurs que les stations d'un ensemble de stations inter-connectées entre elles. Rappelons tout d'abord la classification des ordinateurs due à Flynn [16], qui est de loin la plus connue. Cette classification décrit le modèle d'exécution de la machine utilisée. Elle se fonde sur deux caractéristiques : le flux d'instructions (Single ou Multiple Instruction) et le flux de données (Single ou Multiple Data) sur lesquelles s'effectuent les instructions précédentes. Suivant ces deux critères, cette classification donne lieu à quatre catégories :

- **SISD** (Single Instruction stream Single Data stream) : correspond aux machines monoprocesseur traitant les données de façon séquentielle l'une après l'autre,
- **SIMD** (Single Instruction stream Multiple Data stream) : un flux d'instructions est appliqué à plusieurs données. Les processeurs exécutent de façon synchrone la même instruction sur des données différentes qui leur sont locales,
- **MISD** (Multiple Instruction stream Single Data stream) : chaque processeur effectue des instructions différentes sur une même donnée. N'existe pas en pratique,
- **MIMD** (Multiple Instruction stream Multiple Data stream) : les processeurs effectuent de façon asynchrone des opérations différentes sur des données différentes.

Parmi les architectures **SIMD**, citons les machines MASPAR MP-1 et MP-2, les CM-2 et CM-200. Parmi les architectures **MIMD**, citons le T3E, le T3D, la CM5, l'Intel Paragon, la Ferme d'Alpha, le C90, l'Origin 200 et les réseaux de stations. Celles-ci nous intéressent particulièrement car nous utiliserons un T3E par la suite.

Outre leur modèle d'exécution, **SIMD** ou **MIMD**, les machines parallèles sont classées suivant le critère d'organisation de leur mémoire. Nous avons :

- les machines à mémoire partagée : les processeurs accèdent à une mémoire commune dite partagée. En général, le réseau d'interconnexion entre la mémoire et les processeurs est un simple bus ou un Crossbar et les processeurs sont synchronisés par les accès à la mémoire. En général, pour de telles machines, le nombre de processeurs est très inférieur à celui que l'on peut trouver sur les machines à mémoire distribuée. Ceci est dû à des questions d'efficacité qui est limitée par l'accès de plusieurs processeurs à la même mémoire,

- les machines à mémoire distribuée : chaque processeur possède sa propre mémoire locale et le partage des données se fait par l'envoi de messages. Ces messages représentent l'unique façon de synchroniser les processeurs entre eux. En général, le réseau d'interconnexion est à topologie régulière comme l'anneau, la grille et le tore 2-D, 3-D ainsi que l'hypercube,

- les machines à mémoire virtuellement partagée : certaines machines à mémoire distribuée permettent un adressage global des mémoires locales. Celui-ci peut se situer soit au niveau *hardware* soit au niveau logiciel.

Une fois le modèle d'exécution de la machine et l'architecture de sa mémoire spécifiés, il convient de préciser le modèle de programmation que nous utiliserons. Bien que ceux qui existent, soient directement inspirés des deux modèles d'exécution **SIMD** et **MIMD**, il est possible d'utiliser un modèle de programmation, à priori non adapté au modèle d'exécution de la machine. Les deux principaux modèles d'exécution sont :

- le parallélisme sur le contrôle : le programmeur doit lui-même décomposer son application en processus ou tâches élémentaires, et les répartir sur les processeurs. Il doit aussi gérer les communications entre les différentes tâches. La programmation peut alors être soit **SPMD** (un seul exécutable pour l'ensemble des processus) ou bien **MPMD** (plusieurs exécutables pour l'ensemble des processus, certains pouvant se voir attribuer le même exécutable). Dans un cas comme dans l'autre, des conditions de branchement sur le numéro ou rang du processus permettent de déterminer les différentes actions qui seront effectuées par chacun des processus,

- le parallélisme sur les données : un seul processus agit sur un ensemble de données de façon concurrente. La programmation est **SPMD**.

Nous utiliserons dans notre cas le T3E de l'I.D.R.I.S. [1] qui est une machines **MIMD** à mémoire distribuée. Le modèle de programmation choisi est **SPMD** : les codes sont écrits en Fortran et la bibliothèque de messages utilisée est **MPI** [31].

1. Institut du Développement et des Ressources en Informatique Scientifique, Bâtiment 506 - B.P. 167, 91403 ORSAY CEDEX, FRANCE

Sur les machines à mémoire distribuée, les traitements les plus pénalisants sont les communications. Le coût d'une communication locale, à savoir un processeur accédant à sa propre mémoire, est peu élevé comparé au coût d'accès d'un processeur à une donnée située dans la mémoire locale d'un autre processeur. En effet, cela nécessite la construction et l'envoi de messages devant transiter par le réseau d'interconnexion, ainsi que l'accès à la mémoire distante. Un programme se doit donc de minimiser les accès distants. Mais il doit faire en sorte de ne laisser aucun processeur inactif pendant une partie ou la totalité de l'exécution. Ceci peut être minimisé grâce à un bon équilibrage des charges, directement lié à la répartition des données. Une bonne répartition des données est une des clés en vue d'atteindre de bonnes performances. Le recouvrement des calculs et des communications permet également de diminuer le coût de ces mêmes communications.

1.3.1 Répartition des données

Rappelons tout d'abord quelques généralités. Nous considérons un problème de dimension n. Pour le résoudre, nous construisons un sous-espace de Krylov de dimension m et utilisons dans le cas des méthodes directes, une récurrence de longueur k.

Dans notre cas, les principales données que nous avons à stocker sont la matrice creuse, les vecteurs v_i et u_i du processus de Hessenberg généralisé, ainsi que le second membre b, le vecteur x approchant la solution et le résidu correspondant r. Si l'on considère des méthodes de Krylov à récurrence courte, de longueur 3 pour les méthodes issues du processus de biorthogonalisation de Lanczos et de longueur k pour les méthodes directes ou dérivées des méthodes directes, issues d'un processus d'orthogonalisation incomplet de dimension k, elles nécessitent le stockage respectif de :

- 9 vecteurs de dimension n,
- $2k + 4$ vecteurs de dimension n.

En ce qui concerne les méthodes non directes ou n'impliquant pas de récurrences courtes, comme les méthodes **FOM**, **GMRES** ou **CMRH**, nous prenons en compte leur version redémarrée. Elles nécessitent alors un stockage de $m + 4$ vecteurs de dimension n.

Si l'on considère qu'une matrice creuse possède un maximum de 3% éléments non nuls, nous devons stocker un maximum de $\frac{3n^2}{100}$ éléments pour la matrice A. Afin que le stockage des vecteurs u_i et v_i ne soit pas prépondérant par rapport au coût de stockage de la matrice, nous devons avoir, suivant l'algorithme utilisé,

$$9 < \frac{3n}{100} \quad \text{ou} \quad 2k+4 < \frac{3n}{100} \quad \text{ou} \quad m+4 < \frac{3n}{100}.$$

Nous supposerons ces relations vérifiées dans la suite de la thèse. De la même façon, il apparaît que le stockage de la matrice de Hessenberg et des quelques scalaires des différents algorithmes est négligeable par rapport à celui de la matrice si $m^2 \ll \frac{3n^2}{100}$. Or nous ne prendrons jamais m supérieur à 50 et toujours n supérieur à 1000. Dans ces cas, le stockage de la matrice de Hessenberg est donc négligeable : elle n'est pas distribuée sur les processeurs, mais stockée de façon redondante sur chacun. Il en est de même des scalaires.

De ce fait, la répartition de la matrice sur les processeurs est cruciale. Il ne faut pas négliger la répartition des vecteurs qui doit rester cohérente avec celle de la matrice et ne doit pas induire des coûts de communications excessifs dans l'ensemble des opérations présentes dans les méthodes de Krylov.

1.3.2 Principales opérations

Si nous répertorions les principales opérations vues dans les méthodes de Krylov des paragraphes précédents, nous avons :

- le produit matrice vecteur $y = Ax$,
- le produit matrice vecteur avec la matrice transposée A^T, Nous n'en parlerons pas dans cette section, car les méthodes actuelles visent à ne pas utiliser à la fois les produits matrice vecteur avec A et A^T. De plus, l'objet de cette thèse est l'étude de techniques accélératrices des méthodes **FOM**(m) et **GMRES**(m), qui ne font pas appel aux produits matrice vecteur avec la transposée,
- les produits scalaires (xDOT) $\alpha = (v, u) = \bar{u}^T v$,
- les opérations triadiques (xAXPY) $y = y + \alpha x$.

Nous n'avons pas encore examiné les méthodes préconditionnées qui seront vues plus en détail dans le chapitre suivant. Nous verrons alors que le calcul du préconditionnement peut être directement lié au stockage de la matrice A elle-même. Nous avons en outre :

- les calculs permettant de mettre en place le préconditionnement M. Ceux-ci sont inclus dans la phase de prétraitement (*preprocessing*),
- les opérations de préconditionnement appelées également applications qui consistent à calculer le vecteur $v = M^{-1}u$.

Nous pouvons voir dans la figure 1.1, les pourcentages de temps d'exécution des différentes opérations qui sont : l'application du préconditionnement, ici Jacobi par blocs, les produits matrice vecteur, et la méthode elle-même qui est ici **GMRES** et dont le coût principal est dû aux produits scalaires. L'exemple ci-dessous concerne la matrice EX35 de taille 19716 avec 228208 éléments non nuls. Cette matrice est particulièrement creuse, puisque son pourcentage d'éléments non nuls est de $5.8E-2\%$, ce qui est très inférieur à 3%. Ces tests ont été effectués sur 4 processeurs du T3E de l'I.D.R.I.S.[2] pour une taille m de sous-espace de Krylov allant de 5 à 50.

Nous voyons donc que le temps passé dans la méthode **GMRES** augmente avec la taille du sous-espace de Krylov. Ceci est dû principalement au nombre de produits scalaires qui croît de façon quadratique avec la taille m du sous-espace de Krylov. Les communications se font donc plus nombreuses, mais surtout en séquence, comme nous le verrons par la suite. Pour une telle matrice, le temps passé dans la méthode elle-même est donc loin

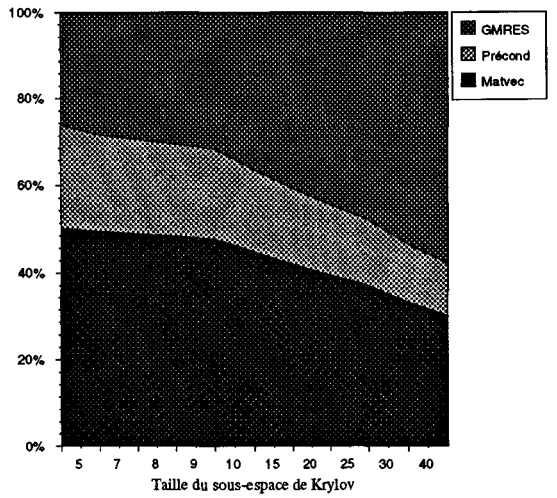


FIG. 1.1 – *Temps passé dans les différentes opérations avec la matrice EX35*

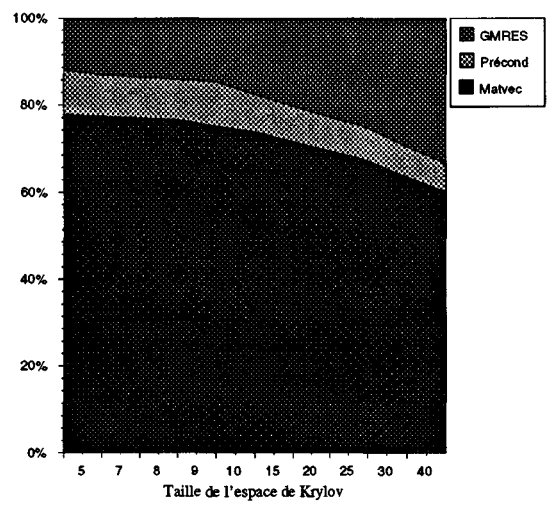


FIG. 1.2 – *Temps passé dans les différentes opérations avec la matrice OLAF*

d'être négligeable par rapport aux produits matrice vecteur. Ceci est dû au fait que la matrice EX35 est très creuse.

Regardons à présent la figure 1.2 représentant les pourcentages de temps des différentes opérations lors de la résolution d'un système linéaire avec la matrice Olaf de dimension 16146 (1015156 éléments non nuls) en utilisant **GMRES** préconditionné à l'aide de la méthode Jacobi par blocs, sur quatre processeurs du T3E et pour des dimensions de redémarrage égales à celles effectuées dans les tests de la matrice EX35. Cette matrice est beaucoup moins creuse que la précédente, puisque son pourcentage d'éléments non nuls est de $3.9E-1\%$. Les résultats sont alors identiques à ceux obtenus précédemment : le temps passé dans **GMRES** lui-même grandit avec la taille du sous-espace de Krylov. Ceci est dû au fait que les calculs et les communications grandissent de façon quadratique avec la dimension du sous-espace, alors que les produits matrice vecteur et les phases de préconditionnement ne croissent que de façon linéaire. Dans cet exemple, les produits matrice vecteur représentent un temps d'exécution prépondérant par rapport aux temps des autres opérations, ce qui n'était pas le cas dans l'exemple précédent. La matrice Olaf est en effet beaucoup moins creuse que la matrice EX35.

Le produit matrice vecteur

Nous allons avant tout étudier le produit matrice vecteur $y = Ax$ qui est l'une des opérations les plus coûteuses dans les méthodes de Krylov et dont les performances sont directement liées au réordonnancement de la matrice et à sa répartition sur les processeurs. Nous ne prenons pas en compte le cas des matrices creuses à structure régulière pour lesquelles des répartitions et des formats adaptés à leur structure peuvent être trouvés. Nous nous plaçons dans le cas des matrices creuses à structure irrégulière et traitons la matrice le plus généralement possible.

Voyons tout de suite quelques exemples de répartition de la matrice A visant à équilibrer la charge. Si l'on considère le partitionnement suivant de la matrice en 4 blocs de colonnes différents sur une machine à 4 processeurs :

$$A = \left(\boxed{A_1} \quad \boxed{A_2} \quad \boxed{A_3} \quad \boxed{A_4} \right).$$

Chacun des blocs A_i affecté au processeur numéro i, est construit de telle sorte que tous les blocs aient un nombre comparable d'éléments non nuls de façon à équilibre la charge de travail. Notons $n \times n_i$ la dimension du bloc A_i. Si nous faisons le choix de répartir chacun des vecteurs de dimension n de la même façon que la matrice A elle-même, à savoir :

$$x^T = \left(\boxed{x_1^T} \quad \boxed{x_2^T} \quad \boxed{x_3^T} \quad \boxed{x_4^T} \right)$$

2. Institut du Développement et des Ressources en Informatique Scientifique, Bâtiment 506 - B.P. 167, 91403 ORSAY CEDEX, FRANCE

le produit matrice vecteur se fait alors comme suit :

- chaque processeur effectue son produit matrice vecteur local $w^i = A_i x_i$ avec w^i un vecteur de dimension n,
- une communication globale (mettant en jeu tout ou partie des processeurs) appelée réduction avec diffusion, permet de diffuser le vecteur $w = \sum_{i=1}^{4} w^i$ sur chacun des processeurs en $log_2(p)$, où p est le nombre de processeurs. Chacun des processeurs ne garde alors que la partie y_i du vecteur w qui lui correspond. Cette communication peut être remplacée par une réduction, laquelle place le vecteur résultat sur un processeur donné, suivie d'une diffusion personnalisée (*scatter*) : à partir du processeur contenant le résultat $w = (y_1^T, \ldots, y_4^T)^T$, cette communication envoie chaque vecteur y_i au processeur numéro i correspondant. Cette seconde solution a le désavantage de mettre en jeu 2 communications globales et surtout d'être deux fois plus coûteuse.

Nous voyons aussi que si nous avions stocké l'intégralité du vecteur x sur chacun des processeurs, nous n'aurions pas réduit le nombre des messages, mais accru le stockage.

Supposons à présent que nous répartissions la matrice A par blocs de lignes consécutives sur chaque processeur de telle sorte que chaque matrice A_i de dimension $n_i \times n$ ait à peu près le même nombre d'éléments non nuls. Le schéma est le suivant :

$$A = \begin{pmatrix} A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline A_4 \end{pmatrix}.$$

Si nous stockons l'ensemble des vecteurs de dimension n sur chaque processeur, le produit matrice vecteur devient :

- un produit matrice vecteur local $w_i = A_i x$ où le vecteur w_i est de dimension n_i,
- une opération de type regroupement (*gather*) permettant de former l'ensemble du vecteur résultat $y = (w_1^T, \ldots, w_4^T)^T$ sur un processeur donné, suivie d'une opération de diffusion permettant de communiquer ce vecteur résultat à l'ensemble des processeurs.

Si nous ne stockons sur chaque processeur que la partie du vecteur x correspondant à la matrice, nous voyons que ce coût de stockage en moins est remplacé par des échanges de messages pour permettre d'avoir sur chaque processeur l'ensemble du vecteur x et pour pouvoir effectuer le produit matrice vecteur local.

Nous voyons alors que découper la matrice horizontalement ou verticalement soulève les mêmes problèmes. Nous pourrions découper la matrice à la fois verticalement et horizontalement avec des dimensions de bloc fixes ou non, en choisissant soit de répartir un bloc par processeur, soit de répartir plusieurs blocs sur chaque processeur de façon

cyclique. Ce choix n'est pas forcément judicieux car il ne tient pas compte du fait que la matrice est creuse et n'a pas de structure régulière. Une solution qui permet de réduire les coûts du produit matrice vecteur mais nécessite une phase de prétraitement (*preprocessing*) est le réordonnancement de la matrice A. Cette technique consiste à appliquer une matrice de permutation Q à droite et P à gauche de A de telle sorte que la matrice

$$\tilde{A} = PAQ \tag{1.28}$$

ait une forme «convenable».

La matrice Q permute les colonnes de A, tandis que la matrice P permute les lignes de A. Nous ne verrons cependant que des permutations symétriques à savoir pour lesquelles $Q = P^T$. L'ensemble des vecteurs utilisés et notamment x et b subissent l'action du réordonnancement. Ils deviennent:

$$x = Px \quad \text{et} \quad b = Pb.$$

Une forme «convenable» serait de pouvoir mettre la matrice sous la forme:

$$A = \begin{pmatrix} A_1 & & & & \\ & A_2 & & & \\ & & A_3 & & \\ & & & A_4 & \\ & & & & A_5 \\ & & & & & A_6 \end{pmatrix}$$

comme une succession de blocs les plus diagonaux possibles. En effet, si nous affectons A_i au processeur i et que nous découpons les vecteurs suivant le même schéma, plus les blocs sont diagonaux, moins chaque processeur aura besoin des valeurs de x contenues sur les autres processeurs. Afin d'obtenir un tel réordonnancement, nous considérons la matrice comme un graphe, appelé graphe d'adjacence, que nous essayons de partitionner en sous-graphes le plus équitablement possible en termes de nombre d'arcs et tels qu'il y ait le moins d'arêtes possibles partant d'un sous-graphe et allant vers un autre sous-graphe: nous essayons d'équilibrer la charge et de réduire au minimum les communications.

Le graphe d'adjacence de la matrice est défini par $G = (V, E)$ où V représente les n inconnues de la matrice et E l'ensemble des arcs de la matrice. E est un sous-ensemble de $V \times V$ défini par:

«Il existe une arête du nœud i vers le nœud j si $a_{i,j} \neq 0$».

On dit que les nœuds i et j sont connectés ou adjacents.

Il existe de nombreuses techniques de réordonnancement comme les réordonnancements multi-niveaux [49, 26, 25].

Une des techniques adoptées par Saad et Malevsky dans la bibliothèque P_SPARSLIB [51, 53] consiste à adapter les techniques de décomposition de domaine au problème de partitionnement de la matrice sur les processeurs. Les méthodes de décomposition de domaine sont apparues initialement pour résoudre des équations aux dérivées partielles. Ces méthodes sont utilisées comme moyen efficace de partitionner le graphe d'adjacence

de la matrice en sous-graphes $V_i \subseteq V$ tels que $\bigcup_i V_i = V$ et pour répartir ces différents sous-graphes sur chaque processeur. Plusieurs sous-graphes peuvent être affectés à un même processeur. Mais nous n'allons considérer ici que le cas où un seul sous-graphe est affecté à un unique processeur.

Supposons qu'un partitionnement judicieux ait été trouvé et qu'après avoir réordonné la matrice A en $\tilde{A}$, celle-ci est partitionnée par lignes. Chaque processeur i contient une matrice $\tilde{A}_i$ de n_i lignes consécutives. Le vecteur x (et les vecteurs en général) est réordonné pour donner le vecteur $\tilde{x}$, lequel est découpé conformément à la matrice $\tilde{A}$.

Si l'on se place du point de vue de la matrice A, les nœuds affectés à chaque processeur sont classés ainsi :

1. les nœuds dits internes : ils correspondent aux nœuds du sous-graphe V_i qui ne sont connectés qu'avec d'autres nœuds de V_i,

2. les nœuds locaux à l'interfaçage, à savoir les nœuds de V_i qui sont connectés avec des nœuds situés sur d'autres processeurs.

Pour effectuer le produit matrice vecteur, chaque processeur aura besoin des nœuds dits externes à l'interfaçage, qui sont les nœuds, appartenant à d'autres processeurs, connectés aux nœuds locaux de l'interfaçage du processeur.

Le vecteur x est partitionné de la même façon que A, c'est-à-dire que ses composantes sur le processeur i correspondent aux nœuds du sous-graphe sur ce même processeur. La partie de x affectée au processeur i est notée $\tilde{x}_i$.

Chaque matrice $\tilde{A}_i$ est alors constituée de deux parties :

- d'une matrice locale $\tilde{A}_{loc,i}$ correspondant aux éléments $\tilde{a}_{i,j}$ pour $i = 1, \ldots, n_i$ et j correspondant à des nœuds internes et locaux à l'interfaçage : c'est une matrice carrée,

- d'une matrice externe $\tilde{A}_{ext,i}$ correspondant aux éléments $\tilde{a}_{i,j}$ pour $i = 1, \ldots, n_i$ et j correspondant à des nœuds externes à l'interfaçage.

Le produit matrice vecteur s'écrit alors :

1. réception des composantes de x correspondant aux nœuds externes renommées $\tilde{x}_{ext}$,

2. le produit matrice vecteur local $\tilde{A}_{loc,i}\tilde{x}_i$,

3. multiplication $\tilde{A}_{ext,i}\tilde{x}_{ext}$ et addition au résultat du produit matrice vecteur local.

Nous voyons que ce produit matrice vecteur ne fait intervenir que le strict minimum en matière de communications, puisqu'il n'implique que celles dont il a exactement besoin. Il permet aussi de faire du recouvrement calculs / communications : le produit local est effectué pendant l'attente du vecteur $\tilde{x}_{ext}$. Le coût de ces communications dépend bien sûr de la qualité du partitionneur de graphe et de l'affectation des sous-graphes sur les processeurs.

Une fois la matrice réordonnée et distribuée, intervient alors la façon dont la matrice va être stockée. Étant creuse, les zéros ne sont pas stockés. Seules les valeurs non nulles

le sont. Il existe de nombreux formats de stockage possibles, dont un grand nombre sont référencés dans le logiciel Sparskit [44]. Bien qu'il existe de nombreux formats, nous ne parlerons que de ceux qui sont les plus courants et que nous utiliserons dans la suite de la thèse : les formats **CSR** et **CSC**. Ces formats représentent la façon dont chaque bloc de la matrice représentée précédemment par A_i est stocké sur le processeur numéro i.

Dans le cas du premier exemple où la matrice est découpée par colonnes, chaque matrice A_i peut être stockée sous le format Compress Sparse Column. Elle pourrait aussi être stockée sous le format Compress Sparse **R**ow que nous verrons ci-dessous. Ce format consiste à mettre bout à bout chaque colonne de la matrice A_i en ayant soin de l'avoir compressée, c'est-à-dire en ayant enlevé les zéros. Ce premier tableau constitue le tableau dénommé TA. Il est de dimension nnz_i le nombre d'éléments non nuls de la matrice A_i. Il faut un deuxième tableau IA de même dimension que TA qui à chaque valeur de TA donne l'indice de ligne auquel correspond cette valeur dans la matrice A_i. Un troisième tableau JA permet de définir complètement le tableau. L'élément $JA(k)$ donne l'indice à partir duquel est rangée la colonne k. Si n_i est le nombre de colonnes de la matrice A_i, alors le tableau JA possède $n_i + 1$ éléments et $JA(n_i + 1) = nnz_i + 1$.

La matrice suivante :

$$A_i = \begin{pmatrix} 1 & 0 & -3 & 0 \\ 0 & -2 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 4 \end{pmatrix}$$

stockée avec le format **CSC** engendre les trois tableaux suivants :

TA	1	−1	2	−2	3	−3	4
IA	1	3	4	2	4	1	4
JA	1	4	6	7	8		

Le format Compress Sparse **R**ow consiste à effectuer la même chose, mais cette fois-ci en mettant chaque ligne bout à bout après avoir enlevé tous les éléments nuls de la matrice. Ceci constitue le tableau TA. Le tableau JA donne l'indice de colonne des éléments de TA par rapport à leur position dans la matrice A_i tandis que la matrice IA pointe sur les éléments non nuls de début de chaque ligne. Pour la même matrice A_i, le format **CSR** devient :

TA	1	−3	−2	−1	2	3	4
JA	1	3	2	1	1	2	4
IA	1	3	4	5	8		

Par la suite nous utiliserons la bibliothèque P_SPARSLIB qui, après avoir réordonné la matrice, la partitionne par blocs de lignes consécutives A_i et la traite comme une matrice locale $A_{loc,i}$ et une matrice externe $A_{ext,i}$. Les vecteurs sont réordonnés et partitionnés de la même façon. Sur chaque processeur, la matrice est alors stockée sous le format **CSR** et le produit matrice vecteur local fait à l'aide de la routine **amux** de Sparskit [44].

Quand nous n'utilisons pas la bibliothèque P_SPARSLIB, aucun réordonnancement n'est fait. Nous appliquons exactement le partionnement montré dans le premier exemple soit un partitionnement de la matrice par blocs de colonnes A_i tel que chaque matrice A_i ait un nombre comparable d'éléments non nuls. Le stockage utilisé pour chaque matrice A_i

est alors le format **CSC**, qui est celui à partir duquel sont lues les matrices. Nous utilisons en effet des matrices de diverses collections et entre autres de la collection Harwell-Boeing [11], pour laquelle existe un format de fichier précis qui tend à se généraliser et où les matrices sont stockées sous le format **CSC**. Nous utilisons alors la routine **atmuxr**, qui multiplie un vecteur par la transposée de la matrice lorsque la matrice elle-même est stockée sous le format **CSR**, ou de façon équivalente, qui multiplie une matrice stockée sous le format **CSC** par un vecteur.

Nous allons voir à présent les autres opérations utilisées dans les méthodes de Krylov.

Le produit scalaire

Bien que le produit matrice vecteur soit l'une des opérations les plus pénalisantes et que, comparativement, le coût d'un produit scalaire ne soit pas élevé, l'ensemble des produits scalaires peut devenir à son tour une opération pénalisante. C'est le cas dans les processus de Hessenberg généralisé et d'Arnoldi (algorithmes 1.1 et 1.2) où les produits scalaires se font en séquence : Av_{j+1} ne peut être orthogonalisé par rapport aux vecteurs u_i que si le vecteur v_{j+1} est calculé.

Dans le processus de Hessenberg, ces produits scalaires sont remplacés par des calculs de composantes de vecteurs. Cette opération est moins coûteuse, certes, mais moins qu'il n'y paraît car elle induit tout de même des communications.

Regardons d'abord comment s'effectue le produit scalaire $\alpha = (v, u)$ du processus de Hessenberg généralisé. Les vecteurs u et v sont répartis de la même façon sur chaque processeur. Soit u_i et v_i l'ensemble des composantes de u et v sur le processeur i. Nous effectuons alors sur chaque processeur i le produit scalaire local $\alpha_i = (u_i, v_i)$. Puis il nous faut sommer tous les α_i. Cette opération se fait au travers d'une seule communication globale effectuant simultanément la réduction permettant de calculer la somme $\alpha = \sum_i \alpha_i$ et sa diffusion. Cette opération peut par exemple être effectuée à l'aide de communiations en *butterfly*, qui induisent un coût en $log_2(p)$, où p est le nombre de processus. Nous avons donc au total, pour chaque passage dans la boucle j, $j+1$ communications globales (calcul de la norme y compris).

Dans le cas du processus de Hessenberg, le produit scalaire est remplacé par le mécanisme suivant :

- recherche du processeur qui possède la composante du vecteur demandée,
- opération de diffusion de la valeur correspondante vers les autres processeurs.

Le produit scalaire local est donc remplacé par une simple affectation. Il n'y a plus de réduction, mais la diffusion demeure, qui s'effectue en $log_2(p)$. Au sortir de la boucle la plus interne, la mise à jour du coefficient $\eta_{j+1} = (v_{j+1})_{j+1}$ nécessite également la recherche du processeur à qui appartient la composante $j+1$ du vecteur v_{j+1}, puis la diffusion de cette composante. Les coûts de communication sont donc les mêmes que pour les processus de Hessenberg généralisé, dans sa forme la plus générale et le processus d'Arnoldi. Néanmoins, l'algorithme ne peut s'employer tel quel, car sinon des divisions par zéro surviennent. Il nécessite d'employer des pivots lors du calcul du coefficient η_{j+1}. Le calcul de ce coefficient dans l'algorithme avec pivot nécessite à chaque itération une

réduction avec diffusion au sortir de la boucle la plus interne (Algorithme 1.10), de même coût que la simple diffusion lorsqu'aucun pivot n'est utilisé. L'algorithme final est donc moins coûteux que le processus de Hessenberg généralisé dans sa forme la plus générale, d'un point de vue nombre d'opérations, mais ce n'est pas vrai pour les communications qui sont exactement de même coût que dans le processus de Hessenberg généralisé. Le calcul des vecteurs Av_j demeure encore intrinsèquement séquentiel.

L'opération triadique

L'opération triadique (xAXPY) $y = y + \alpha\, x$ est une opération qui ne fait pas intervenir de communications. En effet, comme les vecteurs x et y sont partitionnés de la même façon sur les processeurs et puisque les scalaires sont stockés de façon redondante, l'opération suivante s'effectue sur chaque processeur :

$$y_i = y_i + \alpha\, x_i,$$

où y_i et x_i sont les portions des vecteurs y et x locales au processeur i.

Le préconditionnement

Nous ne parlerons pas des opérations permettant de calculer le préconditionneur et le mettant en jeu, car elles dépendent totalement du préconditionneur choisi. Ces opérations sont étudiées dans le chapitre suivant.

1.3.3 Conclusion

Nous venons de voir que la distribution des données jouait un rôle fondamental dans le coût des communications et avons vu quelques techniques existantes qui permettent de réduire le temps d'exécution des méthodes de Krylov sur machines parallèles. Ces techniques sont directement liées à l'implantation des méthodes et plus particulièrement à l'optimisation des opérations les plus utilisées par ces méthodes.

L'objectif de la thèse est de réduire le temps de calcul des méthodes **FOM**(m) et **GMRES**(m) sur machines parallèles, en essayant de supprimer les zones de calcul les moins parallèles de ces méthodes et de diminuer le nombre de leurs opérations les plus coûteuses, en agissant d'un point de vue algorithmique.

Le chapitre suivant énumère de façon non exhaustive les accélérations les plus générales des méthodes de Krylov, puis celles plus particulières associées à la méthode **GMRES**(m), qui est la méthode à laquelle nous nous intéressons dans la thèse.

Chapitre 2

Méthodes d'accélération de la convergence

Sommaire

Dans ce chapitre, nous nous intéressons aux techniques permettant d'améliorer les méthodes vues dans le chapitre précédent. Ces techniques sont générales et peuvent s'appliquer à l'ensemble des méthodes de Krylov. Nous nous attardons ensuite au cas particulier de la méthode **GMRES**.

Les améliorations apportées par ces techniques se situent principalement à deux niveaux :

- l'accélération numérique. Elle consiste à améliorer la méthode d'un point de vue de la convergence et de la robustesse, en, respectivement, diminuant le nombre total d'itérations pour converger ou en faisant en sorte d'éliminer les pannes. Ces

techniques sont les plus courantes : elles peuvent dépendre d'un problème très spécifique, c'est le cas des techniques de lissage (*smoothing*), ayant pour objectif d'éviter le comportement oscillatoire de la norme du résidu des méthodes **PO**, ou bien très générales comme les préconditionnements qui sont vus dans la section suivante et les méthodes par blocs. Nous verrons dans le chapitre 4 une technique d'accélération des méthodes **FOM** et **GMRES** voulant réduire la perte d'informations due aux redémarrages,

- l'amélioration de l'efficacité sur machines parallèles. De façon générale, les méthodes de Krylov n'ont pas été inventées avec la préoccupation d'être parallélisables et leur implantation sur machines parallèles soulève souvent des difficultés comme c'est le cas des produits scalaires du processus de Hessenberg généralisé. Ces techniques tentent de supprimer les zones de calcul les moins parallèles des méthodes de Krylov, en les améliorant d'un point de vue algorithmique. Nous verrons à cet effet dans le chapitre 3 les méthodes **ADOP** accélérant les méthodes **FOM** et **GMRES** sur machines parallèles.

Nous commencons par les techniques de préconditionnement.

2.1 Préconditionnement

Les méthodes de préconditionnement sont parmi les plus répandues et les plus développées. Elles permettent, lorsque le préconditionneur est adapté, non seulement de réduire le temps de calcul de la solution mais aussi d'accroître l'efficacité et la robustesse des méthodes de Krylov.

2.1.1 Principe

Les méthodes de préconditionnement consistent à résoudre un système linéaire plus facile que celui initial, de même solution et tel que : si nb_p et nb sont le nombre total d'itérations pour résoudre le problème respectivement préconditionné et non préconditionné, nb_p doit être inférieur à nb. De plus le coût total du calcul de la solution avec le système préconditionné, comprenant le calcul du préconditionnement lui-même dans la phase de prétraitement, doit être inférieur à celui du système préconditionné. Ce coût comprend à la fois, le temps d'exécution (converger en un nombre d'itérations moindre mais deux fois plus coûteuses n'est pas nécessairement un gain) et l'allocation des ressources mesurée en terme de complexité mémoire. Dans le cas d'un préconditionneur parallèle, ces contraintes doivent être respectées sur architectures parallèles.

Il existe trois sortes de préconditionnement :

- le préconditionnement à gauche consiste à résoudre le problème suivant :

$$M^{-1}Ax = M^{-1}b$$

où la matrice M est appelée le préconditionneur et est choisie de telle sorte qu'elle soit proche de A dans un certain sens et en même temps facile à appliquer,

- le préconditionnement à droite qui résout le problème suivant :

$$AM^{-1}t = b \text{ et } x = M^{-1}t,$$

- si le préconditionneur se met sous la forme $M = M_1M_2$, il est alors possible de considérer le préconditionnement dit à droite et à gauche qui résout :

$$M_1^{-1}AM_2^{-1}t = M_1^{-1}b \text{ avec } x = M_2^{-1}t.$$

Lorsqu'elles sont utilisées avec un préconditionnement à gauche, les méthodes qui ont été vues au chapitre précédent restent valables, sachant que pour chacune d'entre-elles, les modifications suivantes doivent être faites :

- la matrice du système est à présent $M^{-1}A$ et le second membre $M^{-1}b$,

- le résidu considéré est $\tilde{r}_0 = M^{-1}r_0$ où r_0 est le résidu initial du système non préconditionné. Ceci est particulièrement important pour les critères d'arrêt. En effet, si pour le système préconditionné, nous voulons utiliser le même critère d'arrêt que pour le système non préconditionné, ce qui n'est pas forcément raisonnable, il faut alors former explicitement le vecteur $r_m = M\tilde{r}_m$, car, dans l'algorithme de la méthode préconditionnée à gauche, seul le résidu $\tilde{r}_m$ est disponible.

L'espace de Krylov construit est $\mathrm{K}_m(M^{-1}A, M^{-1}r_0)$ où r_0 est le résidu initial du système non préconditionné. Nous avons alors $\mathrm{L}_m = \mathrm{K}_m(M^{-1}A, M^{-1}r_0)$, dans le cas d'une projection **PO** et $\mathrm{L}_m = M^{-1}A\mathrm{K}_m(M^{-1}A, M^{-1}r_0)$ dans le cas d'une projection **PM**. Le processus de Hessenberg généralisé s'écrit :

Algorithme 2.1 $[V_{m+1}, \bar{H}_m]$ = *Hessenberg-généralisé-précond-gauche*(A, M, r_0, m, U_m)

1. *Initialisation :*

   (a) *Choix de* $\beta \neq 0$ *coefficient normalisateur de* $\tilde{r}_0 = M^{-1}r_0$

   (b) *Calcul de* $v_1 = \tilde{r}_0/\beta$ *et* $\eta_1 = (v_1, u_1)$

2. *Boucle principale*
   Pour $j = 1, \ldots, m$ *Faire*

   (a) $w = M^{-1}Av_j$

   (b) *Pour* $i = 1, \ldots, j$ *Faire*
   $h_{i,j} = (w, u_i)/\eta_i$
   $w = w - h_{i,j}v_i$
   Fin Pour

   (c) *Choix de* $h_{j+1,j}$ *facteur normalisateur de* v_{j+1}

   (d) $v_{j+1} = w/h_{j+1,j}$ *et* $\eta_{j+1} = (v_{j+1}, u_{j+1})$

   Fin Pour

3. Définir $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $\bar{H}_m = \{h_{i,j}\}$

Dans le cas de la biorthogonalisation de Lanczos (algorithme 1.11), les vecteurs u_i appartiennent au sous-espace de Krylov $\mathrm{K}_m((M^{-1}A)^T, u_1)$ avec u_1 tel que $(\tilde{r}_0, u_1) = 1$.

Une fois les matrices $\bar{H}_m$ et V_{m+1} construites, les méthodes restent inchangées : l'équation (1.10) est résolue dans le cas d'une méthode **PO**, tandis que l'équation (1.13) est résolue pour les méthodes **PM**. La solution approchée est, quant à elle, mise à jour à l'aide de l'équation $x_m = x_0 + V_m z_m$ et le résidu correspondant avec $r_m = r_0 - M^{-1}AV_m z_m$.

Considérons à présent le préconditionnement à droite $AM^{-1}t = b$. Si $t_0 = Mx_0$, nous avons alors :

$$b - AM^{-1}t_0 = b - Ax_0 = r_0.$$

Le résidu du système est le même que celui de la méthode non préconditionnée. L'espace de Krylov quant à lui est $\mathrm{K}_m(AM^{-1}, r_0)$ et le processus de Hessenberg généralisé s'écrit :

Algorithme 2.2 $[V_{m+1}, \bar{H}_m]$ = *Hessenberg-généralisé-précond-droite(A,M,*r_0*,m,*U_m*)*

1. Initialisation :

(a) Choix de $\beta \neq 0$ *coefficient normalisateur de* r_0

(b) Calcul de $v_1 = r_0/\beta$ *et* $\eta_1 = (v_1, u_1)$

2. Boucle principale
Pour $j = 1, \ldots, m$ *Faire*

(a) $w = AM^{-1}v_j$

(b) Pour $i = 1, \ldots, j$ *Faire*
$h_{i,j} = (w, u_i)/\eta_i$
$w = w - h_{i,j}v_i$
Fin Pour

(c) Choix de $h_{j+1,j}$ *facteur normalisateur de* v_{j+1}

(d) $v_{j+1} = w/h_{j+1,j}$ *et* $\eta_{j+1} = (v_{j+1}, u_{j+1})$

Fin Pour

3. Définir $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $\bar{H}_m = \{h_{i,j}\}$

Lors du processus de biorthogonalisation de Lanczos (algorithme 1.11), les vecteurs u_i forment une base du sous-espace de Krylov $\mathrm{K}_m((AM^{-1})^T, r_0)$.

Les équations (1.10) et (1.13) sont ensuite résolues dans les cas respectifs des méthodes **PO** et **PM**. Puis t_m, solution approchée du système $AM^{-1}t = b$, est mis à jour ainsi :

$$t_m = t_0 + V_m z_m \tag{2.1}$$

et

$$x_m = x_0 + M^{-1}V_m z_m. \tag{2.2}$$

Le résidu étant le même pour les méthodes préconditionnées à droite et non préconditionnées et comme x_m s'exprime indépendamment de t_m, il est inutile d'utiliser t_m. Cela permet de réduire l'espace mémoire et les calculs. La matrice M^{-1} apparaît alors seulement dans le processus de Hessenberg généralisé et dans le calcul de x_m.

Le préconditionnement à droite et à gauche allie les deux préconditionnements vus précédemment. Il prend en compte les modifications appliquées et aux préconditionnements à gauche et aux préconditionnements à droite.

Dans les paragraphes suivants, nous énumérons différents préconditionnements dits algébriques, car ils sont construits à partir des propriétés algébriques de la matrice et ne sont pas liés au problème dont elle est issue. Cette énumération n'est pas exhaustive. Notamment, nous ne parlerons pas des préconditionneurs de type *approximate inverse* ni de ceux dérivés des méthodes de décomposition de domaine.

Le choix d'un bon préconditionneur est difficile, car M doit être le plus proche possible de A, mais avec les contraintes supplémentaires suivantes :

- le calcul du préconditionneur lui-même inclus dans la phase de prétraitement (*preprocessing*) ne doit pas être de coût excessif, et éventuellement parallélisable,

- le calcul $M^{-1}v$ doit être de coût relativement faible. Ce coût peut aussi être mesuré en terme de parallélisme.

Il y a donc un compromis à trouver entre la qualité du préconditionneur (diminution réelle du nombre d'itérations pour converger) et les coûts induits pour le calculer mais aussi pour l'appliquer (calcul de $M^{-1}v$).

2.1.2 Préconditionnement dérivé des méthodes stationnaires

Dans ce paragraphe, nous énumérons quelques préconditionnements qui n'ont pas été créés dans le souci que M^{-1} approche l'inverse de la matrice A et qui sont dérivés des méthodes itératives stationnaires [49].

Celles-ci sont des méthodes de point fixe qui à chaque itération calculent le nouvel itéré

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b \tag{2.3}$$

avec

$$A = M - N,$$

avec M choisie non singulière et facilement inversible. Nous réécrivons la matrice A sous la forme $A = D - E - F$ où D, $-E$ et $-F$ sont respectivement, la diagonale, la partie triangulaire strictement inférieure et la partie triangulaire strictement supérieure de A. Nous avons alors ci-dessous les différentes valeurs de M correspondant aux différentes

méthodes stationnaires :

Nom de la méthode	matrice M
Jacobi	D
Gauss-Seidel	$D - E$
Gauss-Seidel Symétrique	$(D - E)D^{-1}(D - F)$
Successive Over Relaxation (SOR)	$D - \omega E$
Symmetric Successive Over Relaxation (SSOR)	$(D - \omega E)D^{-1}(D - \omega F)$

avec ω un paramètre de relaxation compris entre 0 et 2.

L'algorithme décrit par l'equation (2.3) revient à vouloir résoudre le système suivant :

$$(I - M^{-1}N)x = M^{-1}b \tag{2.4}$$

soit encore

$$M^{-1}Ax = M^{-1}b. \tag{2.5}$$

L'idée des préconditionnements stationnaires est de prendre les matrices M égales aux différentes valeurs indiquées dans le tableau ci-dessus. Nous voyons alors que le préconditionnement de Jacobi est équivalent au préconditionnement le plus simple et le plus connu, à savoir le préconditionnement diagonal.

Le principal atout de ces préconditionneurs est qu'ils ne nécessitent pas de phase de prétraitement (*preprocessing*) ni de stockage supplémentaire. Néanmoins, ils ne sont pas très performants en terme de réduction du nombre total d'itérations et, excepté pour le préconditionnement de Jacobi, ne sont pas très parallélisables, puisqu'ils nécessitent les résolutions de systèmes de la forme $Lv = y$ et $Uv = y$ où v est l'inconnue recherchée, L une matrice triangulaire inférieure et U une matrice triangulaire supérieure. Néanmoins, des versions par blocs ont été développées qui, à l'aide d'un réordonnancement de la matrice, permettent un plus grand parallélisme. Ces préconditionnements sont détaillés dans le paragraphe suivant.

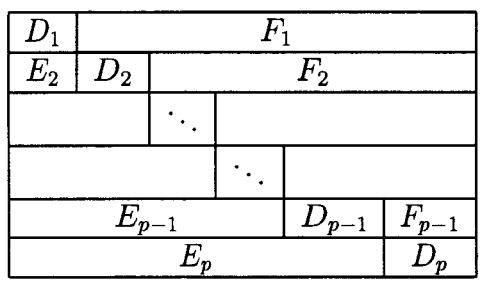
2.1.3 Préconditionnement par blocs

Les méthodes décrites ci-dessus mises sous forme par blocs sont plus efficaces que leur version par point et permettent un plus grand parallélisme, excepté pour la méthode Jacobi, qui est déjà très parallèle.

La méthode Jacobi par blocs consiste à considérer la matrice D du paragraphe précédent comme une matrice diagonale par blocs, les blocs pouvant se recouvrir ou non. Ces matrices blocs n'ont pas de spécificité particulière et surtout ne sont pas nécessairement diagonales. Le calcul de $M^{-1}v$ nécessite d'inverser en parallèle les différents blocs de la diagonale, puis d'adopter une règle soit de correction soit de poids pour les composantes de v correspondant aux recouvrements. En général, le fait de prendre des blocs qui se recouvrent permet de gagner quant à la qualité du préconditionneur et de réduire le nombre total d'itérations. Cependant le recouvrement des blocs fait que les composantes de v correspondant à ces blocs recouverts ne sont pas calculées de la même façon pour chaque v. Le préconditionneur varie au sein d'une même itération. Ceci ne peut se faire qu'avec

des méthodes de Krylov qui autorisent des préconditionneurs flexibles. C'est le cas de la méthode **Flexible GMRES** qui sera étudiée un peu plus loin.

Pour les préconditionnements stationnaires par blocs autres que Jacobi, la matrice D est une matrice diagonale par blocs, où chaque bloc est quelconque. Un réordonnancement judicieux de la matrice A est celui pour lequel la matrice D est une matrice diagonale par blocs de blocs diagonaux. Un tel réordonnancement est possible à l'aide des techniques de multi-coloriage [49]. Les matrices E et F sont respectivement la partie triangulaire strictement inférieure de A en ayant ôté les blocs diagonaux et la partie triangulaire strictement supérieure de A sans les blocs diagonaux. Les calculs $(D-E)^{-1}v$ et $(D-F)^{-1}v$ sont alors plus parallèles. Si p est le nombre de blocs diagonaux, la matrice peut s'exprimer ainsi :



Le calcul de $(D-E)^{-1}v$ devient alors, si nous décomposons $v = (v_1, \ldots, v_p)^T$ de la même façon que la matrice D et écrasons le vecteur v par le résultat $(D-E)^{-1}v$:

Algorithme 2.3 *Résolution de* $(D-E)^{-1}v$

1. *Initialisation :* $v_1 = D_1^{-1}v_1$
2. *Pour* $i = 2, \ldots, p$ *Faire*
   $v_i = D_i^{-1}v_i + E_i(v_1^T, \ldots, v_{i-1}^T)^T$
   Fin Pour

L'algorithme pour résoudre $(D-F)^{-1}v$ s'écrit :

Algorithme 2.4 *Résolution de* $(D-F)^{-1}v$

1. *Initialisation :* $v_p = D_p^{-1}v_p$
2. *Pour* $i = p-1, \ldots, 1$ *Faire*
   $v_i = D_i^{-1}v_i + F_i(v_{i+1}^T, \ldots, v_p^T)^T$
   Fin Pour

Les méthodes Gauss-Seidel, Gauss-Seidel Symétrique, SOR et SSOR par blocs ne nécessitent donc que des produits matrice vecteur et des produits de l'inverse d'une matrice par un vecteur. Ces opérations sont plus intéressantes que celles de la version par point de ces méthodes, car elles mettent en jeu des produits matrice vecteur, là où l'algorithme initial par point (lorsque D est égal à la matrice diagonale de A) n'utilise que des produits scalaires. De plus certains produits matrice vecteur peuvent être calculés en parallèle, notamment en décomposant les produits $F_i(v_{i+1}^T, \ldots, v_p^T)^T$ en $\sum_{j=i+1}^{p} F_{i,j}v_j$ où la matrice F_i est découpée selon v. Ainsi, une fois le vecteur v_p calculé, tous les produits matrice vecteur $F_{i,p}v_p$ peuvent être effectués en parallèle pour $i = 1, \ldots, p-1$. Le parallélisme reste néanmoins limité par la boucle principale i qui est séquentielle.

2.1.4 Préconditionnement de type ILU

Nous abordons à présent une classe de préconditionnements qui en général sont de bien meilleure qualité numérique que les préconditionnements précédents, mais intrinsèquement séquentiel. Ce sont des préconditionnements dérivés des méthodes directes et plus particulièrement de la méthode de Gauss.

La méthode de Gauss transforme la matrice A en un produit de deux matrices, L triangulaire inférieure à diagonale unitaire et U triangulaire supérieure, telles que $A = LU$. La résolution du système linéaire $Ax = b$ est alors immédiate. Nous présentons dans l'algorithme suivant la factorisation de Gauss ikj qui permet de calculer les coefficients des matrices L et U. Les coefficients diagonaux de la matrice L étant égaux à 1, ne sont pas mis à jour par cet algorithme et par les algorithmes de type **ILU** suivants.

Algorithme 2.5 $[L, U]$ = *Factorisation-Gauss-ikj(A)*
Pour $i = 1, \ldots, n$ *Faire :*

1. $u_{i,:} = a_{i,:}$
2. *Pour* $k = 1, \ldots, i-1$ *Faire :*
   (a) $l_{i,k} = u_{i,k}/u_{k,k}$
   (b) *Pour* $j = k+1, \ldots, n$ *Faire :*
   $u_{i,j} = u_{i,j} - l_{i,k} * u_{k,j}$
   Fin Pour

   Fin Pour

Fin Pour

De façon générale, une telle factorisation entraîne un remplissage non négligeable des matrices L et U : alors que la matrice A est creuse, les matrices L et U sont en général plus denses. Nous disons qu'il y a remplissage lorsque pour un élément $a_{i,j}$ nul de la matrice A correspond un élément $l_{i,j}$ de L ou $u_{i,j}$ de U qui est non nul. Le principe des préconditionnements **I**ncomplete **LU** (**ILU**) est d'effectuer cette factorisation partiellement, en interdisant certains remplissages des matrices L et U, c'est-à-dire en imposant que certains de leurs éléments soient nuls. Nous définissons alors une structure (*pattern*) P des matrices L et U, sur lequel nous interdisons le remplissage de ces matrices : P fixe les couples (i, j) de L et U qui doivent être égaux à zéro. Nous effectuons alors une factorisation LU incomplète de A où les coefficients $l_{i,j}$ de L vérifient :

$$l_{i,j} = 0 \text{ si } j > i \text{ ou } (i,j) \in P.$$

De même pour U, nous avons :

$$u_{i,j} = 0 \text{ si } i > j \text{ ou } (i,j) \in P$$

avec $u_{i,j}$ les coefficients de la matrice U. Nous avons en outre la condition suivante qui est satisfaite :

$$m_{i,j} = a_{i,j} \text{ pour } (i,j) \notin P \tag{2.6}$$

avec $m_{i,j}$ les coefficients de la matrice $M = LU$.

La factorisation de Gauss est légèrement modifiée et devient :

Algorithme 2.6 $[L,U]$=*Factorisation-ILU(A,P)*
Pour $i = 1,\ldots,n$ *Faire :*

1. $u_{i,:} = a_{i,:}$
2. *Pour* $k = 1,\ldots,i-1$ *et si* $(i,k) \notin P$ *Faire :*
   (a) $l_{i,k} = u_{i,k}/u_{k,k}$
   (b) *Pour* $j = k+1,\ldots,n$ *et si* $(i,j) \notin P$ *Faire :*
   $u_{i,j} = u_{i,j} - l_{i,k} * u_{k,j}$
   Fin Pour

   Fin Pour

Fin Pour

La factorisation la plus simple est celle pour laquelle L et U ont exactement la même structure (*pattern*) que A :

$$P = \{(i,j) \ / \ a_{i,j} = 0\}.$$

Une telle factorisation s'appelle la factorisation **ILU** et offre un avantage important, qui est de n'autoriser aucun remplissage par rapport à la structure de la matrice A. Les matrices L et U peuvent donc être stockées sous le même schéma que A. Ce préconditionnement fonctionne relativement bien pour nombre de matrices, mais montre une qualité moindre pour les matrices issues de problèmes aux dérivées partielles (particulièrement les problèmes elliptiques [9]). Une légère modification consiste à transformer la condition (2.6) en imposant à la place :

$$m_{i,j} = a_{i,j} \text{ si } (i \neq j \text{ et } (i,j) \notin P) \text{ et } \sum_{j=1}^{n} m_{i,j} = \sum_{j=1}^{n} a_{i,j}.$$

L'algorithme qui en découle s'appelle Modified **ILU** et est décrit ci-dessous :

Algorithme 2.7 $[L,U]$=*Factorisation-MILU(A,P)*
Pour $i = 1,\ldots,n$ *Faire :*

1. $u_{i,:} = a_{i,:}$
2. *Pour* $k = 1,\ldots,i-1$ *et si* $(i,k) \notin P$ *Faire :*
   (a) $l_{i,k} = u_{i,k}/u_{k,k}$
   (b) *Pour* $j = k+1,\ldots,n$ *Faire :*
   Si $(i,j) \notin P$
   $u_{i,j} = u_{i,j} - l_{i,k} * u_{k,j}$
   sinon
   $u_{i,i} = u_{i,i} - l_{i,k} * u_{k,j}$
   Fin Si
   Fin Pour

Fin Pour

Fin Pour

En vue d'augmenter à la fois la fiabilité de ce préconditionnement et le parallélisme, d'autres variantes ont été adoptées [9] :

- la factorisation **ILU** relaxée se situant entre **ILU** et **MILU**, à la façon dont **SOR** se situe entre Jacobi et Gauss-Seidel,
- la factorisation **ILU** de la matrice $A + \alpha I_n$,
- des factorisations **ILU** et **MILU** par blocs,
- davantage de remplissage,
- des techniques de réordonnancement et de multi-niveaux visant à augmenter le parallélisme.

Nous nous intéressons à présent à ces deux dernières variantes. La première consiste à autoriser davantage de remplissage [49]. La méthode **ILU** réécrite sous le nom **ILU(0)** est la méthode qui n'autorise aucun remplissage soit encore du remplissage de niveau zéro. Soient L_0 et U_0 les matrices obtenues après cette factorisation. Nous définissons les matrices L_1 et U_1 de la factorisation LU Incomplète de niveau 1, comme les matrices issues de la factorisation LU incomplète de A où la structure (*pattern*) P est choisi comme l'ensemble des éléments nuls de $M_0 = L_0 U_0$. Cette définition permet de construire les matrices L_p et U_p récursivement. On appelle cette factorisation la factorisation **ILU** d'ordre p notée **ILU(p)**. Pour ce faire, nous définissons ainsi le niveau de remplissage de chaque élément à l'initialisation :

Définition 2.1 *Les niveaux de remplissage des éléments de la matrice creuse A sont définis comme suit :*

$$\text{level}(a_{i,j}) = \begin{cases} 0 \text{ si } a_{i,j} \neq 0 \text{ ou si } i = j \\ \infty \text{ sinon} \end{cases}$$

puis récursivement d'après la mise à jour :

$$u_{i,j} = u_{i,j} - l_{i,k} * u_{k,j}$$

comme

$$\text{level}(u_{i,j}) = \min\{\text{level}(u_{i,j}), \text{level}(u_{i,k}) + \text{level}(l_{k,j}) + 1\}. \tag{2.7}$$

Ceci provient du fait que le niveau d'un élément devrait être en rapport avec son ordre de grandeur. Plus le niveau d'un élément est élevé, plus son ordre de grandeur est supposé faible. La méthode **ILU(p)** interdit le remplissage des éléments de niveau supérieur à p. L'algorithme correspondant est le suivant :

Algorithme 2.8 *$[L, U]$=Factorisation-ILU(p)(A)*
Pour tous les éléments $a_{i,j}$ non nuls

Faire $\text{level}(a_{i,j}) = 0$ *sinon* $\text{level}(a_{i,j}) = \infty$
Pour $i = 1, \ldots, n$ *Faire :*

1. *Pour* $j = i, \ldots, n$ *Faire :*
   $u_{i,j} = a_{i,j}$
   $\text{level}(u_{i,j}) = \text{level}(a_{i,j})$
   Fin Pour

2. *Pour* $k = 1, \ldots, i-1$ *et si* $u_{i,k} \neq 0$ *Faire :*

   (a) $l_{i,k} = u_{i,k}/u_{k,k}$

   (b) $\text{level}(l_{i,k}) = \text{level}(u_{i,k})$

   (c) *Pour* $j = k+1, \ldots, n$ *Faire :*
   $u_{i,j} = u_{i,j} - l_{i,k} * u_{k,j}$
   Mettre à jour le niveau de $u_{i,j}$ *à l'aide de l'équation (2.7)*
   Si $\text{level}(u_{i,j}) > p$ *alors*
   $u_{i,j} = 0$
   Fin Si
   Fin Pour

   Fin Pour

Fin Pour

Les principaux inconvénients de cette méthode sont que :

- le remplissage n'est pas prévisible,
- le coût de mise à jour des niveaux peut être élevé,
- le niveau n'est pas toujours un bon indicateur de l'ordre de grandeur d'un élément.

Saad a introduit une méthode tenant compte de l'ordre de grandeur des éléments en introduisant des règles numériques : il s'agit de la factorisation **ILUT** [48] avec **T** pour **T**reshold (seuil en français). Ces règles numériques peuvent également limiter le nombre de remplissages supplémentaires par ligne de U et de L. L'algorithme est le suivant :

Algorithme 2.9 $[L, U]$=*Factorisation-ILUT(A)*
Pour $i = 1, \ldots, n$ *Faire :*

1. $w = a_{i,:}$

2. *Pour* $k = 1, \ldots, i-1$ *et là où* $w_k \neq 0$ *Faire :*

   (a) $w_k = w_k/a_{k,k}$

   (b) *Appliquer la règle (1) à* w_k

   (c) *Si* $w_k \neq 0$ *Alors :*
   $w = w - w_k * u_{k,:}$
   Fin Si

Fin Pour

3. *Appliquer la règle (2) à* w
4. $l_{i,j} = w_j$ *pour* $j = 1, \ldots, i-1$
5. $u_{i,j} = w_j$ *pour* $j = i, \ldots, n$

Fin Pour

Les règles de suppression (1) et (2) adoptées par Saad dans l'algorithme **ILUT** renommé **ILUT(p, τ)** sont les suivantes :

- la règle (1) annule l'élément w_k si $|w_k| \leq \tau \|a_{i,:}\|_2$,
- la règle (2) s'applique à l'ensemble de la ligne w. Elle applique tout d'abord la règle précédente à tous les éléments de w, puis ne garde que les p plus grands éléments en valeur absolue dans la partie de L. De même, elle ne conserve que les p plus grands éléments restants dans U ainsi que l'élément diagonal de U.

La qualité de ce nouveau préconditionneur est réelle et permet de diminuer notablement le nombre d'itérations pour converger. Cependant, ce préconditionneur n'est absolument pas parallèle. En effet, utiliser une technique de multi-coloriage ou de *wavefront* [49] pour réordonnancer la matrice permet d'obtenir un préconditionneur **ILU(0)** parallèle. En ce qui concerne le préconditionneur **ILUT**, il introduit des remplissages qui ne sont tout d'abord pas prévisibles et qui, de plus, détruisent la structure «parallèle» obtenue grâce au réordonnancement.

Pour y remédier, Saad propose la méthode **ILUM** [46], à savoir **ILU** avec Multi-élimination, qui permet d'allier à la fois qualité numérique et parallélisme. Cette méthode est dérivée des solveurs directs parallèles qui, à partir de techniques et d'heuristiques de partitionnement du graphe d'adjacence de la matrice fait décroître la dimension du système à résoudre. Ceci se fait par le calcul récursif d'ensembles indépendants. Si $G = (V, E)$ est le graphe d'adjacence de la matrice, on dit que $S \in V$ est un ensemble indépendant s'il vérifie :

$$\text{Si } x \in S \text{ alors } \{(x, y) \in E \text{ ou } (y, x) \in E\} \rightarrow y \notin S.$$

Le principe est alors le suivant :
notons $A_0 = A$ la matrice du système et supposons que nous ayons calculé la matrice A_j à l'itération j. Après avoir trouvé un ensemble indépendant S_j de A_j et calculé la matrice de permutation correspondante P_j, nous pouvons réordonner la matrice A_j et la mettre sous la forme :

$$P_j A_j P_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix}$$

avec D_j une matrice diagonale. En éliminant les inconnues de l'ensemble indépendant S_j (inconnues correspondant à D_j), la matrice du système résultant est alors la suivante :

$$A_{j+1} = C_j - E_j D_j^{-1} F_j$$

ce qui correspond aussi à avoir effectué la factorisation **LU** suivante :

$$P_j A_j P_j^T = \begin{pmatrix} I & 0 \\ E_j D_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ 0 & A_{j+1} \end{pmatrix}. \tag{2.8}$$

Ce procédé peut s'effectuer jusqu'à obtention d'une matrice A_p qui soit de dimension minimale, ou bien qui soit si dense que nous ne puissions plus trouver d'ensemble indépendant suffisamment grand. La résolution du système linéaire se fait ensuite par la résolution de deux systèmes triangulaires, la résolution du système linéaire avec la matrice A_p, qui constitue la difficulté principale, l'inversion des matrices D_k et les produits matrice vecteur avec les matrices $E_k D_k^{-1}$ et F_k pour $k = 0, \ldots, p-1$.

Au fur et à mesure du procédé, le remplissage devient de plus en plus important et la matrice est plus coûteuse à calculer. Le principe de la méthode **ILUM** consiste à calculer une factorisation **LU Incomplète** de $P_j A_j P_j^T$ en interdisant certains remplissages. Ainsi la matrice A_{j+1} s'écrit :

$$A_{j+1} = (C_j - E_j D_j^{-1} F_j) - R_j$$

avec R_j la matrice représentant les éléments qui sont éliminés. Les règles exprimées pour la méthode **ILUT** peuvent s'appliquer ici aussi.

À chaque étape j, nous devons garder les matrices D_j, E_j, F_j et P_j qui nous permettent de conserver la factorisation. Une partie de ces éléments sont stockés dans la matrice B_{j+1} qui vaut :

$$B_{j+1} = \begin{pmatrix} D_j & F_j \\ E_j D_j^{-1} & 0 \end{pmatrix}.$$

La matrice L finale s'écrit alors :

$$L = \begin{pmatrix} I & & & \\ E_0 D_0^{-1} & I & & \\ & E_1 D_1^{-1} & \ddots & \\ & & \ddots & I \\ & & & E_{p-1} D_{p-1}^{-1} \quad I \end{pmatrix}. \tag{2.9}$$

De la même façon, on obtient la matrice U :

$$U = \begin{pmatrix} D_0 & F_0 & & \\ & D_1 & F_1 & \\ & & \ddots & \ddots \\ & & & D_j \quad F_{p-1} \\ & & & \quad A_p \end{pmatrix}. \tag{2.10}$$

L'algorithme final est le suivant :

Algorithme 2.10 $[L, U]$ = *Factorisation-ILUM(A,p)*

- *Initialisation :* $A_0 = A$

- *Pour* $i = 0, \ldots, p-1$ *Faire :*

  1. *calculer* P_i, *la matrice de permutation liée à un ensemble indépendant de* A_i,
  2. *appliquer* P_i *à* A_i,
  3. *appliquer* P_i *à* $B_1, \ldots, B_i$,
  4. *appliquer* P_i *à* $P_0, \ldots, P_{i-1}$,
  5. *calculer les matrices* A_{i+1} *et* B_{i+1}.

  Fin Pour

- *Mise à jour de* L *et* U *grâce aux relations (2.9) et (2.10)*

L'application de ce nouveau préconditionnement nécessite de résoudre $(LU)^{-1}\tilde{x} = U^{-1}L^{-1}\tilde{x}$, où $\tilde{x}$ est le vecteur x ayant subi la permutation $P_{p-1} \ldots P_0$. Notons $\tilde{x} = \tilde{x}_0$ et :

$$\tilde{x}_j = \begin{pmatrix} \tilde{y}_j \\ \tilde{x}_{j+1} \end{pmatrix}$$

pour $j = 0, \ldots, p-1$ de telle sorte que $\tilde{x}_j$ soit partitionné conformément à $P_j A_j P_j^T$ (2.8).

L'algorithme de résolution du système $(LU)^{-1}\tilde{x}$ est le suivant :

Algorithme 2.11 $[x]$ = *Résolution-ILUM(L,U,*$\tilde{x}$*,p)*

- *Pour* $j = 0, \ldots, p-1$ *Faire :*
  $\tilde{x}_{j+1} = \tilde{x}_{j+1} - E_j D_j^{-1} \tilde{y}_j$
  Fin Pour

- *Résoudre* $A_p \tilde{x}_p = \tilde{x}_p$

- *Pour* $j = p-1, \ldots, 0$ *Faire :*
  $\tilde{y}_j = D_j^{-1}(\tilde{y}_j - F_j \tilde{x}_{j+1})$
  Fin Pour

- $x = P_0^T \ldots P_{p-1}^T \tilde{x}$

Parmi les préconditionnements **ILU**, **ILUM** est le plus parallèle. Il ne l'est cependant pas autant que les préconditionnements polynomiaux, que nous étudions dans le paragraphe suivant.

2.1.5 Préconditionnement polynomial

Les préconditionnements polynomiaux sont tels que la matrice de préconditionnement M satisfait $M^{-1} = p_k(A)$ où p_k est un polynôme quelconque de degré k.

La phase de prétraitement (*preprocessing*) qui consistait auparavant à calculer la matrice M est remplacée par le calcul de la matrice M^{-1}. Or nous ne la formons pas explicitement, car son calcul est beaucoup trop coûteux et parce que son remplissage augmente très rapidement avec le degré k du polynôme. Nous calculons seulement les coefficients du

polynôme p_k dans la base $(x^i)_{0\leq i\leq k}$ ou toute autre base, ou bien les paramètres permettant de calculer ces coefficients récursivement. Les coûts de stockage et de calcul en sont allégés. Si p_k s'écrit

$$p_k(x) = \sum_{i=0}^{k} \alpha_i x^i,$$

le calcul $p_k(A)Av = Ap_k(A)v$ revient alors à effectuer $k+1$ produits matrice vecteur à l'aide du schéma de Hörner suivant :

$$\begin{array}{l} w = \alpha_k v \\ \text{Pour } \ i = 1, \ldots, k \ \text{ Faire} \\ \quad w = Aw + \alpha_{k-i} v \\ \text{Fin Pour} \\ w = Aw. \end{array} \tag{2.11}$$

De tous les préconditionnements vus jusqu'à présent, ce préconditionnement est certainement le plus parallèle.

Pour que l'utilisation d'un tel préconditionnement soit justifiée, il faut que le nombre d'itérations pour converger du système préconditionné soit inférieur à celui du système non préconditionné et que leur coût total soit moindre, ce qui revient à dire que le nombre de produits matrice vecteur du système préconditionné pour converger doit être inférieur à celui du système non préconditionné. À nombre de produits matrice vecteur égal, il peut être quand même intéressant d'utiliser un préconditionnement polynomial pour les méthodes utilisant explicitement le processus de Hessenberg généralisé. En effet, pour ces méthodes, les produits matrice vecteur sont combinés avec des produits scalaires qui se font en séquence et constituent une perte de temps sur machines parallèles (voir chapitre 1). Lors de l'application du préconditionnement, ces produits scalaires n'apparaissent pas. Il devient donc avantageux d'augmenter le degré du préconditionnement polynomial, si le nombre total de produits matrice vecteur reste le même.

Le premier préconditionnement polynomial que nous verrons, le plus classique, mais souvent le moins performant, est issu de la série de Neumann, que nous prenons tronquée. Nous avons :

$$A^{-1} = (I - (I - A))^{-1}. \tag{2.12}$$

Lorsque le rayon spectral de la matrice $I - A$ est strictement inférieur à 1 ($\rho(I - A) < 1$), $(I - (I - A))^{-1}$ est développable en série entière, soit :

$$A^{-1} = \sum_{i=0}^{\infty} (I - A)^i. \tag{2.13}$$

La condition $\rho(I - A) < 1$ impose que la plus grande valeur propre λ_{max} de A vérifie

$$0 < \lambda_{max} < 2.$$

Nous utilisons alors la série tronquée $\sum_{i=0}^{k} (I - A)^i$ comme approximation de A^{-1} et comme préconditionnement polynomial. Pour se rapprocher au plus près de la condition

$\rho(I - A) < 1$, la matrice est utilisée avec un équilibrage (*scaling*) diagonal. Si D est la matrice diagonale constituée de la diagonale de A, nous avons :

$$(D^{-1}A)^{-1} = \sum_{i=0}^{\infty}(I - D^{-1}A)^i \tag{2.14}$$

ce qui revient à utiliser le préconditionnement $\sum_{i=0}^{k}(I - D^{-1}A)^i$.

Un tel préconditionnement est très simple d'utilisation, mais n'est pas très performant en terme d'amélioration de la convergence. De plus, il apparaît de nettes différences de convergence suivant que le degré du polynôme choisi est pair ou impair.

Nous abordons à présent un préconditionnement que nous utilisons avec les méthodes redémarrées et qui nécessite de connaître une partie de l'information spectrale de la matrice A. Cette information spectrale peut être extraite de la méthode redémarrée elle-même, c'est ce que nous verrons ici, ou bien à l'aide d'une méthode spectrale comme la méthode de la puissance (*power method*) ou la méthode d'Arnoldi [45]. Le préconditionnement n'est pas figé et évolue d'un groupe d'itérations à l'autre : il est adaptatif et peut s'utiliser avec une matrice elle-même préconditionnée. Dans la suite nous utiliserons le terme processus pour se référer à un groupe de m itérations contigües entre deux redémarrages consécutifs d'une méthode redémarrée.

Le polynôme p_k recherché est celui pour lequel $p_k(A)A - I_n$ est proche de la matrice nulle. Plusieurs méthodes existent qui tendent à chercher le polynôme p_k tel que $1 - p_k(x)x$ soit minimal au sens d'une certaine norme sur un ensemble H contenant le spectre de la matrice A. Nous n'allons étudier qu'un seul cas, celui pour lequel la norme choisie est la norme infinie et H est pris égal à une ellipse. Il existe cependant d'autres méthodes non exposées ici, qui effectuent d'autres choix en ce qui concerne H et la norme.

Le préconditionnement de Chebyshev cherche à calculer le polynôme p_k qui vérifie :

$$p_k = \arg\min_{s \in \mathbb{P}_k} \max_{x \in \sigma(A)} |1 - xs(x)| \tag{2.15}$$

avec $\sigma(A)$ le spectre de la matrice A et $\mathbb{P}_k$ l'ensemble des polynômes à coefficients dans $\mathbb{R}$ et de degré strictement inférieur à k.

Il n'est possible de résoudre ce problème qu'avec la connaissance de l'ensemble des valeurs propres de la matrice A, ce qui est tout aussi difficile que de résoudre le système linéaire considéré. Une autre solution consiste à résoudre :

$$p_k = \arg\min_{s \in \mathbb{P}_k} \max_{x \in H} |1 - xs(x)| \tag{2.16}$$

avec H choisi comme un ensemble connexe qui contient $\sigma(A)$ ou une approximation d'un sous-ensemble de $\sigma(A)$ que l'on aura approché à l'aide de la méthode de Krylov utilisée.

Un premier choix considéré par Manteuffel [32] a été de prendre H égal à une ellipse contenant les valeurs propres approchées de la matrice. Si $H = E(d, c, a)$ est une ellipse de centre d, de focales $d + c$, $d - c$, et de demi-grand axe a, alors les polynômes de Chebyshev définis ainsi

$$T_k(x) = \frac{C_k(\frac{d-x}{c})}{C_k(\frac{d}{c})}, \tag{2.17}$$

où C_k est le $k^{\text{ème}}$ polynôme de Chebyshev, défini par la récurrence

$$\begin{aligned} &C_0(x) = 1 \\ &C_1(x) = x \\ &C_{i+1}(x) = 2xC_i(x) - C_{i-1}(x) \quad \text{pour} \quad i \geq 1, \end{aligned} \tag{2.18}$$

tendent asymptotiquement vers la solution du problème

$$\begin{cases} \min\limits_{s \in \mathbb{P}_{k+1}} \max\limits_{x \in H} |s(x)|. \\ s(0) = 1 \end{cases} \tag{2.19}$$

Le polynôme p_k est alors choisi égal à $(1 - T_{k+1}(x))/x$.

L'idée de départ consistait à prendre le polynôme p_k trouvé et à l'appliquer au vecteur approché x_0. Ainsi x_k était mis à jour par $x_k = x_0 + p_k(A)r_0$ et le résidu correspondant valait $r_k = b - Ax_k = (I - Ap_k(A))r_0 = T_{k+1}(A)r_0$. Cette méthode sera présentée dans les paragraphes suivants.

Pour pouvoir calculer le polynôme p_k, diverses étapes sont à respecter :

1. calculer une approximation de certaines valeurs propres de la matrice. Nous recherchons les valeurs λ pour lesquelles il existe un vecteur $v \neq 0$ tel que

$$Av = \lambda v. \tag{2.20}$$

Durant le processus de Hessenberg généralisé, la matrice H_m calculée nous permet de résoudre ce problème de façon approchée en le projetant sur le sous-espace de Krylov K_m orthogonalement à $\mathrm{L}_m = \mathrm{Span}\{U_m\}$. Nous recherchons les couples $(v = V_m z, \lambda)$ tel que

$$U_m^T A V_m z = \lambda \, U_m^T V_m z. \tag{2.21}$$

Nous avons :

$$U_m^T A V_m = L_m H_m \tag{2.22}$$

avec $L_m = U_m^T V_m$ une matrice triangulaire inférieure (voir (1.14)) avec comme éléments diagonaux les facteurs normalisateurs $\eta_j = (v_j, u_j)$. La matrice L_m étant inversible, l'équation (2.21) devient :

$$H_m z = \lambda \, z. \tag{2.23}$$

Les couples $(V_m z_i, \lambda_i)_{1 \leq i \leq m}$ vérifiant (2.23) sont dits des valeurs et vecteurs de Ritz. Les couples qui auront convergé, seront pris pour former l'ellipse,

2. calculer les paramètres d, c et a de l'ellipse choisie. Différents choix peuvent être faits que nous n'exposerons pas [32, 45].

Les paramètres de l'ellipse étant connus, les coefficients du polynôme s'expriment et se calculent récursivement. Néanmoins ceux-ci ne sont pas formés et seules les récurrences permettant de les calculer sont utilisées. Ainsi lors de l'application du préconditionnement dans les caculs des vecteurs $z = M^{-1}v$ et $w = AM^{-1}v$, les formule de récurrence à trois

termes des polynômes de Chebyshev (2.18) sont utilisées sur des vecteurs. Le vecteur w est exactement égal à $Ap_k(A)v = (I - T_{k+1}(A))v$ et son calcul se déduit des relations de récurrence des polynômes T_k. D'après les relations (2.17), (2.18) et en notant $\sigma_k = C_k(d/c)$, nous avons :

$$\sigma_{k+1}T_{k+1}(x) = 2\frac{d-x}{c}\sigma_k T_k(x) - \sigma_{k-1}T_{k-1}(x). \tag{2.24}$$

Si nous notons $\rho_k = \sigma_k/\sigma_{k+1}$, l'équation précédente devient :

$$T_{k+1}(x) = 2\frac{d-x}{c}\rho_k T_k(x) - \rho_k\rho_{k-1}T_{k-1}(x). \tag{2.25}$$

La relation (2.18) nous permet de dériver les coefficients σ_k ainsi :

$$\begin{aligned} \sigma_{k+1} &= 2\frac{d}{c}\sigma_k - \sigma_{k-1} \\ &= 2\sigma_1\sigma_k - \sigma_{k-1} \end{aligned} \tag{2.26}$$

(2.27)

ce qui nous permet de calculer les coefficients ρ_k récursivement :

$$\begin{cases} \rho_0 = \frac{1}{\sigma_1} \\ \rho_{k+1} = \frac{1}{2\sigma_1 - \rho_k} \quad \text{pour } k \geq 0. \end{cases} \tag{2.28}$$

En notant $w_{-1} = 0$, et $w_0 = v$, le vecteur w est obtenu à l'aide du schéma itératif suivant :

$\sigma_1 = d/c, \quad \rho_{-1} = 0, \quad \rho_0 = 1/\sigma 1$
Pour $i = 0, \ldots, k$ Faire
  $w_{i+1} = 2\frac{dI_n - A}{c}\rho_i w_i - \rho_i\rho_{i-1}w_{i-1}$
  $\rho_{i+1} = \frac{1}{2\sigma_1 - \rho_i}$
Fin Pour
$w = v - w_{k+1}$.

En ce qui concerne le vecteur $z = p_k(A)v = A^{-1}(I_n - T_{k+1}(A))v$, nous le renommons z_k et initialisons z_0 à v. Soit d_k le vecteur défini par :

$$d_k = z_{k-1} - z_k = A^{-1}(T_{k+1}(A) - T_k(A))z_0. \tag{2.29}$$

D'après l'équation (2.28), $1 = (2\sigma_1 - \rho_k)\rho_{k+1}$ et nous obtenons :

$$\begin{aligned} T_{k+1}(x) - T_k(x) &= 2\tfrac{d-x}{c}\rho_k T_k(x) - \rho_k\rho_{k-1}T_{k-1}(x) - (2\sigma_1 - \rho_k)\rho_{k-1}T_k(x) \\ &= -2\tfrac{x}{c}\rho_k T_k(x) + \rho_{k-1}\rho_k(T_k(x) - T_{k-1}(x)), \end{aligned}$$

ce qui nous permet d'écrire

$$\frac{T_{k+1}(x) - T_k(x)}{x} = \rho_k\left(-\frac{2}{c}T_k(x) + \rho_{k-1}\frac{T_k(x) - T_{k-1}(x)}{x}\right) \tag{2.30}$$

et

$$d_k = \rho_k(-\frac{2}{c}w_k + \rho_{k-1}d_{k-1}). \tag{2.31}$$

De plus nous avons $w_k = Az_k$. L'algorithme de calcul de $z = z_k$ est donc le suivant

$z_0 = v, \quad d_0 = -2\frac{v}{d}, \quad \sigma_1 = \frac{d}{c}, \quad \rho_0 = \frac{1}{\sigma_1}$
Pour $i = 1, \ldots, k$ Faire
$z_i = z_{i-1} - d_{i-1}$
$w_i = Az_i$
$\rho_i = 1/(2\sigma_1 - \rho_{i-1})$
$d_i = \rho_i(-\frac{2}{c}w_i - \rho_{i-1}d_{i-1})$
Fin Pour
$z = z_k$.

Nous venons de voir comment se calculait l'action du préconditionnement sur un vecteur. L'algorithme final permettant d'utiliser ce préconditionneur consiste :

1. à effectuer un processus[3] d'une méthode de Krylov redémarrée, qui nous permet à partir de H_m de construire le préconditionnement polynomial,
2. puis de faire un processus de cette même méthode de Krylov cette fois-ci préconditionnée avec le préconditionnement polynomial.

Le procédé est ensuite réitéré en retournant en 1. L'algorithme final qui s'applique à la matrice préconditionnée $N^{-1}A$ est décrit ci-dessous

Algorithme 2.12 $[x, r]$ = *Preconditionnement-adaptatif-Krylov(A, N,* x_0, m_1, m_2,*k)*

1. *Initialisation :* $x = x_0$ *et* $r_0 = b - Ax$
2. *Résolution du système linéaire :*
   (a) $[V_{m+1}, \bar{H}_m]$ = *Hessenberg-généralisé*$(N^{-1}A, r, m_1, U_{m_1})$
   (b) *Calcul de* x_{m_1} *et* r_{m_1} *avec la méthode redémarrée choisie*
   (c) $x = x_{m_1}$ *et* $r = r_{m_1}$
3. *Construction du préconditionnement* p_k *:*
   (a) *Résolution de* $H_m\lambda = \lambda\ z$
   (b) *Mise à jour de H*
   (c) *Calcul des coefficients de* p_k
4. *Résolution du système linéaire préconditionné :*
   (a) $[V_{m+1}, \bar{H}_m]$ = *Hessenberg-généralisé*$(p_k(N^{-1}A)N^{-1}A, r, m_2, U_{m_2})$
   (b) *Calcul de* x_{m_2} *et* r_{m_2} *avec la méthode redémarrée choisie*
   (c) $x = x_{m_2}$ *et* $r = r_{m_2}$

3. Nous appelons processus pour une méthode redémarrée dont la dimension du sous-espace de Krylov est m, l'ensemble de m itérations contigües entre deux redémarrages consécutifs

(d) Si convergence fin, sinon aller en 2.

D'autres méthodes ont vu le jour, qui consistent à choisir différemment H et la norme utilisée. Smolarski et Saylor ont proposé de prendre H égal à un polygone, ce qui permet de mieux approcher le spectre de la matrice lorsque sa forme générale n'a rien avoir avec une ellipse, et d'utiliser la norme 2. Saad a alors proposé d'utiliser ce polygone avec la norme issue d'un produit scalaire intégral directement lié à ce polygone, où, en fait, aucun calcul intégral n'est nécessaire. Il propose également d'exprimer le polynôme solution non pas dans la base $(x^i)_{0\leq i\leq k}$ qui est mal conditionnée, mais dans la base des polynômes de Chebyshev associés à l'ellipse d'aire maximale contenant le polygone. L'algorithme est détaillé dans [43]. D'autres références pour différents choix de H et de la norme peuvent être trouvées dans [37].

2.2 Méthodes par blocs

Les méthodes par blocs s'utilisent surtout lorsque l'on désire résoudre plusieurs systèmes linéaires simultanément avec des seconds membres différents. Supposons qu'il y en a s :

$$Ax^i = b^i \quad \text{pour} \quad i = 1, \ldots, s. \tag{2.32}$$

Nous notons $r_0^i = b^i - Ax_0^i$ les résidus initiaux correspondant aux vecteurs x_0^i.

Comme les méthodes de Krylov génèrent une base du sous-espace de Krylov

$$\mathrm{K}_m(A, r_0),$$

les méthodes de Krylov par blocs génèrent une base du sous-espace suivant

$$\mathrm{Span}\{\mathrm{K}_m(A, r_0^1), \ldots, \mathrm{K}_m(A, r_0^s)\} \tag{2.33}$$

de dimension $m \times s$, contenant s sous-espaces de Krylov. Or ce sous-espace est exactement égal à

$$\mathrm{K}_m(A, R_0) = \mathrm{Span}\{R_0, AR_0, \ldots, A^{m-1}R_0\} \tag{2.34}$$

avec $R_0 = [r_0^1, \ldots, r_0^s]$ la matrice des résidus initiaux. Pour construire une base de ce sous-espace, nous allons utiliser exactement les mêmes algorithmes que dans le chapitre 1 et ici avec le processus de Hessenberg généralisé, mais cette fois-ci sur des blocs : les vecteurs v_i et u_i sont remplacés par des blocs V_i et U_i de dimension $n \times s$. L'algorithme correspondant est le processus de Hessenberg généralisé par blocs :

- chaque produit scalaire (v_j, u_i) est remplacé par un produit matrice matrice $U_i^T V_j$ de dimensions $(s \times n)$ par $(n \times s)$,
- chaque produit matrice vecteur Av_j est remplacé par un produit matrice matrice AV_j de dimensions $(n \times n)$ par $(n \times s)$,
- chaque opération triadique $w = w - h_{i,j}v_j$ est remplacée par une opération matricielle $W = W - V_j \times H_{i,j}$ avec $H_{i,j}$ une matrice de dimension $s \times s$ et W et V_j des matrices de taille $n \times s$.

Comme dans le cas par point, il est nécessaire de définir les matrices $U_i = [u_{(i-1)s+1}, \ldots, u_{is}]$ pour $i = 1, \ldots, m$, soit en les fixant à l'avance, soit en les construisant au fur et à mesure que les matrices $V_i = [v_{(i-1)s+1}, \ldots, v_{is}]$ sont elles-mêmes construites. Nous noterons Υ_m la matrice formée de l'ensemble des vecteurs des matrices U_i pour $i = 1, \ldots, m$:

$$\Upsilon_m = [U_1, \ldots, U_m] = [u_1, \ldots, u_{ms}], \tag{2.35}$$

et de la même façon,

$$\Psi_m = [V_1, \ldots, V_m] = [v_1, \ldots, v_{ms}]. \tag{2.36}$$

L'algorithme résultant est le suivant :

Algorithme 2.13 $[\Psi_{m+1}, \bar{H}_m]$ = *Hessenberg-généralisé-blocs (A,m,s,* R_0, Υ_m*)*

1. *Initialisation :*
   (a) *Choix de* B_0 *non singulière et calcul de* $V_1 = R_0 B_0^{-1}$
   (b) *Calcul de* $\Theta_1 = U_1^T V_1$
2. *Boucle principale*
   Pour $j = 1, \ldots, m$ *Faire*
   (a) $W = AV_j$
   (b) *Pour* $i = 1, \ldots, j$ *Faire*
   $H_{i,j} = \Theta_i^{-1} U_i^T W$
   $W = W - V_i H_{i,j}$
   Fin Pour
   (c) *Choix de* $H_{j+1,j}$ *non singulière et calcul de* $V_{j+1} = W H_{j+1,j}^{-1}$
   (d) *Calcul de* $\Theta_{j+1} = U_{j+1}^T V_{j+1}$

   Fin Pour
3. *Définir* $\Psi_{m+1} = [V_1, \ldots, V_{m+1}]$, $\bar{H}_m = \{H_{i,j}\}$.

La matrice $\bar{H}_m$ est la matrice formée des matrices blocs $H_{i,j}$. De façon à conserver sa structure, les blocs $H_{i+1,i}$ sont choisis triangulaires : la matrice $\bar{H}_m$ est ainsi une matrice $(m+1)s \times ms$ de Hessenberg par bande à s sous-diagonales. Dans le cas $s = 1$, on retrouve la forme de Hessenberg de $\bar{H}_m$.

Nous retrouvons les relations (1.2) et (1.3) qui, appliquées ici aux matrices blocs, s'écrivent :

$$A\Psi_m = \Psi_{m+1}\bar{H}_m \tag{2.37}$$

$$= \Psi_m H_m + V_{m+1} H_{m+1,m} E_{m,s}^{(ms)T} \tag{2.38}$$

avec $E_{i,j}^{(k)} = [e_{(i-1)j+1}^{(k)}, \ldots, e_{ij}^{(k)}]$.

Une fois la base du sous-espace construite, les méthodes par blocs calculent les solutions approchées

$$x_m^i = x_0^i + \Psi_m y_m^i \quad \text{pour} \quad i = 1, \ldots, s \tag{2.39}$$

avec y_m^i un vecteur de dimension ms. Matriciellement, cette équation s'écrit :

$$X_m = X_0 + \Psi_m Y_m \tag{2.40}$$

avec $X_m = [x_m^1, \ldots, x_m^s]$ et $Y_m = [y_m^1, \ldots y_m^s]$. Une fois les vecteurs y_m^i calculés, la phase de projection, matérialisée par l'équation précédente, peut donc se faire grâce au produit matrice matrice $\Psi_m Y_m$. Le résidu correspondant est alors égal à :

$$\begin{aligned} R_m &= B - A(X_0 + \Psi_m Y_m) && (2.41)\\ &= R_0 - A\Psi_m Y_m && (2.42)\\ &= R_0 - \Psi_{m+1}\bar{H}_m Y_m && (2.43)\\ &= \Psi_{m+1}(E_{1,s}^{((m+1)s)} B_0 - \bar{H}_m Y_m) && (2.44) \end{aligned}$$

avec $B = [b^1, \ldots, b^s]$ et B_0 la matrice coefficient normalisateur de R_0.

Dans le cas des méthodes **PO**, nous calculons la solution projetée x_m^i telle que :

$$\begin{cases} (x_m^i - x_0^i) \in \mathrm{K}_m(A, R_0) \\ r_m^i \perp_{(\Psi_{m+1}^{Left})^T \Psi_{m+1}^{Left}} \mathrm{K}_m(A, R_0), \end{cases} \tag{2.45}$$

ce qui nous amène à résoudre les s systèmes linéaires suivants :

$$y_m^i = H_m^{-1}(E_{1,s}^{((m+1)s)} B_0 e_i^{(s)}) \text{ pour } i = 1, \ldots, s. \tag{2.46}$$

Dans le cas des méthodes **PO**, nous calculons la solution projetée x_m^i telle que :

$$\begin{cases} (x_m^i - x_0^i) \in \mathrm{K}_m(A, R_0) \\ r_m^i = \min \|b - Ax_m^i\|_{(\Psi_{m+1}^{Left})^T \Psi_{m+1}^{Left}}. \end{cases} \tag{2.47}$$

Comme dans le cas par point $\Psi_{m+1}^{Left} = (\Upsilon_{m+1}^T \Psi_{m+1})^{-1} \Upsilon_{m+1}^T$ n'existe que si la matrice $\Upsilon_{m+1}^T \Psi_{m+1}$ est inversible. Or nous avons la relation :

$$U_i^T V_j = \Theta_i \delta_{i,j} \text{ pour } j = 1, \ldots, m+1 \text{ et } i = 1, \ldots, j \tag{2.48}$$

La matrice $\Upsilon_{m+1}^T \Psi_{m+1}$ est donc triangulaire inférieure par blocs, avec les matrices Θ_i sur sa diagonale. Elle est donc inversible si toutes ces matrices sont inversibles. Nous résolvons donc pour les méthodes **PM** les s problèmes de moindres carrés suivants :

$$\begin{aligned} y_m^i &= \arg \min_{y \in \mathbb{R}^{ms}} \|b^i - A\Psi_m y\|_{(\Psi_{m+1}^{Left})^T \Psi_{m+1}^{Left}} && (2.49)\\ &= \arg \min_{y \in \mathbb{R}^{ms}} \|g^i - \bar{H}_m y\|_2 && (2.50) \end{aligned}$$

avec $g^i = E_{1,s}^{((m+1)s)} B_0 e_i^{(s)}$.

Pour cela, nous transformons la matrice bande de Hessenberg $\bar{H}_m$ en une matrice triangulaire supérieure à l'aide de la matrice Q_m de taille $(m+1)s \times (m+1)s$, produit de matrices de rotations de Givens, soit

$$Q_m \bar{H}_m = \bar{R}_m = \begin{pmatrix} R_m \\ O_{s \times ms} \end{pmatrix} \tag{2.51}$$

avec R_m une matrice triangulaire supérieure de dimension $ms \times ms$. La relation (2.50) s'écrit :

$$y_m^i = \arg \min_{y \in \mathbb{R}^{ms}} \|Q_m g^i - \bar{R}_m y\|_2. \tag{2.52}$$

Comparons à présent la résolution de s systèmes linéaires grâce à une méthode de Krylov **PO** ou **PM** par blocs, qui construit le sous-espace défini par (2.34) avec s méthodes de Krylov respectivement **PO** ou **PM**, résolvant ces s systèmes de façon indépendante en construisant chacune un sous-espace de Krylov $\mathrm{K}_m(A, r_0^i)$ de dimension ms pour $i = 1, \ldots, s$:

- la première solution construit un sous-espace de dimension ms pour effectuer la projection, tandis que la seconde en construit s de dimension ms. La première solution permet donc d'effectuer $(s-1)ms$ produits matrice vecteur en moins,

- la première solution transforme la matrice de Hessenberg par bande de dimension $(ms+s) \times ms$ en une matrice triangulaire supérieure, ce qui n'est pas plus coûteux que de transformer les s matrices de Hessenberg de dimension $(ms+1) \times ms$ de la seconde solution en une matrice triangulaire supérieure,

- la première solution et la seconde résolvent s systèmes linéaires,

- la première solution effectue la projection définie par (2.40) à l'aide d'un produit matrice matrice, ce qui est équivalent au coût des s produits matrice vecteur effectués lors du calcul des s solutions approchées de la seconde solution.

Nous voyons donc que les méthodes par blocs permettent de diminuer le nombre de produits matrice vecteur, mais surtout d'utiliser les **BLAS** de plus haut niveau. Les **BLAS**, acronyme de **B**asic **L**inear **A**lgebra **S**ubprogram, sont des sous-programmes monoprocesseur, initialement écrits en f77, optimisés sur chaque machine (éventuellement réécrits en assembleur) et qui permettent de faire les opérations les plus élémentaires de l'algèbre linéaire pleine. Les **BLAS1** regroupent les opérations entre vecteurs, les **BLAS2** entre matrices et vecteurs et les **BLAS3** entre matrices seules. Les **BLAS3** sont plus efficaces et permettent, en général un nombre d'opérations par seconde plus élevé que les **BLAS1** et **BLAS2**. Les méthodes par blocs utilisent les **BLAS3** là où les méthodes par point utilisent des **BLAS1** et **BLAS2**, ce qui leur permet d'être plus efficaces. Les méthodes par blocs sont également plus robustes, car elles peuvent approcher des valeurs propres défectives de la matrice A et ainsi accélérer la convergence de la méthode, ce qui n'est pas possible dans le cas des méthodes par point, car la matrice $\bar{H}_m$ utilisée est une matrice de Hessenberg irréductible.

Ces techniques peuvent également s'appliquer aux méthodes non redémarrées : c'est le cas notamment des méthodes **CG**, **Bi-CG** [38] et **QMR** [19].

2.3 Méthodes hybrides

2.3.1 Principe

Nous présentons dans cette section quelques méthodes appelées hybrides. Le terme *hybride* regroupe de nombreux aspects, à la fois algorithmique et logiciel. Nous essayons d'en extraire un schéma général. Nous pouvons dire ceci d'une méthode hybride :

- une méthode hybride est une technique d'accélération qui vise donc, soit à diminuer le nombre d'itérations d'une méthode spécifique pour converger d'un point de vue algorithmique, soit à l'accélérer d'un point de vue efficacité sur machine séquentielle ou parallèle, en l'implantant de la façon la plus appropriée qui soit,

- une méthode hybride utilise au moins deux méthodes classiques qui unies permettent, soit d'apporter une information supplémentaire que la méthode initiale ne peut acquérir seule, soit de la décharger de certaines opérations qui peuvent être coûteuses.

Nous présentons deux accélérations de type algorithmique ainsi qu'une accélération à la fois algorithmique et logicielle en environnements parallèles. Mais naturellement, cette présentation est loin d'être exhaustive.

Nous abordons en premier l'accélération polynomiale fortement liée aux préconditionnements polynomiaux vus plus haut.

2.3.2 Accélération polynomiale

Une accélération polynomiale cherche à approcher un polynôme de degré donné de telle sorte que lorsqu'il s'applique une ou plusieurs fois à un résidu r correspondant à une solution approchée x, la norme de ce nouveau résidu soit amoindrie ou tout au moins, que ce nouveau résidu soit devenu négligeable dans un sous-espace donné. En général, ce sous-espace est un sous-espace invariant que nous aurons approché.

La méthode se scinde donc en deux :

1. calcul du polynôme p_k,

2. application de ce même polynôme. Retour en 1.

Il existe de nombreuses variantes suivant que l'on utilise ou non une méthode de Krylov pour approcher la solution, voire pour calculer le polynôme p_k. Le choix du polynôme lui-même est, naturellement, un critère ainsi que la façon dont il est appliqué. Toutes ces méthodes sont référencées dans [37].

Ainsi, Manteuffel, la première personne à avoir introduit ce type d'algorithme [33], utilisait la méthode de la puissance pour approcher les valeurs propres de la matrice. Puis, comme cela a été fait avec les préconditionnements polynomiaux, il déterminait le polynôme de norme minimal au sens de la norme infinie sur une ellipse H contenant ces valeurs propres approchées. La deuxième phase consistait alors à appliquer ce polynôme sur le résidu.

D'autres normes et d'autres ensembles différents des ellipses peuvent être utilisés. Nous avons vu avec les préconditionnements polynomiaux, le cas de l'ensemble H pris égal à un polygone et de la norme choisie égale à la norme 2 ou à la norme intégrale [43].

Il est également possible de combiner la recherche du polynôme p_k avec le calcul de la solution approchée, notamment avec des méthodes redémarrées, qui construisent explicitement la matrice H_m issue de la projection de la matrice A sur le sous-espace de Krylov $\mathrm{K}_m(A, r_0)$ orthogonalement au sous-espace des contraintes.

Enfin, le polynôme lui-même peut être appliqué soit à l'aide de récurrence impliquant des vecteurs, comme nous l'avons vu dans le cas des préconditionnements polynomiaux, soit à l'aide de la méthode de Richardson. Cette dernière nécessite de connaître les zéros du polynôme.

Dans [37], les auteurs appliquent le polynôme résiduel calculé à l'aide de la méthode **GMRES**(m) plusieurs fois lorsque celui-ci a suffisament fait diminuer la norme du résidu. Ce polynôme est appliqué grâce à la méthode de Richardson, où les zéros ont été ordonnés à l'aide du réordonnancement de Leja [7, 8].

Les techniques d'accélération polynomiales sont donc très variées.

2.3.3 La procédure hybride de Brezinski et Redivo Zaglia

Nous présentons dans ce paragraphe une accélération qui consiste à tirer partie de plusieurs méthodes itératives pour obtenir une meilleure solution approchée. Ces méthodes itératives peuvent être des méthodes stationnaires et naturellement des méthodes de Krylov. Nous ne nous attacherons ici qu'au cas des méthodes de Krylov.

La procédure hybride [4] consiste, à partir de k méthodes de Krylov calculant chacune une solution approchée $(x_m^i)_{1\leq i\leq k}$ de résidu respectif $(r_m^k)_{1\leq i\leq k}$, à construire la solution approchée dont le résidu correspondant est le suivant :

$$\rho_m^k = \sum_{i=1}^{k} a_m^i r_m^i \tag{2.53}$$

tel que $\sum_{i=1}^{k} a_m^i = 1$. La solution approchée est calculée grâce à $y_m^k = \sum_{i=1}^{k} a_m^i x_m^i$. Différents choix pour les coefficients a_m^i sont possibles, notamment celui pour lequel ρ_m^k est minimal pour la norme 2.

La procédure hybride englobe plusieurs méthodes connues :

- lorsque $x_m^i = x_{m-i+1}$, nous retrouvons les méthodes semi-itératives,
- lorsque $k = 2$, $x_m^1 = y_{m-1}^2$ et x_m^2 est issue d'une méthode de Krylov donnée, nous retrouvons les méthodes de lissage [61] : celles qui minimisent la norme 2 de ρ_m^2, appelée *minimal residual smoothing* et celle qui permet de calculer les itérés de la méthode **QMR** à partir de ceux du **Bi-CG**, dénommée *quasi-minimal residual smoothing.*

De nombreuses autres possibilités existent qui sont mentionnées dans [3].

2.3.4 Exemple

Nous voyons à présent un exemple de méthode hybride à la fois liée à une accélération algorithmique et logicielle : c'est la méthode hybride parallèle asynchrone **GMRES/LS-Arnoldi** développée dans [2, 14], qui implante une méthode hybride de type polynomial. La méthode de calcul de la solution est **GMRES**(m), tandis que la méthode de calcul du polynôme est la méthode **L**east **S**quare **A**rnoldi. L'approximation des valeurs propres s'effectue avec la méthode d'Arnoldi. Puis le polynôme minimal au sens de la norme intégrale de [43] sur un polygone contenant ces valeurs propres est appliqué.

Tout algorithme comporte des zones de calcul, dont certaines sont plus adaptées au parallélisme de tâches et d'autres au parallélisme de données. L'originalité de la méthode consiste à différencier les différents types de parallélisme qui existent au sein d'une méthode et à les répartir convenablement sur un réseau hétérogène, comportant une MASPAR, une ferme d'Alpha et deux stations de travail.

2.4 Techniques liées à GMRES

De nombreuses techniques d'accélération ont été développées pour chaque méthode de Krylov, mais surtout pour **GMRES**(m). Nous nous occupons ici de cette méthode et abordons une technique qui permet de faire varier le préconditionnement à l'intérieur même d'un processus : c'est la méthode **F**lexible **GMRES**. Nous verrons dans les chapitres suivants d'autres techniques permettant de pallier à différents inconvénients de **GMRES**(m).

La méthode **FGMRES**(m) construit une base orthonormale $(v_i)_{1\leq i\leq m}$ du sous-espace suivant :

$$\{r_0, (AM_1^{-1})r_0, \ldots, \prod_{i=1}^{m-1}(AM_{m-i}^{-1})r_0\}$$

et calcule la solution dont la norme du résidu est minimale sur ce sous-espace, permettant ainsi à chaque pas de faire varier le préconditionnement au sein d'un même processus. L'algorithme qui permet de construire une base orthonormale d'un tel sous-espace est le suivant :

Algorithme 2.14 $[x_m, r_m, Z_m, V_{m+1}, \bar{H}_m]$ = *FGMRES(A,x_0,m)*

1. *Initialisation :* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $v_1 = r_0/\beta$
2. *Algorithme de Gram-Schmidt modifié*
   Pour $j = 1, \ldots, m$ *Faire*
   (a) $z_j = M_j^{-1} v_j$
   (b) $w = Az_j$
   (c) *Pour* $i = 1, \ldots, j$ *Faire*
   $h_{i,j} = (w, v_i)$
   $w = w - h_{i,j} v_i$
   Fin Pour

(d) $h_{j+1,j} = \|w\|_2$

(e) *si* $h_{j+1,j} \neq 0$ *alors* $v_{j+1} = w/h_{j+1,j}$ *sinon stop*

Fin Pour

3. *Calcul de la solution approchée:*

   - $z_m = \arg\min_z \|\beta e_1^{(m+1)} - \bar{H}_m z\|_2$
   - $x_m = x_0 + Z_m z_m$
   - $r_m = r_0 - AZ_m z_m$.

Si nous définissons la matrice Z_m par $[z_1, \ldots, z_m]$, elle satisfait:

$$AZ_m = V_{m+1}\bar{H}_m,$$

qui remplace donc l'équation classique (1.2).

L'algorithme **FMGRES** est en fait bien plus général qu'il n'y paraît et permet d'enrichir un sous-espace de Krylov de vecteurs quelconques. Ainsi rien ne nous empêche de choisir $z_j = v_j$ pour $j = 1, \ldots, m-k$, puis de prendre $z_i = u_i$ pour $i = m-k+1, \ldots, m$, les vecteurs u_i étant indépendants des vecteurs v_i.

Nous étudierons au chapitre 4 une de ces techniques, qui enrichit le sous-espace de Krylov avec des vecteurs u_i égaux à des vecteurs propres ou des approximations de vecteurs propres.

Une autre idée consiste à choisir le vecteur u_{i+1} comme solution approchée de la solution du système $Ax = v_i$. En effet, si ce dernier système est résolu exactement, la solution calculée par la méthode **FGMRES**(m) est alors la solution du système $Ax = b$. Une solution consiste à résoudre le système $Ax = v_i$ avec la méthode **GMRES**(m), de façon approchée, en se fixant à l'avance le nombre de processus et la dimension du sous-espace utilisés: nous appliquons l'algorithme **GMRES**(m) au sein de **FGMRES**(m).

2.5 Conclusion

Nous avons présenté de façon non exhaustive quelques techniques permettant d'accélérer les méthodes de Krylov sur machines parallèles. Une première solution consiste à accélérer les méthodes de Krylov au niveau de la convergence et de la robustesse, en faisant en sorte que la méthode converge en un nombre d'itérations et un coût moindre. Cela permet sur machines parallèles de réduire le nombre total des opérations et donc aussi des opérations les plus coûteuses. Nous avons vu une seconde accélération qui est plus liée au désir de supprimer les zones de calcul les moins parallèles d'un algorithme ou tout du moins de les implanter plus efficacement.

Nous voyons dans les chapitres suivants deux techniques qui accélèrent les méthodes **FOM**(m) et **GMRES**(m). La première est une méthode qui vise à enlever leur partie la moins parallèle, que nous avons déjà évoquée dans le chapitre précédent: les produits scalaires du processus de Hessenberg généralisé et ici d'Arnoldi constituent une perte de temps importante à cause des différentes synchronisations effectuées en séquence, qu'ils

imposent. Nous proposons, pour y remédier, de changer de produit scalaire et de le choisir discret. Cette technique est développée dans le chapitre 3.

Nous abordons dans le chapitre 4 une autre technique, visant, cette fois-ci à réduire le nombre total d'itérations de telle sorte que, globalement, l'algorithme soit moins coûteux. La méthode tente de pallier aux pertes d'informations dues aux redémarrages des méthodes **FOM**(m) et **GMRES**(m) en enrichissant le sous-espace de Krylov sur lequel s'effectue la projection. Ceci est fait de façon implicite grâce à la méthode **IRA** [58] due à Sorensen.

Chapitre 3

Accélération liée au produit scalaire

Sommaire

Nous nous intéressons dans ce chapitre aux méthodes redémarrées utilisant le processus de Hessenberg généralisé. En effet, nous avons vu dans le chapitre 1 que la performance parallèle de ces méthodes était limitée par la construction de la base de Krylov. Lorsque nous utilisons l'algorithme de Gram-Schmidt modifié (**MGS**) dans le processus d'Arnoldi, de même coût que l'algorithme de Gram-Schmidt classique (**CGS**), mais plus stable numériquement, les produits scalaires en jeux se font en séquence, ce qui peut diminuer de façon sévère les performances de la méthode sur machines parallèles. Une autre alternative déjà mentionnnée dans le chapitre 1, consiste à utiliser le processus de Gram-Schmidt classique, qui est plus parallèle, puisqu'il induit des produits matrice vecteur au lieu de produits scalaire et à réorthogonaliser les vecteurs de la base une seconde fois. Cette version présente le désavantage d'être deux fois plus coûteux en terme de complexité et d'être encore séquentielle : nous ne l'utiliserons pas. En revanche, dans la seconde partie des tests exposés à la fin de ce chapitre, nous comparons la méthode accélératrice présentée dans ce chapitre avec **CGS** seul.

Nous développons dans ce chapitre deux méthodes de type **RQO** et **RQM**, visant à pallier le problème de séquentialité de l'algorithme **MGS** en choisissant un produit scalaire beaucoup moins onéreux. Afin de le construire, nous prenons, comme point de départ, les méthodes **FOM**(m) et **GMRES**(m) et considérons l'espace de Krylov construit au travers du processus d'Arnoldi comme un espace de polynômes. Nous définissons alors un produit scalaire sur l'espace de Krylov à partir d'un produit scalaire polynomial, que nous choisissons discret, pour qu'il soit moins coûteux. Ce dernier utilise un ensemble de valeurs propres approchées de la matrice A de dimension réduite, que nous mettons à jour à chaque redémarrage. Dans le cas de la méthode **PO**, la solution approchée est calculée de telle sorte que le résidu correspondant soit orthogonal à K_m au sens de la norme induite par le produit scalaire discret. De la même façon, dans le cas de la méthode **PM**, le problème de moindres carrés sous-jacent est résolu avec le produit scalaire discret, ce qui est équivalent à résoudre un problème de moindres carrés sur l'espace des polynômes.

Dans le cas d'une matrice à spectre réel, le produit scalaire discret permet de construire les vecteurs de la base de Krylov à l'aide d'une récurrence à trois termes. Lorsque le spectre de la matrice est complexe, la récurrence devient longue. Nous choisissons alors un autre produit scalaire dérivé du précédent, afin de conserver cette récurrence courte et surtout le calcul en arithmétique réelle.

Nous exposons tout d'abord le principe de la méthode. Nous considérons en premier lieu le cas où le spectre de la matrice est réel, puis adaptons la technique au cas des matrices à spectre complexe. Nous comparons ensuite les coûts de stockage des nouvelles méthodes **RQO** et **RQM** à **FOM**(m) et **GMRES**(m) ainsi que leur complexité en temps. Nous analysons enfin les résultats numériques.

Cette technique a été présentée à IMACS World Congress'97 à Berlin et a été développée dans [29].

3.1 Principe

Nous avons vu dans le chapitre 1 que, outre les produits matrice vecteur et les opérations de préconditionnement, la construction de la base de Krylov dans le processus d'Arnoldi (et le processus de Hessenberg généralisé de façon générale) constitue une des phases les plus pénalisantes des méthodes redémarrées sur machines parallèles. En effet chaque nouveau vecteur de la base de Krylov dépend directement de tous les vecteurs de cette même base déjà construits. Ainsi, pour le calcul de v_{i+1}, il est nécessaire de connaître tous les vecteurs $(v_k)_{1\leq k\leq i}$, de façon à multiplier le vecteur v_i par A et à l'orthogonaliser par rapport à ces mêmes vecteurs $(v_k)_{1\leq k\leq i}$. Cette orthogonalisation peut se faire d'une façon plus parallèle au travers de la version classique du processus d'Arnoldi notée **CGS** pour **C**lassical **G**ram-**S**chmidt et dont la boucle principale est décrite ci-dessous

Pour $j = 1, \ldots, m$ Faire
  $w = Av_j$
  $h_{1:j,j} = V_j^T w$
  $w = w - V_j h_{1:j,j}$
  $v_{j+1} = w/\|w\|_2$
Fin Pour

ou bien à l'aide d'une version, dite modifiée notée **MGS** pour **M**odified **G**ram-**S**chmidt, qui est plus stable numériquement. Elle induit moins de perte d'orthogonalité entre les vecteurs v_i, par rapport à la version classique. Une autre possibilité afin de conserver une bonne orthogonalité entre les vecteurs de la base, consiste à utiliser **MGS** avec réorthogonalisation. Cette solution a le désavantage d'être deux fois plus coûteuse en nombre d'opérations et en volume de communications. Nous ne l'utiliserons pas.

Nous considérons à présent l'algorithme **MGS** qui représente la version utilisée dans les algorithmes 1.1 et 1.2. La boucle principale est la suivante :

Pour $j = 1, \ldots, m$ Faire
  $w = Av_j$
  Pour $i = 1, \ldots, j$
    $h_{i,j} = v_i^T w$
    $w = w - h_{i,j} v_i$
  Fin Pour
  $v_{j+1} = w/\|w\|_2$
Fin Pour

À chaque pas, nous effectuons donc soit 1 produit matrice vecteur de dimensions $(j \times n)$par n avec l'algorithme **CGS**, soit j produits scalaires de dimension n avec l'algorithme **MGS**. Ces deux versions sont mathématiquement équivalentes et de même coût. Cependant, la première permet d'utiliser les **BLAS2** (voir chapitre 2) et d'effectuer une communication sur un vecteur de taille j pour chaque pas de la boucle j. La seconde utilise les **BLAS1** et effectue j communications sur un scalaire pour chaque pas de la boucle j. Il y a donc davantage de synchronisations avec l'algorithme **MGS** qu'avec celui **CGS** et il utilise des sous-programmes moins performants que pour **CGS**. **MGS** est donc moins parallèle que

CGS. Nous utiliserons l'algorithme **MGS** dans la première série de tests, puis l'algorithme **CGS** dans la seconde, mais nous ne parlerons que de produits scalaires, même quand il s'agit de produits matrice vecteur, car ils peuvent tous se mettre sous cette forme.

Notre technique consiste à remplacer le produit scalaire euclidien $(w, v_i) = v_i^T w$ par un produit scalaire discret, en se plaçant d'un point de vue polynomial. La base de Krylov construite sera alors orthogonale pour ce produit scalaire discret.

Tout vecteur w de l'espace de Krylov $\mathrm{K}_m(A, r_0)$ peut se mettre sous la forme $w = q(A)r_0$ où q appartient à l'espace $\mathbb{P}_m$ des polynômes à coefficients dans $\mathbb{R}$ et de degré strictement inférieur à m. De cette façon, il est permis de définir un produit scalaire sur l'ensemble $\mathbb{P}_m$ à partir du produit scalaire euclidien restreint à K_m :

$$\begin{aligned} \langle \cdot, \cdot \rangle : \quad \mathbb{P}_m \times \mathbb{P}_m &\to \mathbb{R} \\ \langle p_1, p_2 \rangle &= (p_1(A)r_0, p_2(A)r_0) \end{aligned} \tag{3.1}$$

où $(\cdot, \cdot)$ représente le produit scalaire euclidien usuel sur $\mathbb{R}^n \times \mathbb{R}^n$, que nous restreignons ici à $\mathrm{K}_m \times \mathrm{K}_m$.

Pour que $\langle \cdot, \cdot \rangle$ soit une forme bilinéaire symétrique définie positive sur $\mathbb{P}_m \times \mathbb{P}_m$, il faut que K_m soit de dimension exactement égale à m, ce que nous supposerons par la suite.

À l'inverse et c'est cette propriété que nous utiliserons dans ce chapitre, il est également possible de définir un produit scalaire sur K_m à partir d'un produit scalaire polynomial quelconque sur $\mathbb{P}_m$:

$$\begin{aligned} (\cdot, \cdot)_d : \quad \mathrm{K}_m \times \mathrm{K}_m &\to \mathbb{R} \\ (w_0, w_1)_d &= \langle p_1, p_2 \rangle \end{aligned} \tag{3.2}$$

où $(\cdot, \cdot)_d$ est notre nouveau produit scalaire sur K_m et p_1 et p_2 sont les uniques polynômes tels que $w_0 = p_1(A)r_0$ et $w_1 = p_2(A)r_0$.

Il est important de noter que n'importe quel produit scalaire polynomial pouvant prétendre à porter ce nom (forme bilinéaire symétrique définie positive) convient. Notre objectif étant de faire en sorte que le produit scalaire $(\cdot, \cdot)_d$ soit le moins onéreux possible sur machines parallèles et puisque celui-ci dépend directement du produit scalaire polynomial $\langle \cdot, \cdot \rangle$, le choix de $\langle \cdot, \cdot \rangle$ est très important. Une fois ce dernier choisi, le produit scalaire $(\cdot, \cdot)_d$ est défini et nous pouvons remplacer le produit scalaire euclidien du processus d'Arnoldi par ce nouveau produit scalaire.

Nous construisons ensuite une base de Krylov orthonormale pour notre nouveau produit scalaire, que nous notons $(w_i)_{1 \leq i \leq m+1}$ pour la différencier de celle issue du produit scalaire euclidien. Nous calculons alors les solutions **PO** et **PM** correspondant à cette base. Rappelons que d'après le chapitre 1, si nous notons $W_{m+1} = [w_1, \ldots, w_{m+1}]$, les résidus r_m^{PO} et r_m^{PM} vérifient respectivement :

$$r_m^{PO} \perp_{(W_{m+1}^{Left})^T W_{m+1}^{Left}} \mathrm{K}_m(A, r_0)$$

et

$$r_m^{PM} \perp_{(W_{m+1}^{Left})^T W_{m+1}^{Left}} A\mathrm{K}_m(A, r_0)$$

avec $W_{m+1}^{Left} = (W_{m+1}^T W_{m+1})^{-1} W_{m+1}^T$, qui est exactement égale à la matrice pseudo-inverse de W_{m+1} et que nous noterons $W_{m+1}^{\dagger}$.

Nous voyons donc que pour définir le résidu à l'itération m, il est nécessaire de connaître le vecteur w_{m+1} et par conséquent le polynôme p_m de degré m. Le produit scalaire $\langle \cdot, \cdot \rangle$ doit donc être défini sur $\mathbb{P}_{m+1}$ et non $\mathbb{P}_m$. De la même façon, $(\cdot, \cdot)_d$ doit être défini sur K_{m+1} et non sur K_m.

Nous considérons tout d'abord le cas des matrices à spectre réel.

3.2 Matrices à spectre réel

La différence entre la technique utilisée pour les matrices à spectre réel et celles à spectre complexe se situe dans le choix du produit scalaire polynomial. Celui choisi pour les matrices à spectre réel induit une récurrence courte à trois termes pour les vecteurs $(w_i)_{1 \leq i \leq m}$ de la base de Krylov.

3.2.1 Choix du produit scalaire polynomial

Nous définissons le produit scalaire suivant sur l'ensemble $\mathbb{P}_{m+1}$:

$$\begin{aligned} \langle \cdot, \cdot \rangle : \quad \mathbb{P}_{m+1} \times \mathbb{P}_{m+1} &\to \mathbb{R} \\ \langle p, q \rangle &= \sum_{\theta_k \in \mathrm{S}} \eta_k p(\theta_k) q(\theta_k) \end{aligned} \tag{3.3}$$

avec η_k des valeurs strictement positives appelées poids et S un ensemble de valeurs propres approchées de A de cardinalité $|\mathrm{S}| \geq m + 1$.

Cette définition est celle d'un produit scalaire non dégénéré lorsque tous les θ_k sont différents et que $|\mathrm{S}| \geq m + 1$. Néanmoins, les différents tests ont montré que prendre $|\mathrm{S}| = m$ fonctionne mieux que prendre $|\mathrm{S}| = m + 1$.

Nous voyons que le produit scalaire sur K_{m+1} qui en découle est beaucoup moins onéreux que le produit scalaire euclidien, puisqu'il ne met en jeu que des vecteurs de dimension de l'ordre de m et nous faisons l'hypothèse que $m \ll n$ (voir chapitre 1). Le produit scalaire euclidien nécessite $2n - 1$ opérations et surtout une communication globale, tandis que le nouveau produit scalaire, que nous appellerons « discret », nécessite $2m - 1$ opérations et aucune communication.

Sur machines parallèles, lorsque les vecteurs de dimension n sont distribués sur les p processeurs de telle sorte que chacun d'entre eux ait n_i éléments d'un vecteur de dimension n pour le processeur i (voir chapitre 1), le produit scalaire euclidien se décompose en :

- un produit scalaire local pour chaque processeur i, faisant intervenir $2n_i - 1$ opérations, que nous effectuons au travers d'un appel au **BLAS1**,

- une réduction diffusion du résultat local c'est-à-dire un scalaire qui, lorsqu'elle est optimisée sur la machine parallèle utilisée, s'effectue en $log_2(p)$, p étant le nombre de processeurs.

La dimension m du sous-espace de Krylov, étant supposée très petite devant n, nous effectuons chaque produit scalaire discret de façon redondante sur chaque processeur. Chacun effectue donc $2m-1$ opérations et aucune communication n'est nécessaire. Ainsi ce sont les communications qui constituaient notre perte de temps, car elles se faisaient en séquence, induisant des synchronisations et que nous évitons.

Une fois le produit scalaire polynomial défini, nous construisons alors la base $(w_i)_{1\leq i\leq m}$ du sous-espace de Krylov $\mathrm{K}_m(A, r_0)$, orthonormale pour le produit scalaire discret $(\cdot,\cdot)_d$ issu de ce produit scalaire polynomial.

3.2.2 Construction de la matrice W_{m+1}

Chacun des vecteurs w_j satisfait la relation $w_j = p_{j-1}(A)r_0$ pour $j = 1,\ldots,m+1$ où les polynômes p_j pour $j = 0,\ldots,m$ sont de degré respectif j et forment une base orthonormale de polynômes pour le produit scalaire (3.3). Ils sont construits ainsi :

$p_0 = 1/\alpha$ tel que $\langle p_0, p_0\rangle = 1$
Pour $j = 1,\ldots,m$,
  $q = tp_{j-1}(t)$
  Pour $i = 1,\ldots,j$
    $h_{i,j} = \langle q, p_{i-1}\rangle$
    $q = q - h_{i,j}p_{i-1}(t)$
  Fin Pour
  $h_{j+1,j} = \sqrt{\langle q, q\rangle}$
  $p_j = q/h_{j+1,j}$
Fin Pour.

Les polynômes p_j satisfont donc la récurrence :

$$h_{j+1,j}p_j(t) = tp_{j-1}(t) - \sum_{i=0}^{j-1} h_{i+1,j}p_i(t) \quad \text{pour} \quad j = 1,\ldots,m. \tag{3.4}$$

Comme les poids η_k et les nœuds θ_k sont réels, la relation précédente se simplifie en une relation à trois termes. En effet, nous avons :

$$\begin{aligned} \langle p, tq\rangle &= \sum_{\theta_k\in S} \eta_k p(\theta_k)(\theta_k q(\theta_k)) \\ &= \sum_{\theta_k\in S} \eta_k(\theta_k p(\theta_k))q(\theta_k) \\ &= \langle tp, q\rangle. \end{aligned} \tag{3.5}$$

Les coefficients $h_{i+1,j}$ pour $i = 0,\ldots j-1$ sont déduits de la condition d'orthonormalité des polynômes p_j et de la récurrence (3.4). Du fait que le produit scalaire pris en compte

vérifie la condition (3.5), ils vérifient :

$$\begin{aligned} h_{i+1,j} &= \frac{\langle tp_{j-1}(t), p_i(t)\rangle}{\langle p_i(t), p_i(t)\rangle} \\ &= \langle p_{j-1}(t), tp_i(t)\rangle \\ &= \langle p_{j-1}(t), h_{i+2,i+1}p_{i+1}(t) + \textstyle\sum_{k=0}^{i} h_{k+1,i+1}p_k(t)\rangle. \end{aligned}$$

Le polynôme p_{j-1} étant orthogonal aux polynômes p_k pour $k = 0, \ldots, j-2$, les coefficients $h_{i+1,j} = 0$ pour $i = 0, \ldots, j-3$. Nous avons en outre :

$$h_{j+1,j} = \langle tp_{j-1}, p_j\rangle = \langle tp_j, p_{j-1}\rangle = h_{j,j+1}. \tag{3.6}$$

En notant $\beta_{j+1} = h_{j+1,j}$ et $\alpha_j = h_{j,j}$, l'équation (3.4) se simplifie en :

$$\beta_{j+1}p_j(t) = tp_{j-1}(t) - \alpha_j p_{j-1}(t) - \beta_j p_{j-2}(t), \qquad j = 1, \ldots, m \tag{3.7}$$

avec comme convention $p_{-1} \equiv 0$, $\beta_0 = 0$ et $p_0 \equiv 1/\alpha$ tel que $\|p_0\| = 1$ où $\|\cdot\|$ est la norme induite par le produit scalaire $\langle\cdot,\cdot\rangle$. Les polynômes sont calculés à l'aide de la procédure de Stieltjes [21] suivante :

Algorithme 3.1 *Procédure de Stieltjes pour produit scalaire défini*

Initialisation : $p_1(t) = 0$, $p_0(t) = 1/\alpha$, *avec* $\alpha = \sqrt{\langle 1, 1\rangle}$
Pour $j = 1, \ldots, m$ *Faire*
  $q = tp_{j-1} - \beta_j p_{j-2}$
  $\alpha_j = \langle q, p_{j-1}\rangle$
  $q = q - \alpha_j p_{j-1}$
  $\beta_{j+1} = \sqrt{\langle q, q\rangle}$. *Si* $\beta_{j+1} = 0$ *alors arrêt*
  $p_j = q/\beta_{j+1}$
Fin Pour.

Les vecteurs w_j à leur tour, vérifient alors la même récurrence :

$$\begin{aligned} w_1 &= p_0(A)r_0 \\ \beta_{j+1}w_{j+1} &= Aw_j - \alpha_j w_j - \beta_j w_{j-1}, \quad j = 1, \ldots, m \end{aligned} \tag{3.8}$$

avec la convention que $w_0 = 0$.

Matriciellement cette dernière équation s'écrit :

$$AW_m = W_{m+1}\bar{T}_m \tag{3.9}$$

où la matrice $\bar{T}_m$ de dimension $(m+1) \times m$ est définie ainsi :

$$\bar{T}_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & \beta_m & \alpha_m \\ & & & & \beta_{m+1} \end{pmatrix}.$$

Nous voyons alors que lorsque la matrice A est normale ($A^H A = AA^H$), le calcul du produit scalaire discret $\langle p_i, p_j \rangle$ est une bonne approximation du produit scalaire (w_{i+1}, w_{j+1}), cette fois-ci euclidien, pour des poids η_k bien choisis. En effet, lorsque A est normale, elle admet une base orthonormale de vecteurs propres. Nous notons u_i ces vecteurs propres, pour $i = 1, \ldots, n$ et λ_i leur valeur propre correspondante. Puisque nous considérons que A est une matrice à spectre réel, elle admet une base orthonormale de vecteurs propres réels. Si nous exprimons r_0 dans cette base, nous obtenons :

$$r_0 = \sum_{k=1}^{n} \xi_k u_k$$

$$p_i(A) r_0 = \sum_{k=1}^{n} p_i(\lambda_k) \xi_k u_k$$

et

$$(w_i, w_j) = \sum_{k=1}^{n} \xi_k^2 \, p_i(\lambda_k) p_j(\lambda_k)$$

avec $(\cdot, \cdot)$ le produit scalaire euclidien.

Lorsque S est égal au spectre de A et que $\eta_k = \xi_k^2$, nous avons alors égalité entre le produit scalaire discret et le produit scalaire euclidien.

Supposons à présent que S ait l éléments distincts θ_k pour $k = 1, \ldots, l$. Alors les racines du polynôme p_l sont exactement les valeurs θ_k, pour $k = 1, \ldots, l$. Lorsque $i > l$, les polynômes orthogonaux p_i sont sous-déterminés. Tout polynôme orthogonal p_i pour $\langle \cdot, \cdot \rangle$ est de la forme :

$$p_i(t) = q_{i-l}(t) p_l(t) \;\forall\; i > l \tag{3.10}$$

où q_{i-l} est un polynôme quelconque de degré $i - l > 0$. Le polynôme p_i vérifie $\langle p_i, p_i \rangle = 0$ pour $i \geq l$.

Enfin, lorsqu'une valeur θ_k est présente plusieurs fois dans l'ensemble S, nous pouvons la remplacer par une seule valeur dont le poids correspondant est la somme des poids de toutes ses instances. Comme nous ne voulons pas que $\|p_k\|$, au sens de la norme induite par le produit scalaire polynomial, soit égale à zéro pour $k = 0, \ldots, m$, nous devons choisir S tel que $|S| \geq m+1$ où S contient au moins $m+1$ valeurs θ_k distinctes. Cependant, dans les différents tests effectués, il s'est avéré que choisir $|S|$ égal à m donne de bons résultats, souvent meilleurs que pour $|S|$ égal à $m+1$, ou encore plus grand. Utiliser $|S| = m$ n'est pas gênant, car nous n'utilisons pas explicitement le vecteur $w_{m+1} = p_m(A) r_0$.

Nous calculons à présent la solution approchée.

3.2.3 Calcul de la solution approchée

En vue de calculer la solution approchée, les vecteurs w_i pour $i = 1, \ldots, m+1$ sont normalisés au sens de la norme 2. En effet, lorsque la taille du sous-espace grandit, la norme

des vecteurs w_i devient de plus en plus grande. Effectuer les calculs avec de tels vecteurs induit des erreurs qui peuvent être évitées en les normalisant. Ainsi, nous introduisons la matrice $D_{m+1} = \text{Diag}(\|w_1\|_2, \ldots, \|w_{m+1}\|_2)$ et calculons la matrice $\tilde{W}_{m+1} = W_{m+1}D_{m+1}^{-1}$ dont les colonnes forment une base de vecteurs normés au sens de la norme 2. Mathématiquement, projeter la solution approchée sur W_m ou $\tilde{W}_m$ est équivalent. La solution approchée calculée est la même, puisque les matrices engendrent le même sous-espace. Ce n'est cependant pas équivalent en arithmétique finie, où la norme, parfois trop grande, des derniers vecteurs de la base w_i peuvent occasionner des erreurs d'arrondis.

Nous calculons à présent les solutions **PO** et **PM** pour le produit scalaire $(\cdot,\cdot)_d$. Nous avons :

$$x_m = x_0 + W_m\overline{y}_m = x_0 + \tilde{W}_m\tilde{y}_m \tag{3.11}$$

avec $\overline{y}_m = D_m^{-1}\tilde{y}_m$. Le résidu s'exprime alors ainsi :

$$\begin{aligned} r_m &= r_0 - AW_m\overline{y}_m \\ &= r_0 - AW_mD_m^{-1}\tilde{y}_m \\ &= r_0 - W_{m+1}\bar{T}_mD_m^{-1}\tilde{y}_m \\ &= r_0 - \tilde{W}_{m+1}D_{m+1}\bar{T}_mD_m^{-1}\tilde{y}_m \end{aligned} \tag{3.12}$$

Nous notons $\tilde{S}_m$ la matrice $D_{m+1}\bar{T}_mD_m^{-1}$ et avons la relation $r_0 = \alpha w_1$, avec α tel que $p_0(t) = 1/\alpha$.

La solution **PO** impose que le résidu soit orthogonal à $\mathrm{K}_m(A, r_0)$ au sens du produit scalaire $(\cdot,\cdot)_d$, soit :

$$(r_m^{PO}, w_i)_d = 0 \text{ pour } i = 1, \ldots, m$$

soit encore

$$(\alpha w_1 - W_{m+1}\bar{T}_m\overline{y}_m^{PO}, w_i)_d = 0 \quad \text{pour } i = 1, \ldots, m, \tag{3.13}$$

ce qui est équivalent à résoudre

$$\langle \alpha p_0 - [p_0, \ldots, p_m]\bar{T}_m\overline{y}_m^{PO}, p_i)_d = 0 \quad \text{pour } i = 0, \ldots, m-1. \tag{3.14}$$

Les vecteurs w_i étant deux à deux orthogonaux pour le produit scalaire discret, l'équation (3.13) est équivalente à la résolution du système suivant :

$$\left\{\begin{array}{rcl} ((\alpha_1(\overline{y}_m^{PO})_1 + \beta_2(\overline{y}_m^{PO})_2)w_1, w_1)_d &=& (\alpha w_1, w_1)_d \\ ((\beta_2(\overline{y}_m^{PO})_1 + \alpha_2(\overline{y}_m^{PO})_2 + \beta_3(\overline{y}_m^{PO})_3)w_2, w_2)_d &=& 0 \\ \vdots &=& \vdots \\ ((\beta_m(\overline{y}_m^{PO})_{m-1} + \alpha_m(\overline{y}_m^{PO})_m)w_m, w_m)_d &=& 0. \end{array}\right.$$

De la même façon, les polynômes p_i étant deux à deux orthogonaux pour le produit scalaire polynomial, l'équation (3.14) se simplifie en :

$$\left\{\begin{array}{rcl} \langle(\alpha_1(\overline{y}_m^{PO})_1 + \beta_2(\overline{y}_m^{PO})_2)p_1, p_1\rangle &=& \langle \alpha p_1, p_1\rangle \\ \langle\beta_2(\overline{y}_m^{PO})_1 + \alpha_2(\overline{y}_m^{PO})_2 + \beta_3(\overline{y}_m^{PO})_3)p_2, p_2\rangle &=& 0 \\ \vdots &=& \vdots \\ \langle(\beta_m(\overline{y}_m^{PO})_{m-1} + \alpha_m(\overline{y}_m^{PO})_m)p_m, p_m\rangle &=& 0. \end{array}\right.$$

Ces deux systèmes se simplifient en le système suivant en tenant compte du fait que les vecteurs w_i et les polynômes p_i sont normés pour le produit scalaire discret et polynomial respectivement. Nous obtenons :

$$\left\{\begin{array}{rcl} \alpha_1(\overline{y}_m^{PO})_1 + \beta_2(\overline{y}_m^{PO})_2 & = & \alpha \\ \beta_2(\overline{y}_m^{PO})_1 + \alpha_2(\overline{y}_m^{PO})_2 + \beta_3(\overline{y}_m^{PO})_3 & = & 0 \\ \vdots & = & \vdots \\ \beta_m(\overline{y}_m^{PO})_{m-1} + \alpha_m(\overline{y}_m^{PO})_m & = & 0 \end{array}\right.$$

Cette dernière équation n'est rien d'autre que la représentation sous forme d'équations de la résolution matricielle suivante :

$$T_m \overline{y}_m^{PO} = \alpha e_1^{(m)}.$$

Cette équation se retrouve aussi par le biais suivant : le résidu r_m^{PO} est également orthogonal au sous-espace K_m au sens du produit scalaire $(\cdot,\cdot)_{(W_{m+1}^{\dagger})^T W_{m+1}^{\dagger}}$, ce qui s'exprime ainsi pour $i = 1, \ldots, m$

$$\begin{array}{ll} & (r_m^{PO}, w_i)_{(W_{m+1}^{\dagger})^T W_{m+1}^{\dagger}} = 0 \\ \Longleftrightarrow & (W_{m+1}(\alpha e_1^{(m+1)} - \bar{T}_m \overline{y}_m^{PO}), W_{m+1} e_i^{(m+1)})_{(W_{m+1}^{\dagger})^T W_{m+1}^{\dagger}} = 0 \\ \Longleftrightarrow & (\alpha e_1^{(m+1)} - \bar{T}_m \overline{y}_m^{PO}, e_i^{(m+1)})_2 = 0 \end{array}$$

Ceci est équivalent à $T_m \overline{y}_m^{PO} = \alpha e_1^{(m)}$.

De façcon pratique, nous n'utilisons pas la matrice W_{m+1} mais $\tilde{W}_{m+1}$ et calculons en fait $\tilde{y}_m^{PO}$ solution de

$$\tilde{S}_m \tilde{y}_m^{PO} = \|w_1\|_2 \alpha e_1^{(m)}, \tag{3.15}$$

et mettons à jour la solution avec $x_m^{PO} = x_0 + \tilde{W}_m \tilde{y}_m^{PO}$.

La solution de type **PM** se calcule en imposant que le résidu soit orthogonal au sous-espace $A\mathrm{K}_m$ au sens de la norme discrète, soit

$$(r_m^{PM}, Aw_i)_d = 0 \qquad \text{pour } i = 1, \ldots, m$$

ce qui est équivalent à calculer $\overline{y}_m^{PM}$ solution de

$$\begin{array}{rcl} \overline{y}_m^{PM} & = & \arg\min\limits_{y \in \mathbb{R}^m} \|W_{m+1}(\alpha e_1^{(m+1)} - \bar{T}_m y)\|_d \\ & = & \arg\min\limits_{y \in \mathbb{R}^m} \|[p_0, \ldots, p_m](\alpha e_1^{(m+1)} - \bar{T}_m y)\| \\ & = & \arg\min\limits_{y \in \mathbb{R}^m} \|\alpha e_1^{(m+1)} - \bar{T}_m y\|_2 \\ & = & \arg\min\limits_{y \in \mathbb{R}^m} \|r_0 - W_{m+1} \bar{T}_m y\|_{(W_{m+1}{}^{\dagger})^T W_{m+1}^{\dagger}}. \end{array}$$

De façon pratique, nous calculons $\tilde{y}_m^{PM}$ au lieu de $\bar{y}_m^{PM}$ avec l'équation suivante :

$$\tilde{y}_m^{PM} = \arg\min_{y \in \mathbb{R}^m} \left\| \alpha \|w_1\|_2 \, e_1^{(m+1)} - \tilde{S}_m y) \right\|_2,$$

et mettons à jour la solution approchée $x_m^{PM} = x_0 + \tilde{W}_m \tilde{y}_m^{PM}$.

3.2.4 Mise à jour des θ_k et η_k

Une fois que nous avons calculé la solution approchée, nous devons mettre à jour l'ensemble S contenant les valeurs θ_k ainsi que les poids η_k de façon à pouvoir répéter le procédé. Initialement, les valeurs θ_k sont prises aléatoirement dans l'intervalle $[\tilde{\lambda}_{min}, \tilde{\lambda}_{max}]$, où $\tilde{\lambda}_{min}$ et $\tilde{\lambda}_{max}$ sont les approximations respectives de la plus petite et la plus grande valeur propre de A fournies par le théorème de Gershgorin. Dans les processus[4]suivants, les θ_k prennent pour valeurs celles des valeurs propres du problème généralisé suivant :

$$\tilde{W}_m^T A \tilde{W}_m \, u = \lambda \, \tilde{W}_m^T \tilde{W}_m \, u. \tag{3.16}$$

Ceci constitue une projection du problème $Au = \lambda u$ sur le sous-espace engendré par $\tilde{W}_m$ orthogonalement à ce même sous-espace.

Cette opération est coûteuse car elle nécessite de former la matrice $\tilde{W}_m^T \tilde{W}_m$ qui demande $m(m-1)/2$ produits scalaires. Une fois cette matrice formée, la matrice $\tilde{W}_m^T A \tilde{W}_m$ est facilement calculable à partir de la relation :

$$\tilde{W}_m^T A \tilde{W}_m = \tilde{W}_m^T \tilde{W}_{m+1} \tilde{S}_m \tag{3.17}$$

Le problème aux valeurs propres généralisé (3.16) permet d'approcher m valeurs propres de la matrice A, parmi lesquelles m' sont distinctes. Nous pouvons alors soit mettre à jour m' valeurs de l'ensemble S ou bien les lui ajouter. Cependant, la dimension de S ne doit pas devenir trop grande, puisque nous voulons que notre produit scalaire ne soit pas onéreux. Les tests que nous avons faits pour des choix de S différents ont montré que la convergence n'est pas accélérée en prenant S de dimension plus grande. La procédure peut même être ralentie.

Nous devons former la matrice $\tilde{W}_m^T \tilde{W}_{m+1}$ de dimension $m \times (m+1)$. Puisque sa partie carrée de dimension $m \times m$ est symétrique et que la diagonale est constituée de 1, seule sa partie triangulaire supérieure doit être calculée. Former cette matrice est donc équivalent à effectuer $(m(m+1)/2)$ produits scalaires de dimension n chacun, ce qui est à peu près le nombre de produits scalaires qui doivent être effectués dans le processus d'Arnoldi pour la méthode **GMRES**(m). La différence est que les vecteurs qui doivent être orthogonalisés sont maintenant connus à l'avance. Ainsi chaque processeur i effectue les produits $\tilde{w}_i^T \tilde{w}_j$ pour $j = 2, \ldots m$ et $i \leq j$ sur ses données locales, à savoir sur les n_i éléments dont dispose le processeur i. Ces opérations sont suivies d'une réduction diffusion sur un vecteur de dimension $m(m+1)/2$. Nous effectuons donc localement le même nombre d'opérations, soit $\frac{m(m+1)}{2}(2n_i - 1)$. Mais ces opérations peuvent être effectuées en utilisant des **BLAS** de niveau 2 ou 3, alors que les méthodes **FOM**(m) et **GMRES**(m) ne peuvent utiliser que des **BLAS1** dans le cas de **MGS** et des **BLAS2** dans le cas de **CGS**. De plus, au lieu d'effectuer $m(m+1)/2$ réductions diffusions sur un scalaire, dans le cas de **MGS** et m réductions diffusions sur un vecteur de dimension j grandissante, dans le cas de **CGS**, nous n'effectuons qu'une seule réduction diffusion. Nous diminuons donc les synchronisations globales, mais aussi le coût des communications. En effet, nous n'envoyons plus qu'un seul

4. Nous appelons processus pour une méthode redémarrée dont la dimension du sous-espace de Krylov est m, l'ensemble de m itérations contigües entre deux redémarrages consécutifs

message dont la longueur totale est égale à la somme des longueurs de tous les messages envoyés par les versions **MGS** ou **CGS** du processus d'Arnoldi. Le temps d'envoi des messages reste identique, puisque le volume total des données est le même. Le gain en temps est dû à la réduction du temps passé dans la préparation des messages mais aussi dans la diminution du nombre de synchronisations globales. Le nombre de messages à préparer ainsi que de synchronisations passe de $m(m+1)/2$ pour **MGS** et m pour **CGS** à 1.

Nous avons vu au début de ce chapitre, que l'implantation du produit scalaire euclidien nécessitait l'utilisation soit des **BLAS1** (**CGS**), soit des **BLAS2** (**MGS**). Pour former la matrice $\tilde{W}_m\tilde{W}_{m+1}$, nous avons également le choix :

- d'effectuer m produits matrice vecteur $[\tilde{w}_{i+1},\ldots,\tilde{w}_m]^T w_i$ pour $i=1,\ldots,m$ utilisant les **BLAS2**, ce qui est mathématiquement équivalent à effectuer $m(m+1)/2$ produits scalaires,

- ou d'effectuer le produit matrice matrice $\tilde{W}_m^T\tilde{W}_{m+1}$ en utilisant les **BLAS3**. Ceci est mathématiquement équivalent à effectuer $m(m+1)$ produits scalaires, soit deux fois plus que dans le processus d'Arnoldi des méthodes **FOM**(m) et **GMRES**(m).

Il y a donc un compromis à trouver entre utiliser les **BLAS3** et les **BLAS2**, sachant que la première solution effectue deux fois plus d'opérations que la seconde, chacune étant effectuée, en moyenne, plus rapidement.

En ce qui concerne le calcul des poids η_k, nous avons suggéré précédemment que, dans le cas d'une matrice normale, un bon choix est $\eta_i = (r_{k+1}, z_i)^2$ où z_i est un vecteur propre de la matrice A. Puisque nous ne disposons pas des vecteurs propres, nous allons utiliser leur approximation grâce à la résolution du problème aux valeurs propres généralisé (3.16). Les vecteurs obtenus sont appelés des vecteurs de Ritz $\tilde{W}_m z_i$. En supposant qu'ils sont de norme euclidienne égale à 1, ils nous permettent de choisir $\eta_i = (r_m, \tilde{W}_m z_i)^2$. Nous avons :

$$\begin{aligned}(r_m, \tilde{W}_m z_i) &= (r_0 - A\tilde{W}_m\tilde{y}, \tilde{W}_m z_i)\\ &= (\ \alpha\|w_1\|\ e_1^{(m+1)} - \tilde{S}_m\tilde{y}, \tilde{W}_{m+1}^T\tilde{W}_m z_i).\end{aligned} \tag{3.18}$$

Dans le cas de la méthode **PO**, nous avons :

$$(r_m^{PO}, \tilde{W}_m z_i) = (-\tilde{s}_{m,m}(\tilde{y}_m^{PO})_m e_{m+1}^{(m+1)}, \tilde{W}_{m+1}^T\tilde{W}_m z_i) \tag{3.19}$$

avec $\tilde{s}_{i,j}$ les coefficients de la matrice $\tilde{S}_m$. Nous voyons que seule la dernière composante du vecteur $\tilde{W}_{m+1}^T\tilde{W}_m z_i$ est nécessaire. La matrice $\tilde{W}_{m+1}^T\tilde{W}_m$ étant déjà formée, le calcul des poids pour la méthode **PO** n'est pas onéreux.

Regardons à présent le cas de la solution **PM**. Si Q_m est la matrice produit de rotations de Givens (1.18), qui permet de transformer la matrice $\tilde{S}_m$ en la matrice triangulaire supérieure $\bar{R}_m$ de dimension $(m+1)\times m$ et le vecteur $\alpha\|w_1\|\ e_1^{(m+1)}$ en $\Gamma_{m+1}^{(m)} = (\gamma_1,\ldots,\tilde{\gamma}_{m+1})^T$ (1.19), alors $\tilde{y}^{PM} = R_m^{-1}(\gamma_1,\ldots,\gamma_m)^T$ où R_m est la partie supérieure de $\bar{R}_m$ de dimension

$m \times m$. Nous avons alors :

$$\begin{aligned}(r_m^{PM}, \tilde{W}_m z_i) &= (Q_m\ (\ \alpha\|w_1\|\ e_1^{(m+1)} - \tilde{S}_m \tilde{y}^{PM}), Q_m \tilde{W}_{m+1}^T \tilde{W}_m z_i) \\ &= (\Gamma_{m+1}^{(m)} - \bar{R}_m \tilde{y}^{PM}, Q_m \tilde{W}_{m+1}^T \tilde{W}_m z_i) \\ &= (\tilde{\gamma}_{m+1} e_{m+1}^{(m+1)}, Q_m \tilde{W}_{m+1}^T \tilde{W}_m z_i). \end{aligned} \tag{3.20}$$

Nous voyons donc que seule la dernière composante du vecteur $Q_m \tilde{W}_{m+1}^T \tilde{W}_m z_i$ nous intéresse. La matrice $\tilde{W}_{m+1}^T \tilde{W}_m$ étant déjà formée, le calcul des poids θ_k à l'aide de l'équation (3.20) est peu onéreux, comme c'est le cas pour la méthode **PO**.

Dans la seconde partie des tests que nous présenterons en section 3.6.2, les poids seront pris égaux à 1, car il est apparu que les résultats n'étaient pas forcément meilleurs, lorsque les poids, que nous venons de définir, étaient considérés. Les prendre tous égaux à 1 donnent souvent de meilleures performances.

3.2.5 Algorithme final

Nous appelons les méthodes présentées ci-dessus, respectivement **ADOPOR** et **ADOPMR** pour les méthodes **PO** et **PM**, ce qui signifie respectivement **A**cceleration by **D**iscrete **O**rthogonal **P**olynomial with an **O**rthogonal **R**esidual et **D**iscrete **O**rthogonal **P**olynomial with a **M**inimal **R**esidual.

L'algorithme final **ADOPMR** est résumé ci-dessous :

Algorithme 3.2 $[x_0, r_0]$ = *ADOPMR(A,x_0,b,m,iter,tol)*

1. *Initialisation :*
   (a) *Estimer* $\tilde{\lambda}_{min}$ *et* $\tilde{\lambda}_{max}$ *en utilisant le théorème de Gershgorin*
   (b) *Générer de façon aléatoire m valeurs contenues dans l'intervalle* $[\tilde{\lambda}_{min}, \tilde{\lambda}_{max}]$ *et formant l'ensemble initial S*
   (c) $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$
2. *Calculer les coefficients* α_i *et* β_i *à l'aide de la procédure de Stieltjes et former la matrice* $\bar{T}_m$
3. *Former la matrice* $W_m = [p_0(A)r_0, \ldots, p_{m-1}(A)r_0]$
4. *Redémarrage :*
   (a) *Calcul de la solution approchée :*
      - $\tilde{y} = \arg\min \|\beta e_1^{(m+1)} - \bar{T}_m y\|_2$
      - $x_0 = x_0 + W_m \tilde{y}$
      - $r_0 = r_0 - AW_m \tilde{y}$
   (b) $iter = iter + 1$
   (c) *Si* $(\|r_0\|_2/\beta < tol)$ *ou si* $(iter = maxit)$ *alors arrêt*

(d) Calcul des valeurs propres du système généralisé

$$W_m^T A W_m u = \lambda W_m^T W_m u$$

et mise à jour de S. Aller en 2.

L'algorithme **ADOPOR** est identique à l'algorithme précédent, excepté pour la ligne $\tilde{y} = \arg\min \|\beta e_1^{(m+1)} - \bar{T}_m y\|2$ de (4-a) qui est remplacée par la résolution de l'équation $T_m \tilde{y} = \beta e_1^{(m)}$.

Nous présentons dans la section suivante le cas des matrices à spectre complexe.

3.3 Cas d'une matrice à spectre complexe

Lorsque le spectre est complexe, la récurrence à trois termes des polynômes orthonormaux n'existe plus en général. Deux choix peuvent être faits. Le premier consiste à utiliser l'algorithme avec la récurrence résultante, qui est alors longue. La seconde alternative consiste à utiliser un produit scalaire indéfini de façon à pouvoir restituer la récurrence à trois termes. Nous mettons en évidence les avantages et les défauts de ces deux approches.

3.3.1 Produit scalaire hermitien

Lorsque le spectre est complexe, le produit scalaire symétrique (3.3) est remplacé par le produit scalaire hermitien suivant :

$$\begin{aligned} \langle \cdot, \cdot \rangle : \quad \mathbb{P}_{m+1} \times \mathbb{P}_{m+1} &\to \mathbb{C} \\ \langle p, q \rangle &= \sum_{\theta_k \in S} \eta_k p(\theta_k) \overline{q(\theta_k)} \end{aligned} \tag{3.21}$$

où $\mathbb{P}_{m+1}$ est l'ensemble des polynômes de degré inférieur ou égal à m à coefficients dans $\mathbb{C}$, la notation $\overline{\alpha}$ correspond à la valeur conjuguée de α et les coefficients η_k sont des réels strictement positifs.

La relation (3.5) n'est plus vérifiée. En effet, nous avons :

$$\langle p, tq \rangle = \langle \overline{t}p, q \rangle$$

et les polynômes p_j sont obtenus à l'aide de la longue récurrence (3.4). Nous devons à présent calculer les coefficients $(h_{i,j})_{1 \le i \le j+1}$ en imposant que le polynôme p_j soit orthogonal à tous les polynômes précédents avec comme produit scalaire le produit hermitien défini ci-dessus (3.21). La base de Krylov est construite à l'aide de la même récurrence soit :

$$h_{j+1,j} w_{j+1} = A w_j - \sum_{i=1}^{j} h_{i,j} w_i, \quad j = 1, \ldots, m \tag{3.22}$$

avec toujours pour convention que $w_1 = r_0/\alpha$ et $p_0(t) = 1/\alpha$ tel que $\|p_0\| = 1$.

À l'exception de cette différence et du fait que T_m est remplacée par une matrice de Hessenberg supérieure, les algorithmes résultant de type **PO** et **PM** restent inchangés.

Comme c'est le cas pour les matrices à spectre réel, les produits scalaires du processus d'Arnoldi sont remplacés par le calcul de $\tilde{W}_{m+1}^T \tilde{W}_{m+1}$, à savoir, soit par des produits matrice vecteur, soit par un produit matrice matrice et une seule communication, ce qui est en soi plus parallèle. Le défaut majeur des algorithmes résultants est qu'ils s'effectuent en arithmétique complexe, ce qui n'est pas le cas des méthodes **FOM**(m) et **GMRES**(m), lorsque la matrice utilisée est réelle. Même si les méthodes résultantes sont plus parallèles, le coût induit par les opérations complexes et leur stockage supplémentaire fait que ces méthodes ne sont pas compétitives par rapport aux méthodes **FOM**(m) et **GMRES**(m).

3.3.2 Produit scalaire indéfini

Afin de conserver la récurrence à trois termes, nous définissons le produit scalaire ainsi :

$$\begin{aligned} \langle \cdot, \cdot \rangle : \quad \mathbb{P}_{m+1} \times \mathbb{P}_{m+1} &\to \mathbb{C} \\ \langle p, q \rangle &= \sum_{\theta_k \in S} \eta_k p(\theta_k) q(\theta_k) \end{aligned} \tag{3.23}$$

où $\mathbb{P}_{m+1}$ est l'ensemble des polynômes complexes de degré inférieur ou égal à m et les coefficients η_k des poids réels strictement positifs. La principale différence avec le produit scalaire hermitien (3.21) est que cette forme bilinéaire ne définit plus vraiment un produit scalaire sur $\mathbb{P}_{m+1}$. En effet, la valeur $\langle p, p \rangle$ peut être négative et la norme issue de ce produit scalaire n'est pas, non plus, correctement définie.

Nous faisons l'hypothèse qu'un élément λ_k appartient à S s'il est réel ou si sa paire conjuguée $\lambda_i = \overline{\lambda_k}$ appartient également à S et est telle que son poids correspondant η_i est égal au poids η_k de λ_k. L'ensemble S est donc symétrique par rapport à l'axe des réels, ce qui s'avère être vrai si les éléments de S sont des valeurs propres d'une matrice réelle. Avec ces hypothèses, nous avons que $\langle p, q \rangle$ est réel pour des polynômes p et q à coefficients dans $\mathbb{R}$. Ainsi tous les calculs peuvent être effectués en arithmétique réelle. De plus, ayant choisi de prendre $\eta_k = |(r_m, \tilde{W}_m z_k)|^2$, nous avons, pour des valeurs de Ritz complexes conjuguées $\lambda_i = \overline{\lambda_k}$, $z_i = \overline{z_k}$, la propriété que $\eta_i = |(r_m, \tilde{W}_m z_i)|^2 = |(r_m, \overline{\tilde{W}_m z_k})|^2 = \eta_k$, r_m et $\tilde{W}_m$ étant réels.

La procédure de Stieltjes est modifiée pour prendre en compte le fait que la valeur $\langle p_i, p_i \rangle$ puisse être négative. Dans l'algorithme qui suit, nous supposons que $p_{-1} \equiv 0$ et $\beta_1 = 0$.

Algorithme 3.3 *Procédure de Stieltjes pour produit scalaire indéfini*

1 *Initialisation :* $p_0(t) = 1/\alpha$, *avec* $\alpha = \sqrt{|\langle 1, 1 \rangle|}$
2 *Pour* $j = 1, \ldots, m$ *Faire*
3 $\quad q = t p_{j-1} - \beta_j p_{j-2}$
4 $\quad \alpha_j = \langle q, p_{j-1} \rangle$
5 $\quad q = q - \alpha_j p_{j-1}$
6 $\quad \delta_{j+1} = \sqrt{|\langle q, q \rangle|}$. *Si* $\delta_{j+1} = 0$ *alors arrêt*
7 $\quad p_j = q/\delta_{j+1}$

8 $\quad \beta_{j+1} = \delta_{j+1}\frac{\langle p_j, p_j\rangle}{\langle p_{j-1}, p_{j-1}\rangle}$

9 *Fin Pour.*

Les polynômes sont normalisés de telle sorte que $\langle p_j, p_j\rangle = \pm 1$. Ainsi le scalaire β_{j+1} défini à la ligne 8 est égal à $\pm\delta_{j+1}$, ce qui nécessite seulement un ajustement de signe par rapport à δ_{j+1}. Nous obtenons alors une récurrence semblable au cas du spectre réel :

$$\delta_{j+1}p_j(t) = tp_{j-1}(t) - \alpha_j p_{j-1}(t) - \beta_j p_{j-2}(t). \tag{3.24}$$

Dans le cas particulier où S est un ensemble de réels, nous avons alors que $\langle q, q\rangle$ est toujours positif et $\beta_{j+1} = \delta_{j+1}$. Nous retrouvons donc la récurrence (3.7). Puisque le vecteur w_j est défini par $w_j = p_{j-1}(A)w_1$, la relation (3.9) est encore valable. Mais cette fois-ci $\bar{T}_m$ est de la forme :

$$\bar{T}_m = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \delta_2 & \alpha_2 & \beta_3 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \alpha_{m-1} & \beta_m \\ & & & & \delta_m & \alpha_m \\ & & & & & \delta_{m+1} \end{pmatrix}$$

où $\beta_i = \pm\delta_i$ avec $\delta_i > 0$ pour $i = 2, \ldots, m$.

À chaque pas de la procédure de Stieltjes, il y a une possibilité de panne à la ligne 6. Lorsqu'un tel cas se produit, le mieux est de redémarrer.

3.4 Comparaison des coûts de calcul, de communication et de stockage

Nous comparons les méthodes de type **ADOP** avec **FOM**(m), **GMRES**(m) et **TFQMR**. Cette dernière méthode fait également partie des tests présentées dans la section des résultats numériques.

Dans ce qui suit nous faisons l'hypothèse que $m \ll n$. Ceci induit que les méthodes **ADOPOR** et **ADOPMR** sont approximativement de même coût pour le stockage et la complexité, lorsque le produit scalaire utilisé est celui défini par (3.3) dans le cas des matrices à spectre réel et celui indéfini donné par (3.23), avec les hypothèses faîtes dans le paragraphe précédent, dans le cas des matrices à spectre complexe. Nous regroupons ces méthodes sous l'appellation **ADOP**. De la même façon les méthodes **ADOPOR** et **ADOPMR** utilisant le produit scalaire hermitien sont de même coût et nous les regroupons sous le nom **ADOPH**.

3.4.1 Coût de stockage

Les méthodes **ADOP**, **FOM**(m) et **GMRES**(m) sont à peu près de même coût. Les méthodes **ADOP** font cependant intervenir davantage de tableaux de dimension m qui

sont négligeables.

Les méthodes **ADOPH**, quant à elles, sont plus gourmandes, car elles utilisent une arithmétique complexe. Elles nécessitent donc deux fois plus de stockage que toutes les autres méthodes, l'une pour la partie réelle des données, l'autre pour leur partie imaginaire.

Le tout est résumé dans le tableau suivant :

Stockage	FOM/GMRES	ADOP	ADOPH	TFQMR
Vecteur de dimension n	$m+4$	$m+6$	$2(m+6)$	8

La méthode **TFQMR** (voir chapitre 1) n'est pas une méthode redémarrée. Son stockage est donc indépendant de m.

3.4.2 Complexité en temps et coût des communications

Nous comparons en premier lieu les méthodes **ADOP** avec **FOM**(m) et **GMRES**(m), puis regardons le coût des méthodes **ADOPH** et **TFQMR**. En ce qui concerne les méthodes redémarrées, nous détaillons les différentes opérations de ces méthodes pour un processus où m est la dimension du sous-espace de Krylov. En ce qui concerne la méthode **TFQMR**, nous exposons le coût d'une itération et non pas d'un processus, qui n'existe pas pour les méthodes non redémarrées.

Dans le calcul du coût des communications, nous fixons la latence à savoir le temps de la préparation d'un message sur le réseau à γ, que nous prenons constant, même si sa valeur peut varier en fonction de la taille du message envoyé. Nous notons δ le temps d'envoi sur le réseau d'une donnée, correspondant à un flottant, d'un processeur à un autre.

Pour les méthodes **ADOP** et **GMRES**(m), lors de la construction d'une base de Krylov de dimension m, les différentes opérations sont les suivantes :

ADOP	**FOM(m)/GMRES(m)**
1- Génération des matrices $\bar{T}_m$ et W_m	1- Génération de la matrice V_m à l'aide du processus d'Arnoldi
2- Calcul de $\tilde{y}_m$	2- Calcul de $\tilde{y}_m$
3- Mise à jour de S	

L'étape 2 nécessite le même nombre d'opérations et de communications à la fois pour les méthodes **ADOP**, **FOM**(m) et **GMRES**(m). L'étape 1 des méthodes **ADOP** nécessite :

1. $2m$ produits scalaires de dimension m pour le calcul de la matrice $\bar{T}_m$,

2. pour le calcul de la matrice W_m,
   (a) m produits matrice vecteur avec la matrice A,
   (b) multiplication de la matrice $\bar{T}_m$ par 2 matrices diagonales pour former la matrice $\tilde{S}_m$,
   (c) $2m-1$ opérations triadiques sur des vecteurs de dimension n,
   (d) $m+1$ divisions d'un vecteur de dimension n par un scalaire,
3. pour la résolution du problème aux valeurs propres généralisé,
   (a) 1 produit matrice matrice de dimensions mn par $n(m+1)$ pour former la matrice $W_m^T W_{m+1}$,
   (b) 1 produit matrice matrice de dimensions $m(m+1)$ par $(m+1)m$ pour le calcul de $W_m^T A W_m$ via (3.17),
   (c) la résolution du problème de valeurs propres (3.16) lui-même.

Comme nous faisons l'hypothèse que $m \ll n$, les étapes 1, 2-b, 3-b et 3-c sont négligeables par rapport au reste des calculs. Nous les effectuons de façon redondante sur chaque processeur. Elles n'induisent aucune communication entre les processeurs.

Pour les méthodes **FOM**(m) et **GMRES**(m), la construction de la matrice V_m nécessite :

1. $m(m+3)/2$ produits scalaires de dimension n,
2. m produits matrice vecteur avec la matrice A,
3. $m(m+1)/2$ opérations triadiques sur des vecteurs de dimension n,
4. $m+1$ divisions d'un vecteur de dimension n par un scalaire.

Les étapes 2 et 4 de **FOM**(m) et **GMRES**(m) nécessitent exactement le même nombre d'opérations et de communications que les étapes 2-a et 2-d des méthodes **ADOP**.

Les seules étapes qui nous restent à comparer sont donc les étapes 1 et 3 de **FOM**(m) et **GMRES**(m) avec celles 2-c et 3-a des méthodes **ADOP**. Si le calcul de la matrice $W_m^T W_{m+1}$ est fait à l'aide de m produits matrice vecteur, il est alors équivalent au calcul de $m(m+1)/2$ produits scalaires de dimension n. Il induit ensuite une seule communication de réduction diffusion dont le coût est $(\gamma+\delta m(m+1)/2)log_2(p)$. Dans le cas des méthodes **FOM**(m) et **GMRES**(m), si l'algorithme **MGS** est utilisé, il nécessite alors $m(m+1)/2$ réductions diffusions sur un scalaire, soit une complexité pour les communications égale à $\frac{m(m+1)}{2}(\gamma+\delta)log_2(p)$. Si nous utilisons, en revanche, l'algorithme **CGS**, nous avons alors une réduction diffusion sur un vecteur de longueur j, pour $j = 1, \ldots, m$, ce qui donne une complexité pour les communications égale à $(m\gamma + \delta m(m+1)/2)\log_2(p)$. Dans un cas comme dans l'autre, le nombre d'opérations est inférieur de l'équivalent de m produits scalaires par rapport au coût des méthodes **FOM**(m) et **GMRES**(m), ceci grâce à la

récurrence à trois termes. Le coût des communications est lui-aussi inférieur à celui des méthodes **FOM**(m) et **GMRES**(m) : il nous permet d'éviter toutes les préparations de messages. Dans ce calcul, il n'est pas pris en compte le gain effectué dû à la diminution du nombre de synchronisations, qui est aussi à l'avantage des méthodes **ADOP**. Si, à présent, le calcul de la matrice $W_m^T W_{m+1}$ est fait grâce à un produit matrice matrice faisant appel aux **BLAS3**, son coût est équivalent à $m(m+1)$ produits scalaires de dimension n. La communication qui s'en suit est de complexité égale à $(\gamma+\delta m(m+1))log_2(p)$. Les méthodes **ADOP** deviennent alors plus coûteuses de l'équivalent de $m(m-1)/2$ produits scalaires par rapport aux méthodes **FOM**(m) et **GMRES**(m), mais pas nécessairement en temps d'exécution, du fait que l'on utilise les **BLAS3** et que le coût des communications est inférieur : la donnée γ est en effet très supérieure en ordre de grandeur à celle de δ.

Les méthodes **ADOPH** effectuent exactement les mêmes étapes que les méthodes **ADOP**, mais elles mettent en jeu une arithmétique complexe et sont donc bien plus onéreuses. Ainsi, si nous considérons un scalaire complexe comme deux scalaires réels, le premier constituant la partie réelle et le second la partie imaginaire, le nombre d'opérations et le coût des communications pour les méthodes **ADOP** sont :

1. $2m$ produits scalaires de dimension m pour le calcul de la matrice $\bar{T}_m$ en arithmétique complexe, soit 4 fois plus d'opérations par rapport à l'équivalent en réel,

2. pour le calcul de la matrice W_m,

   (a) m produits matrice vecteur avec la matrice A, soit 2 fois plus d'opérations par rapport aux méthodes **ADOP**, la matrice A étant réelle et un coût des communications égal à $(\gamma+2\delta n)log_2(p)$, au lieu de $(\gamma+\delta n)log_2(p)$ dans le cas des matrices à spectre réel,

   (b) multiplication de la matrice $\bar{T}_m$ par 2 matrices diagonales pour former la matrice $\tilde{S}_m$, soit 2 fois plus d'opérations par rapport aux méthodes **ADOP**, les matrices diagonales étant réelles et la matrice $\bar{T}_m$ complexe,

   (c) $2m-1$ opérations triadiques sur des vecteurs de dimension n, soit 4 fois plus d'opérations par rapport aux méthodes **ADOP**, les vecteurs w_i étant complexes ainsi que les coefficients de la matrice $\bar{T}_m$,

   (d) $m+1$ divisions d'un vecteur de dimension n par un réel, soit 2 fois plus d'opérations par rapport aux méthodes, **ADOP**,

3. pour la résolution du problème aux valeurs propres généralisé,

   (a) 1 produit matrice matrice de dimensions mn par $n(m+1)$ pour former la matrice $W_m^T W_{m+1}$, soit 4 fois plus d'opérations par rapport aux méthodes **ADOP** et un coût des communications égal à $(\gamma+2\delta m(m+1))log_2(p)$ au lieu de $(\gamma+\delta m(m+1))log_2(p)$,

   (b) 1 produit matrice matrice de dimensions $m(m+1)$ par $(m+1)m$ pour le calcul de $W_m^T A W_m$ via (3.17), soit 4 fois plus d'opérations par rapport aux méthodes **ADOP**,

(c) la résolution du problème de valeurs propres (3.16) lui-même, soit 4 fois plus d'opérations par rapport aux méthodes **ADOP**, le problème généralisé à résoudre s'effectuant sur des matrices complexes.

Il s'avère que le nombre total d'opérations est entre 2 à 4 fois plus important que pour les méthodes **ADOP** et donc **FOM**(m) et **GMRES**(m) et que le volume des communications est 2 fois supérieur à celui des méthodes **ADOP**. Ces méthodes ne seront donc pas performantes d'un point de vue temps de calcul total. En revanche, utiliser une telle technique dans le cadre de matrices complexes serait intéressant car les méthodes **GMRES**(m) et **FOM**(m) s'effectuent alors en arithmétique complexe. Le gain des méthodes **ADOPH** par rapport aux méthodes **GMRES**(m) et **FOM**(m) pour une matrice complexe est du même ordre de grandeur que les méthodes **ADOP** par rapport aux méthodes **GMRES**(m) et **FOM**(m) dans le cas des matrices réelles.

La comparaison avec la méthode **TFQMR** est délicate car ce n'est pas une méthode redémarrée. Nous ne parlerons donc pas en terme de processus comme pour les méthodes précédentes, mais en terme d'itérations. Une itération de **TFQMR** induit les calculs suivants :

1. 3 produits scalaire sur des vecteurs de dimension n,
2. 3 produits matrice vecteur avec la matrice A,
3. 6 opérations triadiques sur des vecteurs de dimension n.

Pou un nombre de produits matrice vecteur égal à m, **TFQMR** nécessite m produits scalaires et $2m$ opérations triadiques, ce qui comparé aux méthodes redémarrées qui lors d'un processus effectuent m produits matrice vecteur avec la matrice A, est bien moins onéreux. Une faible complexité et un faible coût des communications sont les atouts majeurs des méthodes non redémarrées.

3.5 Utilisation avec préconditionnement

Il est possible de préconditionner les méthodes **ADOP** et les trois techniques de préconditionnement, à gauche, à droite, à gauche et à droite, peuvent être appliquées.

3.5.1 Préconditionnement à gauche

Si nous utilisons un préconditionnement à gauche, nous résolvons alors le système $M^{-1}Ax = M^{-1}b$ en construisant le sous-espace de Krylov $\mathrm{K}_m(M^{-1}A, r_0)$.

Nous approchons alors les valeurs propres de la matrice $M^{-1}A$. Comme nous l'avons vu dans le chapitre 2, la matrice M^{-1} n'est pas toujours disponible et seule son action sur un vecteur peut être calculée. Nous ne pouvons donc plus approcher les valeurs propres de la matrice à l'aide du théorème de Gershgorin. Nous pouvons utiliser une méthode du type méthode d'Arnoldi. Nous remplaçons alors les étapes 1-a et 1-b de l'initialisation de l'algorithme 3.2 par un processus d'une méthode de type Arnoldi avec comme dimension de sous-espace $|S|$.

La matrice T_m est alors construite ainsi que les vecteurs w_i à l'aide de l'équation :

$$\beta_{i+1} w_{i+1} = M^{-1} A w_i - \alpha_i w_i - \beta_i w_{i-1}, \quad i = 1, \ldots, m. \tag{3.25}$$

Cela induit la relation matricielle $M^{-1}AW_m = W_{m+1}\bar{T}_m$. La nouvelle solution approchée est $x_m = x_0 + W_m y_m$ et le résidu correspondant $r_m = r_0 - M^{-1}AW_m y_m$.

Finalement, le problème aux valeurs propres généralisés 2-e est remplacé par

$$\begin{aligned} W_m^T M^{-1} A W_m \, u &= W_m^T W_{m+1} \bar{T}_m \, u \\ &= \lambda \, W_m^T W_m \, u. \end{aligned} \tag{3.26}$$

Plus généralement, la différence entre les méthodes **ADOP** préconditionnées à gauche et celles qui ne le sont pas réside dans l'initialisation des valeurs propres approchées et l'application de $M^{-1}A$ au lieu de A à chaque fois que A apparaît dans l'algorithme non préconditionné.

3.5.2 Préconditionnement à droite

Si nous utilisons un préconditionnement à droite, nous construisons le sous-espace de Krylov $\mathrm{K}_m(AM^{-1}, r_0)$ et recherchons la solution approchée x telle que $Mx = u$ et $AM^{-1}u = b$. L'ensemble S est initialisé avec $|S|$ valeurs propres approchées de AM^{-1} en utilisant par exemple une méthode de type Arnoldi. La matrice W_m satisfait $AM^{-1}W_m = W_{m+1}\bar{T}_m$. Le vecteur u n'est jamais calculé et la solution approchée est mise à jour par $x_m = x_0 + M^{-1}W_m y_m$. Le résidu correspondant, quant à lui, est mis à jour par $r_m = r_0 - AM^{-1}W_m y_m$.

Rappelons que l'algorithme **GMRES**(m) donne lieu à une version dite flexible [47], que nous avons exposée au chapitre 2 dans l'algorithme 2.14, qui permet de changer de préconditionnement à chaque fois qu'un nouveau vecteur de la base est construit. Ce n'est pas possible avec les méthodes de type **ADOP**, qui approchent les valeurs propres de la matrice AM^{-1}, laquelle doit donc rester la même à chaque pas.

3.5.3 Préconditionnement à gauche et à droite

Ce préconditionnement prend simplement en compte les deux formes précédentes en résolvant $L^{-1}AU^{-1}u = U^{-1}b$ avec $Ux = u$.

3.6 Résultats numériques

Tous les tests ont été effectués sur le CRAY T3E de l'I.D.R.I.S.[5] avec des matrices issues de la collection Harwell-Boeing [11], ainsi que de la collection SPARSKIT [44] disponible à l'Université du Minnesota. La dimension des matrices étudiées est comprise entre 886 et 21200, parmi lesquelles certaines sont très creuses. Nous avons effectué les tests pour un nombre de processeurs compris entre 1 et 8, la taille des matrices utilisées ne

5. Institut du Développement et des Ressources en Informatique Scientifique, Bâtiment 506 - B.P. 167, 91403 ORSAY CEDEX, FRANCE

nécessitant pas l'utilisation de davantage de processeurs. Le programme a tout d'abord été codé en F90, puis pour des raisons d'efficacité, nous avons transformé une grande partie de ce code en F77. Nous utilisons la bibliothèque de communications **MPI** [31].

Dans la première partie des tests, nous nous intéressons au comportement général des méthodes **ADOP** et utilisons des matrices de dimensions petites, pour lesquelles les hypothèses faites dans la section du calcul des coûts de stockage, des opérations et des communications, $m \ll n$ n'est pas vérifiée. Nous comparons les méthodes **ADOP**, qui utilisent le produit scalaire hermitien et indéfini avec les méthodes **FOM**(m), **GMRES**(m) et **QMR**. Nous utilisons la version **MGS** du processus d'Arnoldi pour les méthodes **FOM**(m) et **GMRES**(m). En ce qui concerne la méthode **QMR**, nous utilisons la version, qui évite les produits matrice vecteur avec la matrice A^T, appelée **T**ranspose **F**ree **QMR** (**TFQMR**) [18]. D'un point de vue accélération en temps d'exécution, il s'est avéré que les méthodes **ADOP** avec le produit scalaire indéfini sont bien plus rapides que celles utilisant le produit scalaire hermitien, comme nous l'avons remarqué dans le calcul de la complexité des méthodes. La seconde partie concerne des tests effectués pour comparer la méthode **ADOPMR** avec **GMRES**(m) et **TFQMR** à l'aide de la bibliothèque P_SPARSLIB [51, 53] sur des matrices de dimensions plus importantes. Les méthodes **ADOPOR** et **FOM**(m) ne se comportent pas bien numériquement et divergent pour la plupart des cas testés dans cette seconde partie. Nous ne présenterons pas leurs résultats. Dans cette seconde partie des tests, nous utilisons la version **CGS** du processus d'Arnoldi pour la méthode **GMRES**(m).

Pour tous les cas testés, le résidu des méthodes **ADOP**, **FOM**(m) et **GMRES**(m) ne sont jamais calculés par $r_m = b - Ax_m$, ce qui entraînerait un produit matrice vecteur supplémentaire, mais sont mis à jour à l'aide de l'équation

$$r_m = \tilde{\gamma}_{m+1}\Omega_{m+1}Q_m e_{m+1}^{(m+1)}, \tag{3.27}$$

où $\Omega_{m+1} = W_{m+1}$ pour les méthodes **ADOPMR** et V_{m+1}, pour la méthode **GMRES**(m) et

$$r_m = -\beta(y_m)_m \omega_{m+1} \tag{3.28}$$

où $\beta = t_{m+1,m}$ et $\omega_{m+1} = w_{m+1}$ pour les méthodes **ADOPOR** et respectivement $h_{m+1,m}$ et v_{m+1} pour la méthode **FOM**(m). En ce qui concerne la méthode **TFQMR**, le résidu n'est jamais calculé tout au long du procédé et sa norme est approchée par une borne supérieure [18]. Le résidu et sa norme ne sont donc calculés par la formule $r = b - Ax$ qu'à la toute dernière itération. Ainsi, si nous avions calculé la norme du résidu à chaque pas, nous aurions pu certainement arrêter le procédé un peu plus tôt, mais nous aurions perdu alors beaucoup de temps dans les produits matrice vecteur.

Dans les paragraphes suivants, les acronymes **ADOPOR** et **ADOPMR** sont attribués aux méthodes **ADOP** utilisant un produit scalaire discret indéfini avec des poids pris égaux à 1. Les acronymes **ADOPWOR** et **ADOPRWMR** concernent les méthodes **ADOP** avec le produit scalaire indéfini et les poids pris comme expliqués dans le paragraphe (3.2.4). L'acronyme **ADOPHMR** quant à lui correspond à la méthode

ADOPMR utilisant le produit scalaire hermitien. Nous ne considérons pas l'équivalent pour les méthodes **PO**.

Dans les tableaux de résultats, m indique la dimension du sous-espace de Krylov construit. Pour chaque test, le terme *Proc/Matvec* montre respectivement le nombre de redémarrages ou processus et le nombre de produits matrice vecteur effectués avant que l'algorithme ne s'arrête. Le terme *Temps* montre le temps total d'exécution en secondes, somme des temps passés dans la méthode (**ADOP**, **FOM**(m), **GMRES**(m) ou **TFQMR**), les produits matrice vecteur et les applications du préconditionnement, en excluant la phase de prétraitement. Le terme *Reduction* montre la réduction de la norme finale lorsque l'algorithme s'arrête. Celui-ci est arrêté dès que la réduction de la norme résiduelle est de 10^{-6} ou lorsqu'un maximum de 400 processus ont été effectués. La dernière ligne *Accel.Temps*, représente le temps d'exécution en secondes de la méthode elle-même, jusqu'à convergence ou qu'un nombre maximal de processus ait été atteint. Enfin, le terme *NbPE* représente le nombre de processeurs sur lesquels les tests ont été effectués.

3.6.1 Produits scalaires hermitien et indéfini

Dans ce paragraphe, nous voulons comparer le comportement général des méthodes **ADOP**, **ADOPW**, **ADOPH**, **FOM**(m), **GMRES**(m) et **TFQMR**. Cependant, la méthode **TFQMR** n'a pas été testée lorsqu'un préconditionnement était utilisé.

Les tests sont effectués sur des matrices de petites dimensions et l'hypothèse $m \ll n$ n'est pas vérifiée. Ces tests ont été effectués avec comme objectif d'analyser le comportement général des nouvelles méthodes **ADOP**. Des tests sur des matrices de dimensions plus grandes sont effectués dans la seconde partie.

La matrice A est répartie sur les processeurs de la machine par blocs de *ncol* colonnes consécutives (voir chapitre 1), *ncol* pouvant être différent pour chaque processeur et calculé de telle sorte que chaque processeur possède à peu près le même nombre d'éléments non nuls. Lorsque nous utilisons un préconditionnement, celui de Jacobi par blocs est appliqué à gauche. Chaque bloc correspond alors à la partie blocs diagonale de la matrice locale de chaque processeur. La résolution du système linéaire qu'exige le préconditionnement Jacobi par bloc est effectuée à l'aide de la factorisation $ILU(0)$ sur chacun des blocs. Nous ne connaissons pas la solution à l'avance et le second membre est pris égal au vecteur unité $(1, \ldots, 1)^T$. Le vecteur initial est le vecteur nul. Les différentes dimensions m de redémarrage choisies sont 11 valeurs dans l'intervalle $[5 - 50]$. La procédure est arrêtée dès que $\|r\|_2/\|b\|_2 < 1.0e-6$ ou lorsqu'un nombre maximum de processus égal à 400 a été atteint.

Les tests ont été effectués sur les matrices SHERMAN4, SHERMAN5, ORSIRR1, ORSIRR2, ORSREG1. Leur dimension est comprise entre 886 et 3312. La dimension de l'ensemble S de valeurs propres approchées de la matrice A est prise égale à m. Lors du tout premier processus, celui-ci n'est pas initialisé à l'aide du théorème de Gershgorin, mais avec une méthode de type Arnoldi. La version **MGS** est utilisée dans le processus

d'Arnoldi des méthodes **GMRES**(m) et **FOM**(m).

Dans les tableaux suivants, les cases en gras représentent le meilleur temps d'exécution obtenu pour la méthode et la matrice considérée, tandis que les cases en italique signifient qu'il y a eu un problème dans la convergence : en général, la méthode s'arrête alors que la norme réelle du résidu calculé $r_m = b - Ax_m$ ne satisfait pas le critère d'arrêt. Il y a en fait un décalage entre cette norme, qui est celle affichée et celle approchée par les équations (3.27) et (3.28), utilisée dans le critère d'arrêt.

Nous considérons tout d'abord la matrice ORSIRR1 de la collection Harwell-Boeing [11][6]. Elle est de dimension 1030 avec 6858 éléments non nuls. Nous considérons les méthodes de type **PM** au travers du tableau 3.1 :

1. le temps mis par les méthodes **ADOPWMR** et **ADOPMR** pour exécuter un processus est à peu près le même. Ceci est dû au fait que la méthode **ADOPWMR** ne demande que quelques opérations supplémentaires par rapport à **ADOPMR** pour la mise à jour des poids et que ces opérations sont négligeables,

2. d'un point de vue convergence et gain en temps, la méthode **ADOPWMR** n'est pas très performante par rapport aux méthodes **GMRES**(m) et **ADOPMR**. La méthode **ADOPMR** est très performante comparée à **ADOPWMR**,

3. un processus de la méthode **ADOPHMR** est bien plus coûteux qu'un processus de la méthode **ADOPMR** : elle met en jeu une arithmétique complexe, de coût 2 à 4 fois supérieur à **ADOPMR** et la récurrence dans la construction de la base de Krylov est plus longue que pour la méthode **ADOPHMR**,

4. le nombre de processus de la méthode **ADOPHMR** nécessaires pour converger diminue lorsque m augmente. La méthode ne peut cependant être perfomante du point de vue du temps, un processus étant beaucoup trop coûteux,

5. **ADOPMR** converge plus rapidement que **GMRES**(m) pour des dimensions de redémarrage petites, à savoir entre 5 et 30. Lorsque m est trop grand, la matrice W_m devient mal conditionnée et la méthode ne converge plus. Ceci est également vrai pour la méthode **ADOPWMR**. À l'inverse, la méthode **ADOPHMR** est plus fiable et converge encore pour des dimensions de redémarrage plus grandes,

6. un processus de la méthode **ADOPMR** est toujours plus rapide qu'un processus de **GMRES**(m). Le gain obtenu par **ADOPMR** par rapport à **GMRES**(m) pour un processus augmente juqu'à 33% quand m est de l'ordre de $30 - 40$, puis redescend, pour des valeurs de m plus élevées,

7. le meilleur temps obtenu parmi les tests effectués est réalisé pour la méthode **ADOPMR** avec une dimension de redémarrage égale à 20 avec 2321 produits matrice vecteur et 4.083 secondes et pour **GMRES**(m) avec une dimension de redémarrage égale à 50 avec 1623 produits matrice vecteur et 4.851 secondes,

6. voir également Matrix Market : `http://gams.cam.nist.gov/MatrixMarket/`

TAB. 3.1 – *Matrice ORSIRR1, NbPE = 4, Precon = non*

m	Perf.	GMRES(m)	ADOPMR(m)	ADOPWMR(m)	ADOPHMR(m)
5	Proc/Matvec	400 / 2001	400 / 2001	400 / 2001	400 / 2001
	Temps	3.162	3.121	3.091	4.906
	Reduction	0.674E+00	0.226E-01	0.460E+00	0.658E-02
	Accel. Temps	1.013	0.954	0.956	1.495
7	Proc/Matvec	400 / 2801	400 / 2801	400 / 2801	400 / 2801
	Temps	4.551	4.318	4.318	7.012
	Reduction	0.505E+00	0.233E-05	0.160E+00	0.125E-03
	Accel. Temps	1.570	1.319	1.315	2.243
9	Proc/Matvec	400 / 3601	344 / 3097	400 / 3601	**336 / 3025**
	Temps	6.058	4.742	5.590	**7.906**
	Reduction	0.405E+00	0.888E-06	0.219E-01	**0.672E-06**
	Accel. Temps	2.234	1.496	1.782	**2.632**
10	Proc/Matvec	400 / 4001	312 / 3221	400 / 4001	358 / 3581
	Temps	6.886	4.843	6.245	9.531
	Reduction	0.218E-01	0.144E-06	0.156E-01	0.927E-06
	Accel. Temps	2.611	1.539	2.003	3.297
15	Proc/Matvec	276 / 4141	205 / 3076	400 / 6001	194 / 2911
	Temps	7.786	5.106	9.962	8.387
	Reduction	0.880E-06	0.438E-06	0.666E-03	0.754E-06
	Accel. Temps	3.385	1.828	3.561	3.419
20	Proc/Matvec	364 / 7273	**116 / 2321**	**324 / 6481**	168 / 3361
	Temps	14.998	**4.083**	**11.662**	10.823
	Reduction	0.100E-05	**0.781E-06**	**0.949E-06**	0.580E-07
	Accel. Temps	7.189	**1.644**	**4.775**	4.978
30	Proc/Matvec	95 / 2839	191 / 5731	343 / 10291	88 / 2641
	Temps	6.775	10.996	19.956	10.166
	Reduction	0.998E-06	0.663E-06	0.623E-06	0.686E-06
	Accel. Temps	3.687	5.012	9.085	5.570
40	Proc/Matvec	56 / 2207	161 / 6441	400 / 16001	102 / 4081
	Temps	6.009	14.353	35.181	18.339
	Reduction	0.999E-06	0.400E-06	0.564E-05	0.615E-06
	Accel. Temps	3.636	7.490	18.253	11.324
50	Proc/Matvec	**33 / 1623**	236 / 11801	379 / 18951	110 / 5501
	Temps	**4.851**	31.181	49.182	29.393
	Reduction	**0.999E-06**	0.968E-06	0.525E-06	0.280E-07
	Accel. Temps	**3.080**	18.552	28.980	19.791

8. pour un temps de convergence similaire (4.85s), **ADOPMR** converge avec $m = 10$ en 3221 produits matrice vecteur, tandis que GMRES converge avec $m = 50$ et un nombre de produits matrice vecteur deux fois moins important (1623).

Le tableau 3.2 résume les résultats concernant les méthodes **PO** :

9. les temps d'exécution d'un processus des méthodes **ADOPWOR** et **ADOPOR** sont comparables. En effet, le calcul des poids dans la méthode **ADOPWOR** induit très peu d'opérations supplémentaires, qui sont négligeables,

10. contrairement aux méthodes **PM**, la méthode **ADOPWOR** est tout aussi performante que la méthode **ADOPOR** en ce qui concerne le nombre de processus pour converger et le temps total d'exécution,

11. un processus de la méthode **ADOPOR** est plus rapide en temps d'exécution qu'un processus de la méthode **FOM**(m) pour des valeurs de m supérieures à 7,

12. la méthode **ADOPOR** converge plus rapidement que la méthode **FOM**(m) pour des dimensions de redémarrage comprises entre 7 et 10. Au-delà, la matrice W_m devient mal conditionnée. La procédure s'arrête alors que la norme réelle du résidu calculé avec $r_m = b - Ax_m$ ne satisfait pas le critère d'arrêt. Il y a donc un décalage entre la norme calculée précédemment qui est celle utilisée pour l'affichage et celle calculée par l'équation (3.28) utilisée dans le critère d'arrêt. Il en est de même pour la méthode **ADOPWOR** dès que $m = 10$. La méthode divergeant sans équivoque, les étoiles dans le tableau 3.2 correspondent à des valeurs très élevées,

13. le meilleur temps obtenu parmi les tests effectués est réalisé pour la méthode **ADOPOR** avec une dimension de redémarrage égale à 9 en 2035 produits matrice vecteur et 3.146 secondes, et pour **FOM**(m) avec une dimension de redémarrage égale à 50 et 2386 produits matrice vecteur en 5.634 secondes.

Comme nous pouvons le voir avec le tableau 3.3, **TFQMR** s'est arrêté au bout de 4640 itérations, mais n'a en fait pas convergé. La borne supérieure calculée est en fait inférieure à 1.0E-6, mais ce n'est pas vrai pour la norme du résidu. Si nous laissons la méthode **TFQMR** calculer jusqu'à ce qu'elle ait atteint 15 000 itérations, la norme du résidu est stagne à la valeur 1.2E-5 et n'évolue plus. **TFQMR** ne converge pas.

Nous pouvons faire cette remarque finale : parmi tous les tests effectués, le meilleur temps global est obtenu avec la méthode **ADOPOR** pour m égal à 9 en 3.146 secondes.

Si nous considérons la même matrice préconditionnée à gauche avec un préconditionnement Jacobi par blocs, les remarques faîtes précédemment sont encore valables pour les méthodes **PM**.

La seule différence est que le meilleur temps pour **GMRES**(m) est obtenu avec $m = 30$, et pour **ADOPMR** avec $m = 10$. De plus, la méthode **ADOPMR** se comporte mieux pour des dimensions de redémarrage plus grandes, spécialement pour $m = 50$.

TAB. 3.2 – *Matrice ORSIRR1, NbPE = 4, Precon = non*

m	Perf.	FOM(m)	ADOPOR(m)	ADOPWOR(m)
5	Proc/Matvec	400 / 2001	400 / 2001	400 / 2001
	Temps	3.040	3.121	3.053
	Reduction	0.556E-01	0.132E-02	0.346E-00
	Accel. Temps	0.948	0.985	1.000
7	Proc/Matvec	400 / 2801	400 / 2801	400 / 2801
	Temps	4.283	4.237	4.282
	Reduction	0.603E-01	0.419E-01	0.205E-02
	Accel. Temps	1.422	1.378	1.412
9	Proc/Matvec	400 / 3601	**226 / 2035**	**251 / 2260**
	Temps	5.811	**3.146**	**3.501**
	Reduction	0.4244E-01	**0.826E-06**	**0.739E-06**
	Accel. Temps	2.039	**1.037**	**1.175**
10	Proc/Matvec	400 / 4001	239 / 2391	*163 / 1631*
	Temps	6.524	3.723	*2.534*
	Reduction	0.461E-02	0.318E-06	*0.155E-01*
	Accel. Temps	2.351	1.251	*0.868*
15	Proc/Matvec	400 / 6001	*188 / 2821*	*150 / 2251*
	Temps	10.548	*4.625*	*3.659*
	Reduction	0.683E-03	*0.195E-05*	*0.124E-05*
	Accel. Temps	4.255	*1.731*	*1.353*
20	Proc/Matvec	228 / 4556	*299 / 5981*	*400 / 8001*
	Temps	8.634	*10.205*	*****
	Reduction	0.995E-06	*0.114E-02*	*****
	Accel. Temps	3.812	*3.962*	*****
30	Proc/Matvec	127 / 3809	*400 / 12001*	*400 / 12001*
	Temps	8.247	*22.508*	*****
	Reduction	0.930E-06	*0.114E-01*	*****
	Accel. Temps	4.188	*10.032*	*****
40	Proc/Matvec	**60 / 2386**	*354 / 14161*	*400 / 16001*
	Temps	**5.634**	*29.338*	*****
	Reduction	**0.961E-06**	*0.672E-05*	*****
	Accel. Temps	**3.131**	*14.881*	*****

TAB. 3.3 – *Matrice ORSIRR1, NbPE = 4, Precon = no*

Perf.	TFQMR
Matvec	4640
Temps	4.747
Reduction	0.120E-04
Accel. Temps	0.889

TAB. 3.4 – *Matrice ORSIRR1, NbPE = 4, Precon = Jacobi par blocs*

m	Perf.	FOM(m)	ADOPOR(m)	ADOPWOR(m)
5	Proc/Matvec	101 / 503	139 / 696	130 / 651
	Temps	0.907	1.252	1.171
	Reduction	0.729E-06	0.963E-06	0.410E-06
	Accel. Temps	0.248	0.351	0.334
7	Proc/Matvec	150 / 1045	**44 / 309**	52 / 365
	Temps	1.938	**0.558**	0.656
	Reduction	0.976E-06	**0.736E-06**	0.328E-06
	Accel. Temps	0.566	**0.157**	0.188
9	Proc/Matvec	57 / 510	50 / 451	44 / 397
	Temps	0.971	0.824	0.719
	Reduction	0.951E-06	0.408E-06	0.428E-06
	Accel. Temps	0.300	0.236	0.209
10	Proc/Matvec	50 / 500	34 / 341	38 / 381
	Temps	0.957	0.622	0.692
	Reduction	0.432E-06	0.175E-06	0.381E-06
	Accel. Temps	0.303	0.180	0.207
15	Proc/Matvec	28 / 413	22 / 331	**21 / 316**
	Temps	0.835	0.628	**0.594**
	Reduction	0.671E-06	0.228E-06	**0.422E-06**
	Accel. Temps	0.300	0.205	**0.196**
20	Proc/Matvec	17 / 338	14 / 281	19 / 381
	Temps	0.734	0.570	0.771
	Reduction	0.740E-06	0.296E-06	0.217E-06
	Accel. Temps	0.289	0.206	0.287
30	Proc/Matvec	10 / 285	13 / 391	*27 / 811*
	Temps	0.685	0.892	*1.741*
	Reduction	0.854E-06	0.128E-06	*0.805E-05*
	Accel. Temps	0.312	0.386	*0.695*
40	Proc/Matvec	**6 / 235**	*71 / 2841*	*34 / 1361*
	Time	**0.621**	*6.680*	*3.220*
	Reduction	**0.958E-06**	*0.190E-02*	*0.456E-05*
	Accel. Temps	**0.313**	*3.024*	*1.472*

Le tableau 3.4 résume les résultats concernant les méthodes **PO**. Les points allant de 9 et 10 restent vrais. Nous avons en outre :

1. pour des dimensions de redémarrage comprises entre 7 et 20, la méthode **ADOPOR** converge plus rapidement en nombre de processus que la méthode **FOM**(m). Un processus **ADOPOR** étant plus rapide qu'un processus **FOM**(m), la méthode **ADOPOR** converge plus rapidement en terme de temps d'exécution. Le comportement de la méthode **ADOPWOR** n'est pas aussi bon que celui de **ADOPOR**, mais est meilleur que celui de **FOM**(m) pour m compris entre 7 et 15. La méthode **ADOPWOR** semble moins stable numériquement que **ADOPOR** lorsque la dimension de m augmente,

2. le meilleur temps d'exécution total pour la méthode **ADOPOR** est obtenu avec $m = 7$ et 309 produits matrice vecteur en 0.558 secondes, et pour la méthode **FOM**(m) avec $m = 40$ en 235 pròduits matrice vecteur et 0.621 secondes.

Nous avons finalement, que le meilleur temps total pour toutes les méthodes testées est réalisé par la méthode **ADOPOR** en 0.558 secondes pour $m = 7$.

Nous considérons à présent notre seconde matrice SHERMAN4 de la collection Harwell-Boeing. Elle a 3786 éléments non nuls et est de dimension 1104. Au regard de la table 3.5, les cinq premières observations de 1 à 5 faites précédemment pour la matrice ORSIRR1 non préconditionnée demeurent vraies pour les méthodes **PM**. De plus nous avons :

1. un processus de la méthode **ADOPMR** est toujours plus rapide qu'un processus de la méthode **GMRES**(m). Le gain obtenu en temps atteint 30% pour valeur maximale lorsque m augmente jusque 20, puis décroît,

2. le meilleur temps général est obtenu, pour la méthode **ADOPMR**, avec m égal à 15 en 0.609 secondes, et pour la méthode **GMRES**(m) avec m égal à 50 en 0.736 secondes,

3. le meilleur temps est obtenu par la méthode **ADOPMR** pour un nombre de produits matrice vecteur égal à 376, qui est plus important que les 247 produits matrice vecteur, pour lequel **GMRES**(m) obtient son meilleur temps d'exécution général.

En ce qui concerne les méthodes **PO**, les points $9 - 11$ exposés pour la matrice ORSIRR1 dans le cas du système non préconditionné demeurent. Nous avons en plus :

1. la méthode **ADOPOR** converge plus rapidement que **FOM**(m) pour m inférieur ou égal à 20. Il en est de même pour la méthode **ADOPWOR**. Au-delà, à la fois **ADOPOR** et **ADOPWOR** se comportent mal,

TAB. 3.5 – *Matrice SHERMAN4, NbPE = 4, Precon = non*

m	Perf.	GMRES(m)	ADOPMR(m)	ADOPWMR(m)	ADOPHMR(m)
5	Proc/Matvec	143 / 712	104 / 521	355/1776	136 / 681
	Temps	1.087	0.779	2.714	1.667
	Reduction	0.663E-06	0.941E-06	0.941E-06	0.565E-06
	Accel. Temps	0.361	0.250	0.876	0.527
8	Proc/Matvec	100 / 794	56 / 449	159 / 1273	83 / 665
	Temps	1.280	0.682	1.953	1.718
	Reduction	0.949E-06	0.996E-06	0.907E-06	0.979E-06
	Accel. Temps	0.471	0.216	0.632	0.573
9	Proc/Matvec	88/ 788	58 / 523	133 / 1198	56 / 505
	Temps	1.298	0.799	1.854	1.339
	Reduction	0.980E-06	0.866E-06	0.933E-06	0.132E-06
	Accel. Temps	0.492	0.257	0.604	0.455
10	Proc/Matvec	70 / 694	52 / 521	82 / 821	49 / 491
	Temps	1.163	0.803	1.280	1.326
	Reduction	0.963E-06	0.828E-06	0.868E-06	0.777E-06
	Accel. Temps	0.455	0.262	0.422	0.466
15	Proc/Matvec	54 / 802	**25 / 376**	44 / 661	**30 / 451**
	Temps	1.476	**0.609**	1.087	**1.313**
	Reduction	0.993E-06	**0.978E-06**	0.877E-06	**0.321E-06**
	Accel. Temps	0.649	**0.222**	0.406	**0.551**
20	Proc/Matvec	28 / 561	19 / 381	**29 / 581**	22 / 441
	Temps	1.134	0.658	**1.023**	1.423
	Reduction	0.970E-06	0.667E-06	**0.860E-06**	0.540E-06
	Accel. Temps	0.551	0.265	**0.426**	0.676
30	Proc/Matvec	14 / 421	19 / 571	46 / 1381	14 / 421
	Temps	0.985	1.148	2.750	1.644
	Reduction	0.976E-06	0.705E-06	0.677E-06	0.717E-06
	Accel. Temps	0.546	0.556	1.330	0.926
40	Proc/Matvec	7 / 276	20 / 801	41 / 1641	12 / 481
	Temps	0.956	1.805	3.702	2.280
	Reduction	0.985E-06	0.770E-06	0.874E-06	0.109E-06
	Accel. Temps	0.445	0.985	2.013	1.456
50	Proc/Matvec	**4 / 247**	16 / 801	22 / 1101	6 / 301
	Temps	**0.736**	2.162	3.088	1.619
	Reduction	**0.962E-06**	0.248E-06	0.908E-06	0.107E-06
	Accel. Temps	**0.478**	1.349	1.968	1.101

TAB. 3.6 – *Matrice SHERMAN4, NbPE = 4, Precon = non*

m	Perf.	FOM(m)	ADOPOR(m)	ADOPWOR(m)
5	Proc/Matvec	193 / 965	93 / 466	90 / 451
	Temps	1.461	0.707	0.677
	Reduction	0.866E-06	0.940E-06	0.648E-6
	Accel. Temps	0.469	0.230	0.226
8	Proc/Matvec	99 / 792	56 / 449	**49 / 393**
	Temps	1.260	0.694	**0.609**
	Reduction	0.901E-06	0.781E-06	**0.165E-6**
	Accel. Temps	0.442	0.234	**0.210**
9	Proc/Matvec	69 / 621	46 / 415	46 / 415
	Temps	1.006	0.640	0.641
	Reduction	0.937E-06	0.906E-06	0.906E-6
	Accel. Temps	0.363	0.217	0.221
10	Proc/Matvec	53 / 530	45 / 451	*44 / 441*
	Temps	0.872	0.694	*0.685*
	Reduction	0.842E-06	0.911E-06	*0.341E-04*
	Accel. Temps	0.323	0.241	*0.247*
15	Proc/Matvec	40 / 587	**24 / 361**	28 / 421
	Temps	1.037	**0.590**	0.680
	Reduction	0.991E-06	**0.681E-06**	0.808E-6
	Accel. Temps	0.429	**0.225**	0.263
20	Proc/Matvec	19 / 380	**17 / 341**	18 / 361
	Temps	0.722	**0.590**	0.626
	Reduction	0.532E-06	**0.44E-07**	0.409E-6
	Accel. Temps	0.333	**0.247**	0.263
30	Proc/Matvec	12 / 346	*56 / 1681*	*32 / 961*
	Temps	0.747	*3.129*	*1.803*
	Reduction	0.956E-06	*0.548E+00*	*0.127E+03*
	Accel. Temps	0.389	*1.451*	*0.858*
40	Proc/Matvec	7 / 279	11 / 441	*15 / 601*
	Temps	0.699	0.967	*1.287*
	Reduction	0.820E-06	0.891E-06	*0.549E-05*
	Accel. Temps	0.402	0.520	*0.698*
50	Proc/Matvec	**5 / 205**	*8 / 401*	*28 / 1401*
	Temps	**0.561**	*1.005*	*3.396*
	Reduction	**0.995E-0.6**	*0.270E-04*	*0.369E-04*
	Accel. Temps	**0.344**	*0.595*	*2.011*

TAB. 3.7 – *Matrice SHERMAN4, NbPE = 4, Precon = no.*

Perf.	TFQMR
Matvec	412
Temps	0.403
Reduction	0.198E-06
Accel. Temps	0.079

2. le meilleur temps d'exécution est réalisé pour **ADOPOR** en 0.590 secondes avec $m = 15$ et $m = 20$ en respectivement 361 et 341 produits matrice vecteur et pour **FOM**(m) en 0.561 secondes avec $m = 50$ en 205 produits matrice vecteur.

Si nous comparons les résultats des méthodes **ADOP**, **FOM**(m) et **GMRES**(m) avec **TFQMR** (voir tableau 3.7), nous voyons que **TFQMR** converge en un nombre de produits matrice vecteur (412) plus important que le nombre de produits matrice vecteur des autres méthodes et est cependant bien plus rapide. Ceci est dû à sa très faible complexité. Le meilleur temps est donc obtenu par la méthode **TFQMR**.

En préconditionnant la matrice SHERMAN4, il apparaît que la méthode **ADOPMR** converge mieux que **GMRES**(m) seulement pour des dimensions de redémarrage inférieures à 9, et que pour m égale à 15 et 30, **ADOPWMR** se comporte mieux que **ADOPMR**.

La méthode **ADOPMR** n'est pas très performante pour des dimensions de redémarrage comprises entre 20 et 50, où le nombre de processus de **GMRES**(m) pour converger décroît considérablement. Pour $m = 50$, le nombre de processus pour **GMRES**(m) est 2. C'est aussi le cas pour les méthodes **ADOPMR**, **ADOPWMR** et **ADOPHMR**. La différence est que les méthodes **ADOP** doivent finir l'ensemble du processus, à savoir les 50 vecteurs de la base de Krylov, même si les premiers auraient suffi pour converger. **GMRES**(m), quant à lui, effectue un processus complet, puis construit seulement 2 nouveaux vecteurs de la base de Krylov. D'un point de vue du temps d'exécution, **GMRES**(m) est bien plus rapide car il n'effectue pas les 48 autres produits matrice vecteur ni les préconditionnements correspondants.

Ce cas de convergence assez rapide de la méthode **ADOPMR** pour des dimensions de redémarrage assez élevées n'est pas très commun, parmi les tests effectués. Les tests montrent plutôt que, lorsque m est grand et que **GMRES**(m) converge en 1 ou 2 processus, la convergence de **ADOPMR** est lente. Une solution pour y remédier consiste à effectuer tout d'abord un processus de **GMRES**(m) et de continuer ensuite avec la méthode **ADOPMR**. Ce premier processus passé avec **GMRES**(m) nous permet en outre d'approcher des valeurs propres de la matrice. L'algorithme devient alors plus rapide et plus fiable: lorsque **GMRES**(m) converge très rapidement en 1 ou 2 processus, c'est aussi le cas pour la nouvelle méthode. Lorsque **ADOPMR** converge plus vite, la méthode ainsi formée convergera à la même vitesse. De façon pratique, cela ne nécessite pas de calculs supplémentaires, puisque dans le cas des méthodes **ADOP** préconditionnées, une méthode de type Arnoldi doit être utilisée pour approcher des valeurs propres de la

matrice. Ceci peut se faire avec **GMRES**(m).

En ce qui concerne les méthodes **PO**, nous avons toujours qu'un processus des méthodes **ADOPOR** et **ADOPWOR** est moins coûteux qu'un processus **FOM**(m). La différence avec les tests précédents est que les méthodes **ADOPOR** se comportent bien numériquement et convergent avec le même nombre de processus que la méthode **FOM**(m). Le temps d'exécution est donc amélioré. Nous faisons cependant la même remarque que pour les méthodes **PM**: lorsque la dimension du redémarrage est élevée, les méthodes **PO** convergent en 2 processus, mais **FOM**(m) n'est pas obligé d'aller au bout du dernier processus pour s'arrêter, ce qui lui permet de ne pas faire des produits matrice vecteur inutiles, comme c'est le cas pour **ADOPOR** et **ADOPWOR**.

De façon générale, les tests ont montré que:

1. **ADOPOR** se comporte bien,
2. **ADOPHMR** n'est pas performante d'un point de vue temps d'exécution,
3. **ADOPWMR** n'est pas performante d'un point de vue nombre de processus pour converger,
4. **ADOPWOR** ne se comporte pas mieux que **ADOPOR** et est plus sensible aux dimensions de redémarrage élevées.

Dans la section suivante, nous n'expérimentons donc plus les méthodes **PO**, ni les méthodes **ADOPHMR** et **ADOPWMR**.

3.6.2 ADOPMR et GMRES(m)

Nous comparons la méthode **ADOPMR** avec **GMRES**(m) et **TFQMR** sur des matrices de dimensions plus grandes et pour lesquelles l'hypothèse $m \ll n$ est vérifiée. Nous effectuons nos tests avec la bibliothèque P_SPARSLIB [51, 53], qui implante déjà la méthode **GMRES**(m) préconditionnée à droite et **FMGRES**(m). Celles-ci utilisent l'algorithme du **CGS** dans le processus d'Arnoldi.

Avant d'être distribuée sur les processeurs, la matrice est partitionnée à l'aide de l'algorithme *Recursive Spectral Bisection* [49]. Une fois de plus, lorsque les matrices sont préconditionnées, nous utilisons le préconditionnement Jacobi par blocs. Les blocs diagonaux alors pris en compte sont ceux de la matrice réordonnée, et ceux-ci sont approchés à l'aide de la factorisation $ILU(0)$. Comme dans la section précédente, **TFQMR** n'a pas été testé lorsque nous utilisions un préconditionnement.

Le vecteur solution est le vecteur $(1, \ldots, 1)^T$ et le vecteur initial le vecteur nul. Les différentes dimensions de redémarrage sont 11 valeurs dans l'intervalle $[5-50]$. Le procédé

TAB. 3.8 – *Matrice SHERMAN4, NbPE = 4, Precon = Jacobi par blocs*

m	Perf.	FOM(m)	ADOPOR(m)	ADOPWOR(m)
5	Proc/Matvec	44 / 220	43 / 216	43 / 216
	Temps	0.372	0.360	0.359
	Reduction	0.992E-06	0.974E-06	0.974E-06
	Accel. Temps	0.109	0.106	0.107
8	Proc/Matvec	24 / 193	22 / 177	28 / 225
	Temps	0.336	0.298	0.380
	Reduction	0.812E-06	0.186E-06	0.536E-06
	Accel. Temps	0.110	0.090	0.118
9	Proc/Matvec	18 / 162	17 / 154	**17 / 154**
	Temps	0.290	0.262	**0.258**
	Reduction	0.896E-06	0.661E-06	**0.697E-06**
	Accel. Temps	0.097	0.079	**0.080**
10	Proc/Matvec	20 / 198	17 / 171	**17 / 171**
	Temps	0.356	0.295	0.293
	Reduction	0.893E-06	0.69E-07	0.625E-06
	Accel. Temps	0.121	0.091	0.093
15	Proc/Matvec	12 / 167	11 / 166	11 / 166
	Temps	0.327	0.295	0.299
	Reduction	0.974E-06	0.84E-07	0.35E-07
	Accel. Temps	0.124	0.099	0.103
20	Proc/Matvec	6 / 117	6 / 121	8 / 161
	Temps	0.245	0.223	0.297
	Reduction	0.971E-06	0.167E-06	0.96E-07
	Accel. Temps	0.103	0.081	0.110
30	Proc/Matvec	3 / 86	**3 / 91**	**4 / 121**
	Temps	0.198	**0.181**	**0.251**
	Reduction	0.952E-06	**0.540E-06**	**0.819E-06**
	Accel. Temps	0.095	**0.075**	**0.108**
40	Proc/Matvec	2 / 76	3 / 121	5 / 201
	Temps	0.196	0.279	0.452
	Reduction	0.990E-06	0.7E-08	0.402E-06
	Accel. Temps	0.104	0.139	0.218
50	Proc/Matvec	**2 / 52**	2 / 101	**2 / 101**
	Temps	**0.149**	0.249	**0.250**
	Reduction	**0.966E-06**	0.4E-08	**0.4E-08**
	Accel. Temps	**0.087**	0.131	**0.132**

est arrêté dès que $\|r\|_2/\|b\|_2 < 1.0e-6$ ou lorsque le nombre maximum de processus égal à 400 a été atteint.

Les matrices testées sont EX11, EX19 et EX35 de la collection FIDAP, RAEFSKY3 et INACCURA de la collection Simon. Les dimensions des matrices sont comprises entre 12005 et 21000.

Nous considérons les matrices EX11 et EX35 tout d'abord non préconditionnées. Les résultats des tests sont répertoriés dans les tableaux 3.9 et 3.10 . La matrice EX11 a 1096948 éléments non nuls et sa dimension est 16614. La matrice EX35 est de dimension 19716 et possède 228208 éléments non nuls, ce qui fait d'elle une matrice particulièrement creuse. Nous pouvons faire les remarques suivantes :

1. la méthode **ADOPMR** converge plus rapidement que **GMRES**(m) pour tous les cas testés,

2. un processus de la méthode **ADOPMR** est plus rapide qu'un processus de la méthode **GMRES**(m). (Ceci n'est pas vrai pour m égal à 5),

3. pour la matrice EX35, **GMRES**(m) ne converge jamais, tandis que **ADOPMR** converge pour m compris entre 25 et 50. La réduction et le nombre de processus pour converger décroissent au fur et à mesure que m grandit, pour la méthode **ADOPMR**. Le meilleur temps d'exécution est réalisé par **ADOPMR** pour m égal à 25. Du fait que la matrice EX35 est très creuse, le temps d'exécution passé dans la méthode **ADOPMR** seule est important par rapport au temps d'exécution total, d'où l'importance de l'accélération de la méthode **ADOPMR**,

4. pour la matrice EX11, la norme du résidu de **GMRES**(m) diminue lorsque m grandit pour un nombre de processus égal, mais la convergence n'a lieu que pour $m = 50$. La norme du résidu de la méthode **ADOPMR** décroît aussi lorsque m augmente, mais la convergence a lieu dès $m = 20$. Le meilleur temps d'exécution est réalisé par **ADOPMR** pour $m = 20$.

Lorsque nous comparons les méthodes **ADOPMR** et **GMRES**(m) avec **TFQMR**, au vu des tableaux 3.11 et 3.12, nous pouvons dire :

• pour la matrice EX11,

5. **GMRES**(m) n'est pas très performant comparé à la méthode **TFQMR**, puiqu'il ne converge pas dans tous les cas testés,

6. pour des dimensions de redémarrage supérieures à 20, la convergence de **ADOPMR** est meilleure en nombre de produits matrice vecteur et en temps d'exécution,

7. le meilleur temps d'exécution est réalisé par **ADOPMR** pour $m = 20$ avec 4021 produits matrice vecteur en 90.275 secondes.

• pour la matrice EX35, nous avons,

8. **GMRES**(m) n'est pas très performant comparé à la méthode **TFQMR**,

TAB. 3.9 – *Matrice EX11, NbPE = 4, Precon = non*

m	Perf.	GMRES(m)	ADOPMR(m)
5	Proc/Matvec	400 / 2001	400 / 2001
	Temps	43.747	43.954
	Reduction	0.112E-03	0.851E-05
	Accel. Temps	4.773	5.035
10	Proc/Matvec	400 / 4001	400 / 4001
	Temps	90.196	87.792
	Reduction	0.106E-04	0.664E-05
	Accel. Temps	12.550	10.165
15	Proc/Matvec	400 / 6001	400 / 6001
	Temps	139.688	131.902
	Reduction	0.744E-05	0.251E-05
	Accel. Temps	23.315	15.655
20	Proc/Matvec	400 / 8001	**201 / 4021**
	Temps	192.702	**90.275**
	Reduction	0.607E-05	**0.509E-06**
	Accel. Temps	37.208	**12.386**
25	Proc/Matvec	400 / 10001	172 / 4301
	Temps	245.705	96.992
	Reduction	0.469E-05	0.974E-06
	Accel. Temps	53.786	13.732
30	Proc/Matvec	400 / 12001	153 / 4591
	Temps	303.537	105.229
	Reduction	0.331E-05	0.425E-06
	Accel. Temps	73.282	16.401
40	Proc/Matvec	400 / 16001	114 / 4561
	Temps	427.784	105.873
	Reduction	0.133E-05	0.160E-07
	Accel. Temps	120.791	17.652
50	Proc/Matvec	**283 / 14135**	84 / 4201
	Temps	**397.936**	100.635
	Reduction	**0.100E-05**	0.939E-06
	Accel. Temps	**126.791**	19.352

TAB. 3.10 – *Matrice EX35, NbPE = 4, Precon = non*

m	Perf.	GMRES(m)	ADOPMR(m)
5	Proc/Matvec	400 / 2001	400 / 2001
	Temps	23.711	23.636
	Reduction	0.222E-03	0.551E-04
	Accel. Temps	7.534	7.516
10	Proc/Matvec	400 / 4001	400 / 4001
	Temps	51.218	47.068
	Reduction	0.724E-04	0.255E-04
	Accel. Temps	19.115	15.057
20	Proc/Matvec	400 / 8001	400 / 8001
	Temps	119.687	99.792
	Reduction	0.254E-04	0.154E-05
	Accel. Temps	55.768	35.859
25	Proc/Matvec	400 / 10001	**386 / 9651**
	Temps	160.722	**121.621**
	Reduction	0.194E-04	**0.100E-05**
	Accel. Temps	80.897	**44.598**
30	Proc/Matvec	400 / 12001	341 / 10231
	Temps	206.388	133.234
	Reduction	0.140E-04	0.993E-06
	Accel. Temps	110.632	51.624
40	Proc/Matvec	400 / 16001	289 / 11561
	Temps	311.198	155.591
	Reduction	0.618E-05	0.994E-06
	Accel. Temps	183.622	63.434
50	Proc/Matvec	400 / 20001	199 / 9951
	Temps	434.195	142.275
	Reduction	0.171E-05	0.913E-06
	Accel. Temps	274.790	62.999

TAB. 3.11 – *Matrice EX11, NbPE = 4, Precon = no.*

Perf.	TFQMR
Matvec	8759
Temps	168.726
Reduction	0.500E-07
Accel. Temps	8.953

TAB. 3.12 – *Matrice EX35, NbPE = 4, Precon = no.*

Perf.	TFQMR
Matvec	7372
Temps	64.231
Reduction	0.780E-07
Accel. Temps	11.350

9. **ADOPMR** est plus performant que **GMRES**(m) par rapport à **TFQMR**, surtout en terme de nombre de processus,

10. le meilleur temps d'exécution est effectué par la méthode **TFQMR** en 64.231 secondes.

Ce que nous venons de voir pour les matrices non préconditionnées, n'est pas vrai dans le cas du préconditionnement à l'aide de la méthode Jacobi par blocs. Le tableau 3.13 montre le comportement des méthodes **ADOPMR** et **GMRES**(m) pour la matrice EX11 préconditionnée. Dans ce cas, **ADOPMR** effectue moins de processus pour converger que **GMRES**(m) seulement pour $m = 5$ et 7. Le meilleur temps d'exécution pour la méthode **ADOPMR** est atteint pour $m = 20$, mais n'est alors plus compétitif avec le meilleur temps d'exécution de **GMRES**(m).

De façon générale, les expériences réalisées ont montré les points suivants :

- les méthodes **ADOPWMR** et **ADOPHMR** ne sont pas performantes par rapport à la méthode **ADOPMR**, la première en terme de diminution de processus pour converger, la seconde en terme de temps d'exécution d'un processus,

- dans le cas des matrices non préconditionnées, la méthode **ADOPMR** converge plus rapidement que **GMRES**(m),

- utilisée avec un préconditionnement bloc Jacobi, la méthode **ADOPMR** est rarement meilleure que **GMRES**(m),

- la méthode **ADOPOR** n'est pas toujours plus performante que la méthode **FOM**(m),

- les méthodes **ADOP** ne fonctionnent pas bien pour des tailles de redémarrage trop grande,

- l'ensemble des méthodes **ADOPMR** et **ADOPOR** apportent un gain substantiel par rapport, respectivement, aux méthodes **GMRES**(m) et **FOM**(m), en nombre de processus et en temps d'exécution pour des tailles de redémarrage faible.

TAB. 3.13 – *Matrice EX11, NbPE = 4, Precon = Jacobi par blocs*

m	Perf.	GMRES(m)	ADOPMR(m)
5	Proc/Matvec	134 / 671	122 / 611
	Temps	19.554	17.821
	Reduction	0.982E-06	0.462E-06
	Accel. Temps	2.071	1.909
7	Proc/Matvec	80 / 561	75 / 526
	Temps	16.438	15.182
	Reduction	0.890E-06	0.985E-06
	Accel. Temps	1.896	1.562
9	Proc/Matvec	55 / 488	63 / 568
	Temps	14.415	16.474
	Reduction	0.100E-05	0.974E-06
	Accel. Temps	1.811	1.811
10	Proc/Matvec	**30 / 292**	**52 / 521**
	Temps	**8.667**	**15.075**
	Reduction	**0.991E-06**	**0.821E-06**
	Accel. Temps	**1.133**	**1.627**
15	Proc/Matvec	29 / 424	38 / 571
	Temps	12.925	16.517
	Reduction	0.996E-06	0.617E-06
	Accel. Temps	2.009	1.811
25	Proc/Matvec	22 / 549	30 / 751
	Temps	17.666	22.099
	Reduction	0.999E-06	0.855E-06
	Accel. Temps	3.578	2.812
30	Proc/Matvec	18 / 515	46 / 1381
	Temps	17.030	41.038
	Reduction	0.999E-06	0.352E-06
	Accel. Temps	3.822	5.619
40	Proc/Matvec	9 / 345	33 / 1321
	Temps	11.966	39.596
	Reduction	0.961E-06	0.240E-06
	Accel. Temps	3.118	5.739
50	Proc/Matvec	6 / 276	62 / 3101
	Temps	10.012	94.958
	Reduction	0.982E-06	0.159E-05
	Accel. Temps	2.936	15.507

3.7 Conclusion

Nous avons proposé dans ce chapitre des méthodes redémarrées de type **RQO** et **RQM** utilisant un produit scalaire discret, afin de réduire le temps d'exécution passé dans le processus de Hessenberg généralisé sur machines parallèles. Ces méthodes, appelées **ADOP** ont révélé qu'un gain substantiel en temps d'exécution pouvait être obtenu par rapport aux méthodes **FOM**(m) et **GMRES**(m), ceci sans préconditionnement. Ce gain peut être important, plus particulièrement pour les méthodes **PM** avec des matrices très creuses. Utilisées avec un préconditionnement Jacobi par blocs, ces méthodes n'ont pas donné d'aussi bons résultats.

Ces méthodes présentent en outre les avantages suivants :

- un gain substantiel en temps d'exécution par rapport aux méthodes **FOM**(m) et **GMRES**(m) pour des petites tailles de redémarrage. C'est aussi pour ces mêmes tailles que leur meilleur temps total d'exécution est atteint. Il n'est donc pas nécessaire de faire différentes expérimentations pour trouver une dimension de sous-espace convenable, une valeur comprise entre 8 et 15 donnera de bons résultats,
- ces méthodes peuvent être combinées aux méthodes **FOM**(m) et **GMRES**(m) elles-mêmes, de façon à obtenir une méthode encore plus efficace,
- les méthodes **ADOPMR** peuvent être utilisées dans la méthode **FGMRES** pour que celle-ci tire partie de sa rapidité dans le calcul approché des vecteurs $A^{-1}v_i$ (voir chapitre 2).

Dans ce chapitre, nous n'avons utilisé que le préconditionnement Jacobi par blocs. D'autres préconditionnements plus adaptés aux méthodes **ADOP**, devraient pouvoir être trouvés, notamment ceux tirant partie de l'information spectrale calculée à chaque processus. Nous pensons aux préconditionnements polynomiaux développés dans le chapitre 2.

Chapitre 4

Méthodes de Krylov implicitement redémarrées et déflatées

Sommaire

Nous avons vu dans le chapitre précédent une façon d'accélérer les méthodes **FOM**(m) et **GMRES**(m) sur machines parallèles en remédiant aux produits scalaires fait en séquence dans le processus d'Arnoldi. Dans ce chapitre, nous nous attachons aux pertes d'informations dues au redémarrage de ces mêmes méthodes: le redémarrage a l'avantage de pallier aux problèmes de stockage et de calculs croissants avec la dimension m du sous-espace de Krylov, mais freine la convergence des méthodes **FOM** et **GMRES** et la rend plus difficile à analyser. Ainsi, le comportement superlinéaire de **GMRES**, dans sa version non redémarrée, est étroitement lié à la convergence des valeurs de Ritz [60]. De la même façon, même si la norme résiduelle de **FOM** présente des comportements

oscillatoires, sa convergence est liée à la convergence de ces mêmes valeurs. Or, lorsque ces méthodes sont redémarrées, l'information que nous venons d'acquérir au travers de la construction du sous-espace de Krylov, est partiellement perdue et doit être reconstruite dans le processus[7] suivant. De plus, la convergence des valeurs de Ritz dans un sous-espace de dimension m est rendue difficile, lorsque la valeur de m est petite.

Afin de ne pas perdre totalement les informations du processus courant, nous conservons une partie de cette information, notamment spectrale et l'incorporons dans le processus suivant. Nous allons utiliser ce procédé au travers des techniques de déflation. Celles-ci se scindent en deux classes : celles utilisant l'approche préconditionnement et celles enrichissant directement le sous-espace de Krylov. Ces deux approches sont détaillées dans le paragraphe suivant. Pour l'une ou l'autre de ces approches, les résultats numériques ont montré une amélioration significative des techniques de déflation par rapport aux méthodes redémarrées correspondantes, et pour une dimension de sous-espace de Krylov comparable.

Nous présentons dans ce chapitre une technique de déflation enrichissant le sous-espace de Krylov des méthodes **FOM**(m) et **GMRES**(m), en utilisant la méthode **I**mplicitly **R**estarted **A**rnoldi (**IRA**) de Sorensen [58] . Cette technique apporte les qualités suivantes par rapport aux techniques d'enrichissement du sous-espace déjà existantes :

- l'enrichissement de l'espace de Krylov se fait de façon naturelle, car le sous-espace de Krylov enrichi est lui-même un sous-espace de Krylov,
- les techniques de déflation des méthodes **QR** peuvent être appliquées,
- la méthode offre un coût de stockage plus faible que les techniques de déflation avec enrichissement du sous-espace de Krylov et égal au coût des méthodes **FOM**(m) et **GMRES**(m).

Tout au long de ce chapitre nous désignons respectivement **FOM** et **GMRES**, pour **FOM**(m) et **GMRES**(m) et spécifions le terme «non redémarré» lorsque les méthodes ne sont pas redémarrées.

La première section est consacrée aux techniques de déflation en général. Nous présentons notre technique, que nous appliquons à **FOM**, dans la deuxième section. Nous adaptons ensuite cette technique à la méthode **GMRES**, ainsi qu'à une méthode dérivée de **GMRES**, appelée **MGMRES** pour Modified **GMRES**. Nous comparons alors les coûts de ces nouvelles méthodes avec ceux des méthodes existantes et montrons leur intérêt à la fois en environnements séquentiels et parallèles. Nous présentons ensuite les résultats numériques.

4.1 Techniques de déflation

Depuis quelques années, plusieurs techniques de déflation associées aux méthodes de Krylov sont apparues. Ces techniques visent à accélérer leur convergence, le plus fréquem-

7. Nous appelons processus pour une méthode redémarrée dont la dimension du sous-espace de Krylov est m, l'ensemble de m itérations contigües entre deux redémarrages consécutifs

ment dans le souci d'inhiber les effets ralentissants du redémarrage [1, 6, 10, 13, 34, 50] ou bien en vue d'accélérer une méthode non redémarrée comme le **CG** dans le cadre de la résolution successive de plusieurs systèmes linéaires [55].

Nous nous intéressons ici exclusivement aux techniques visant à pallier la perte d'information due au redémarrage des méthodes. Les techniques existantes, jusqu'à ce jour, s'appliquent presque exclusivement à la méthode **GMRES**.

Lorsque la méthode **GMRES** n'est pas redémarrée, Van der Vorst et Vuik ont montré dans [60] que son comportement superlinéaire apparaissait à chaque fois qu'une valeur de Ritz était suffisamment proche de la valeur propre simple correspondante, ou dans le cas d'une valeur propre multiple de multiplicité algébrique n_i, lorsque n_i valeurs de Ritz étaient suffisamment proches de cette valeur propre. Le processus continue alors comme si cette valeur propre était exclue du spectre de la matrice.

Lorsque la méthode est redémarrée, la convergence est plus lente car, à chaque redémarrage, une partie de l'information spectrale est perdue et doit être à nouveau calculée dans le processus suivant. En effet, les seules informations retenues d'un processus à l'autre sont les vecteurs x_m et r_m. Ces deux seuls vecteurs ne suffisent pas à conserver l'ensemble des informations spectrales que contenait le sous-espace de Krylov: la dimension m du sous-espace de Krylov est souvent trop petite, pour que celui-ci parvienne, en général, à approcher avec une précision suffisante certaines valeurs propres et les informations étant perdues entre deux redémarrages, nous ne tirons pas partie, pour le processus suivant, de l'approximation des valeurs propres obtenue dans le processus courant, même si celle-ci est imprécise.

Le principe des techniques de déflation est de calculer un sous-espace invariant de la matrice A, pour l'incorporer dans les processus suivant en agissant:

- soit directement sur le spectre de la matrice A. Lorsqu'un sous-espace invariant de la matrice A est trouvé, celle-ci est remplacée par la matrice $M^{-1}A$: les vecteurs propres de cette matrice sont les mêmes que ceux de A. Le sous-espace invariant de A est donc aussi un sous-espace invariant de $M^{-1}A$. Seules les valeurs propres diffèrent. Celles associées au sous-espace invariant trouvé de A sont remplacées par la valeur propre multiple μ_n, plus grande valeur propre en valeur absolue de la matrice A, les autres valeurs propres restant inchangées par rapport à celles de A. Le processus suivant s'effectue sur la matrice préconditionnée résultante $M^{-1}A$,

- soit sur le vecteur résidu r_m lui-même en l'orthogonalisant par rapport au sous-espace invariant.

Le terme «déflation» signifie que la méthode évolue comme si une partie du spectre, celle correspondant au sous-espace invariant, en était exclue. Cela est réel dans la déflation avec préconditionnement et fait de façon implicite pour le résidu. Le résidu de chaque processus doit être orthogonalisé par rapport au sous-espace invariant.

Or calculer un tel sous-espace n'est pas évident et requiert autant de travail que de résoudre le système linéaire lui-même. Les techniques utilisées effectuent donc la déflation sur un sous-espace invariant approché. L'incorporant dans les processus suivants, elles l'approchent de mieux en mieux et la déflation «réelle» peut avoir lieu.

Nous nous intéressons plus particulièrement au sous-espace invariant associé aux valeurs propres les plus petites. En effet, lorsque la matrice est normale ou très proche de la normalité, son nombre de conditionnement est donné par

$$\kappa = \frac{|\mu_n|}{|\mu_1|}$$

où μ_n et μ_1 sont respectivement la plus grande et la plus petite valeurs propres de A en valeur absolue. Lorsque la méthode déflatée approche de façon suffisamment précise les plus petites valeurs propres $\mu_1, \ldots, \mu_k$, le processus continue comme si ces valeurs propres étaient exclues du spectre de la matrice et que son nombre de conditionnement était égal à $\tilde{\kappa} = \frac{|\mu_n|}{|\mu_{k+1}|} \leq \kappa$. Naturellement, le nombre de conditionnement seul ne suffit pas à diagnostiquer la convergence mais y contribue.

La raison supplémentaire pour laquelle nous sommes particulièrement intéressés par les plus petites valeurs propres est liée à l'approche polynomiale. Nous avons vu dans le chapitre 1 que le résidu peut se mettre sous la forme $r_m = (I - As_m(A))r_0$ où s_m est un polynôme de degré strictement inférieur à m et à coefficients réels. Le polynôme $p_m(x) = 1 - xs_m(x)$ s'appelle le polynôme résiduel. Le polynôme résiduel recherché est le polynôme minimal associé à A et r_0, qui a donc comme racine les valeurs propres associées aux vecteurs propres le long desquels r_0 n'a pas de composante nulle. Par définition $p_m(0) = 1$ et il est donc difficile de faire en sorte qu'il vaille zéro pour des valeurs correspondant aux plus petites valeurs propres très proches de 0 et 1 en zéro. La déflation effectuée avec des valeurs propres les plus petites est un moyen d'éviter de telles difficultés.

Nous présentons successivement les deux approches des techniques de déflation en commençant par celle utilisant le préconditionnement.

4.1.1 Préconditionnement

Cette technique de déflation appliquée à une méthode de Krylov, ici à **GMRES**, a été introduite par Erhel et al. dans [13] en vue de minimiser les effets du redémarrage.

Nous nous plaçons dans le cas d'une matrice normale et nous supposons qu'un sous-espace invariant de dimension k associé aux k plus petites valeurs propres $\mu_1, \ldots, \mu_k$ a été trouvé. Soit U une matrice orthonormale dont l'image est égale au sous-espace invariant, que nous complétons par W une matrice de dimension $n \times (n-k)$, de telle sorte que la matrice $Z = [U, W]$ soit orthonormale. Nous avons alors

$$Z^T A Z = \begin{pmatrix} T & A_{1,2} \\ 0 & A_{2,2} \end{pmatrix}$$

où T est une matrice triangulaire supérieure et le spectre de la matrice $A_{2,2}$ est égal à l'ensemble des valeurs propres $\mu_{k+1}, \ldots, \mu_n$. La matrice M définie ainsi :

$$M = Z \begin{pmatrix} T/|\mu_n| & 0 \\ 0 & I_{n-k} \end{pmatrix} Z^T$$

est non singulière et vérifie

$$M^{-1} = Z \begin{pmatrix} |\mu_n| T^{-1} & 0 \\ 0 & I_{n-k} \end{pmatrix} Z^T. \tag{4.1}$$

Cette matrice peut se réécrire très simplement sous la forme

$$\begin{aligned} M^{-1} &= |\mu_n| U T^{-1} U^T + (I_n - U U^T) \\ &= I_n + U(|\mu_n| T^{-1} - I_k) U^T. \end{aligned}$$

La matrice $M^{-1}A$ se met alors sous la forme

$$M^{-1}A = Z \begin{pmatrix} |\mu_n| I_k & |\mu_n| T^{-1} A_{1,2} \\ 0 & A_{2,2} \end{pmatrix} Z^T.$$

Les valeurs propres de la matrice ainsi formée sont exactement $\mu_{k+1}, \ldots, \mu_{n-1}$ et $k+1$ fois μ_n. Si nous appliquons l'algorithme **GMRES** sur une telle matrice, le nombre de conditionnement passe de la valeur κ à celle de $\tilde{\kappa}$ et une convergence plus rapide est attendue.

En pratique, un tel sous-espace invariant n'est pas connu exactement et nous devons nous contenter d'un sous-espace approché. Nous l'obtenons à l'aide des informations contenues dans la factorisation d'Arnoldi :

$$AV_m = V_{m+1} \bar{H}_m. \tag{4.2}$$

où H_m est en fait la matrice de projection orthogonale de A sur le sous-espace de Krylov K_m. Cette équation est appelée factorisation d'Arnoldi de longueur m, car elle vérifie :

- H_m est une matrice de Hessenberg supérieure irréductible à sous-diagonale strictement positive,
- V_m est une matrice orthonormale et $V_m^T v_{m+1} = 0$.

Nous calculons alors les couples de Ritz $(y_i = V_m u_i, \mu_i)$ tels que $H_m u_i = \mu_i u_i$. Ils vérifient :

$$A y_i = \mu_i y_i + h_{m+1,m} v_{m+1} e_m^{(m)T} u_i.$$

Lorsque le terme $|h_{m+1,m} e_m^{(m)T} u_i|$ est suffisamment faible, nous disons que le couple (y_i, μ_i) a convergé.

Dans l'algorithme introduit par Erhel et al. dans [13], les couples de Ritz sont pris en considération pour former la matrice U même s'ils n'ont pas convergé. Le procédé est le suivant : pour chaque processus, les couples de Ritz de la matrice H_m sont calculés. Ceux associés aux k plus petites valeurs de Ritz du processus courant sont conservés pour être ajoutés aux vecteurs colonnes de U, la matrice issue du processus précédent, formant ainsi la nouvelle matrice U. Le nouveau préconditionneur M^{-1} qui en découle est utilisé dans le processus suivant.

L'algorithme que nous appelons **DPGMRES** pour **D**eflated **P**reconditionning **GMRES** est résumé ci-dessous :

Algorithme 4.1 $[x_0, r_0]$ = *DPGMRES(A,x_0,b,m,k,iter,tol)*

1. *Initialisation :* $M = I_n$, $U = \emptyset$, $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$

2. 1 *processus de GMRES:*
   $[x_m, r_m, V_{m+1}, \bar{H}_m] = GMRES(M^{-1}A, x_0, m)$
3. *Redémarrage:*
   (a) $x_0 = x_m$, $r_0 = r_m$, $iter = iter + 1$
   (b) *Si* $(\|r_0\|/\beta < tol)$ *ou si* $(iter = maxit)$ *stop*
   (c) *Calcul des k vecteurs de Schur approchés* Y_k *associés aux k plus petites valeurs de Ritz de* H_m
   (d) *Orthogonalisation des vecteurs de Schur par rapport à ceux précédents* $Y_k = (I_n - UU^T)Y_k$ *et mise à jour de* $U = [U, Y_k]$
   (e) *Mise à jour de* M^{-1} *:*
   $T = U^T AU$ *et* $M^{-1} = I_n + U(|\mu_n|T^{-1} - I)U^T$

Un préconditionneur est donc invariant pendant un processus et est mis à jour au moment du redémarrage. L'intérêt de la méthode est que le préconditionnement est facile à appliquer et qu'il peut s'utiliser sur un système lui-même préconditionné. Néanmoins, U augmente de k vecteurs à chaque processus et son stockage ainsi que son application lors des phases de préconditionnement deviennent vite prohibitifs. Par ailleurs, les vecteurs de Ritz formant U n'ont pas nécessairement convergé et le préconditionnement n'est pas très précis.

Cette même méthode a été améliorée en vue de remédier à ces inconvénients [6]:

- les couples de Ritz pris en compte dans U sont à présent issus d'une projection harmonique, que nous verrons dans le prochain paragraphe. Cette projection est plus adaptée à la recherche des valeurs propres les plus proches de zéro pour la méthode **GMRES**,
- la taille de U est limitée à max et U est mis à jour en gardant les max vecteurs de Ritz associés aux valeurs de Ritz les plus petites parmi les max vecteurs de Ritz U du processus précédent et les k vecteurs de Ritz associés aux plus petites valeurs de Ritz du processus courant.

Il est à noter que, dans cette nouvelle version, le préconditionnement ne se fait plus à gauche mais à droite.

Dans [1], les auteurs présentent deux algorithmes qui utilisent la méthode **I**mplicitly **R**estarted **A**rnoldi (**IRA**) de Sorensen [58] pour le calcul de la matrice de préconditionnement. La méthode **IRA** a été introduite dans le cadre de la résolution de valeurs propres et représente l'une des méthodes les plus performantes dans ce domaine. Elle sera vue plus en détail dans la prochaine section. Elle permet d'approcher rapidement et efficacement le sous-espace invariant qui nous intéresse. Les algorithmes de [1] se divisent en deux phases:

- la phase de calcul du préconditionnement. Le sous-espace invariant est approché grâce à la méthode **IRA** tout en mettant à jour la solution approchée du système linéaire, soit à l'aide de la méthode de Richardson (algorithme numéro 1) soit à

l'aide de **GMRES** (algorithme numéro 2). Lorsqu'un sous-espace invariant de A est suffisamment bien approché, la matrice M_1^{-1} est calculée grâce à l'équation (4.1) et le procédé est appliqué à la matrice $M_1^{-1}A$. Lorsqu'un second sous-espace invariant de $M_1^{-1}A$ est suffisamment bien approché, la matrice de préconditionnement correspondante M_2 est calculée. La nouvelle matrice de préconditionnement pour les processus suivants devient alors le produit résultant de toutes les matrices de préconditionnement, soit pour notre cas $M = M_1M_2$. Le procédé est ainsi itéré jusqu'à ce qu'un nombre maximum fixé d'itérations soit atteint,

- la méthode **GMRES** elle-même utilisée avec le dernier préconditionnement de la phase précédente.

Les algorithmes ainsi construits bénéficient de la qualité de la méthode **IRA**, en construisant un préconditionnement efficace. L'utilisation de la méthode **IRA** permet en outre de diminuer le nombre de produits matrice vecteur avec la matrice A par rapport à la méthode **GMRES**, lesquels sont remplacés par des factorisations QR sur une matrice de dimension réduite. Un processus de la première phase est donc moins coûteux qu'un processus de la seconde phase.

Ces méthodes sont très efficaces et réduisent le nombre total de produits matrice vecteur de façon significative par rapport à **GMRES**, pour une même dimension de sous-espace de Krylov. Nous présentons dans le paragraphe suivant une autre approche de la déflation.

4.1.2 Enrichissement du sous-espace de Krylov

Cette seconde approche est celle qui nous intéresse et que nous développerons dans les sections suivantes. Elle consiste à projeter et calculer la solution approchée sur un sous-espace de Krylov qui a été enrichi. Ce sous-espace est de la forme :

$$S_{m,k}(A, r_0, y_1, \ldots, y_k) = \text{Span}\{r_0, Ar_0, \ldots, A^{m-k-1}r_0, y_1, \ldots, y_k\} \tag{4.3}$$

où r_0 est le résidu initial et $y_1, \ldots, y_k$ sont des vecteurs propres associés aux valeurs propres les plus petites. La solution approchée calculée est celle pour laquelle le résidu correspondant est orthogonal à $AS_{m,k}(A, r_0, y_1, \ldots, y_k)$ et qui minimise donc la norme du résidu sur $S_{m,k}(A, r_0, y_1, \ldots, y_k)$. Nous calculons ainsi une solution de type **PM** sur un sous-espace enrichi. Les vecteurs propres étant A-invariants, la construction même de la solution implique que le résidu correspondant est orthogonal aux vecteurs propres. Dans le cas d'une matrice normale, lorsqu'une valeur propre est multiple et que nous ne prenons en compte qu'un seul vecteur propre associé à cette valeur propre, c'est une seule instance de cette valeur propre qui est exclue du procédé.

Comme les vecteurs propres ne sont pas disponibles dès le début de l'algorithme, Morgan introduit dans [34] une méthode qui enrichit le sous-espace de Krylov courant à l'aide de vecteurs de Ritz associés aux valeurs de Ritz les plus petites et qui sont calculés dans le processus précédent. La solution **GMRES** est ensuite projetée sur ce sous-espace.

Nous appelons **Augmented GMRES (AGMRES)**, la méthode introduite par Morgan et notons x_m^{AG} la solution approchée qu'elle calcule. Morgan construit tout d'abord une

base orthonormale du sous-espace $S_{m,k}(A, r_0, y_1, \ldots, y_k)$ où les vecteurs $y_1, \ldots, y_k$ sont des vecteurs de Ritz calculés lors du processus précédent. Il utilise pour construire cette base la même technique que **FGMRES** (voir algorithme 2.14). Le procédé dénommé Augmented-Arnoldi est résumé ci-dessous :

Algorithme 4.2 $[W_m, V_{m+1}, \bar{H}_m]$ = *Augmented-Arnoldi(A,*r_0*,m,k,*$y_1, \ldots, y_k$*)*

1. *Initialisation :* $v_1 = r_0/\|r_0\|_2$
2. *Gram-Schmidt Modifié :*
   Pour $j = 1, \ldots, m$ *Faire*
   - *Si* $j \leq (m-k)$, $w = Av_j$, *sinon* $w = Ay_{j-m+k}$
   - *Pour* $i = 1, \ldots, j$ *Faire*
     $h_{i,j} = (w, v_i)$
     $w = w - h_{i,j} v_i$
   - *Fin Pour*
   - $h_{j+1,j} = \|w\|_2$, $v_{j+1} = w/h_{j+1,j}$

   Fin Pour
3. $V_{m+1} = [v_1, \ldots, v_{m+1}]$, $W_m = [V_{m-k}, y_1, \ldots, y_k]$ *et* $\bar{H}_m = \{h_{i,j}\}$

Les matrices W_m et V_{m+1} définies ci-dessus satisfont l'équation suivante :

$$\begin{aligned} AW_m &= V_{m+1}\bar{H}_m & (4.4)\\ &= V_m H_m + h_{m+1,m} v_{m+1} e_m^T, & (4.5) \end{aligned}$$

qui remplace la factorisation d'Arnoldi (4.2).

Nous calculons alors les vecteurs de Ritz qui seront utilisés pour enrichir le sous-espace de Krylov du processus suivant. Nous les calculons à partir de la projection suivante, appelée projection harmonique :

$$W_m^T A^T W_m u_j = \frac{1}{\mu_j} W_m^T A^T A W_m u_j, \tag{4.6}$$

qui permet de calculer les valeurs propres proches de zéro plus aisément. Les paires de Ritz sont de la forme $(y_j = W_m u_j, \mu_j)$ et nous choisissons celles pour lesquelles les valeurs μ_j sont petites en valeur absolue.

C'est cette même projection qui est utilisée pour le calcul du préconditionnement de la matrice A de la méthode [6], amélioration de la méthode [13], présentée dans le paragraphe précédent.

La solution de la méthode **AGMRES** est calculée, quant à elle, par projection sur $S_{m,k}(A, r_0, y_1, \ldots, y_k)$ de façon à minimiser la norme du résidu

$$\begin{aligned} x_m^{AG} &= x_0 + W_m y_m^{AG} \\ r_m^{AG} &= r_0 - A W_m y_m^{AG} \end{aligned}$$

et

$$\begin{aligned} y_m^{AG} &= \arg\min_{y\in\mathbb{R}^m} \|r_0 - AW_m y_m^{AG}\|_2 \\ &= \arg\min_{y\in\mathbb{R}^m} \|\beta e_1^{(m+1)} - \bar{H}_m y_m^{AG}\|_2 \end{aligned}$$

où y_m^{AG} est calculé à l'aide des rotations de Givens, comme cela est fait pour la méthode **GMRES**. La méthode **AGMRES** peut donc être vue comme une méthode de projection de type **PM** sur un sous-espace de Krylov enrichi.

L'algorithme final est le suivant :

Algorithme 4.3 $[x_0, r_0, V_{m+1}, W_m, \bar{H}_m, iter]$ = *AGMRES(A,*x_0*,b,m,k, maxit,tol)*

1. *Initialisation :* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$

2. *Premier processus :*

   (a) 1 *processus de GMRES :*
   $[x_0, r_0, V_{m+1}, \bar{H}_m]$ = *GMRES(A,*x_0*,m)*

   (b) $W_m = V_m$, $iter = iter + 1$

   (c) *Si* $(\|r_0\|/\beta < tol)$ *ou si* $(iter = maxit)$ *alors arrêt.*

3. *Choix de k vecteurs de Ritz* $y_i = W_m u_i$ *où* (u_i, μ_i) *satisfait*
   $W_m^T A^T W_m u_i = \frac{1}{\mu_i} A^T W_m^T A W_m u_i$

4. *Construction de la base orthonormale :*
   $[W_m, V_{m+1}, \bar{H}_m]$ = *Augmented-Arnoldi(A,*r_0*,m,k,*$y_1, \ldots, y_k$*)*

5. *Calcul de la solution approchée :*

   (a) $\tilde{y} = \arg\min_y \|\beta e_1^{(m+1)} - \bar{H}_m y\|_2$

   (b) $x_0 = x_0 + W_m \tilde{y}$

   (c) $r_0 = r_0 - AW_m \tilde{y}$

6. *Redémarrage :*

   (a) $iter = iter + 1$

   (b) *Si* $(\|r_0\|/\beta < tol)$ *ou si* $(iter = maxit)$ *alors arrêt, sinon aller en* 3.

La solution calculée étant de type **PM**, elle vérifie l'équation (1.22).

Nous proposons d'améliorer cette méthode, grâce notamment à la méthode **IRA**.

4.2 Méthodes implicitement redémarrées et déflatées

Nous introduisons dans ce paragraphe une technique de déflation avec enrichissement du sous-espace de Krylov, qui tire profit des performances de la méthode **IRA** pour rendre cette déflation encore plus efficace. La méthode résultante a été exposée à la conférence ICNPDE[8] au mois de Septembre 98 à Marrakech et est introduite dans [28].

Cette méthode, en plus de calculer la solution approchée x_m à chaque processus, améliore l'approximation des vecteurs propres et des valeurs propres qui nous intéressent. Ceci est rendu possible parce que le sous-espace de Krylov enrichi

$$\text{Span}\{y_1, y_2, \ldots, y_k, r_0, Ar_0, \ldots, A^{m-k-1}r_0\} \tag{4.7}$$

identique à (4.3), est exactement égal à tous les sous-espaces suivants

$$\text{Span}\{y_1, y_2, \ldots, y_k, Ay_i, A^2y_i, \ldots, A^{m-k}y_i\} \tag{4.8}$$

pour $i = 1, \ldots, k$.

Le sous-espace de Krylov enrichi contient donc pour chaque vecteur de Ritz $(y_i)_{1\leq i\leq k}$ qui nous intéresse le sous-espace de Krylov $\mathrm{K}_{m-k+1}(A, y_i)$ pour $i = 1, \ldots, k$. Il va donc nous permettre d'améliorer la précision de chacun des paires de Ritz désirées. De plus, ce même sous-espace (4.7) est lui-même un sous-espace de Krylov $K_m(A, v_1)$ où v_1 est un vecteur combinaison linéaire particulière des vecteurs de Ritz $(y_i)_{1\leq i\leq k}$. Ceci nous permet notamment de ne pas avoir à sauvegarder les vecteurs $y_1, \ldots, y_k$ et donc de réduire les coûts de stockage par rapport à la méthode **AGMRES** de Morgan, où ceux-ci doivent être stockés en plus des vecteurs de la base orthonormale $(w_i)_{1\leq i\leq m}$. Ce sous-espace de Krylov nous permet également d'appliquer la déflation implicite, comme cela est fait pour la méthode d'Arnoldi dans [45], lorsque des paires de Ritz ont convergé.

Nous voulons ajouter que ce travail est indépendant de celui développé récemment par Morgan [36] où il introduit la méthode **IRA** pour enrichir les sous-espaces de Krylov au sein de **FOM** et **GMRES**. De plus, notre approche est différente, plus particulièrement pour **GMRES**.

Nous présentons tout d'abord le principe de la méthode implicitement redémarrée et déflatée, que nous appliquons en premier lieu à **FOM**, notre point de départ .

Nous l'adaptons ensuite et successivement à **GMRES** et à **MGMRES**.

4.2.1 Principe

Nous commençons par présenter la méthode **IRA**. Elle utilise la méthode d'Arnoldi qui projette le problème de valeurs propres $Au = \mu u$ sur un sous-espace de Krylov et résout le système projeté correspondant de dimension plus petite. La première phase consiste à calculer une base orthonormale de $\mathrm{K}_m(A, v_1)$ représentée par V_m à l'aide du processus d'Arnoldi (algorithme 1.2 du chapitre 1). Lorsque $v_1 = r_0$, nous pouvons par ailleurs extraire de cette même base soit une solution approchée de type **FOM** soit une solution approchée de type **GMRES**. Poursuivant la méthode d'Arnoldi, nous calculons

8. International Congress on Numerical Methods for Partial Differential Equations

ensuite les vecteurs et valeurs de Ritz correspondant à la projection, soient les paires $(y_i = V_m u_i, \mu_i)$ telles que :

$$\begin{aligned} V_m^T A V_m u_i &= \mu_i V_m^T V_m u_i \\ H_m u_i &= \mu_i u_i. \end{aligned}$$

Comme c'est le cas pour les méthodes **FOM** et **GMRES**, lorsque m grandit, les coûts de calcul et de stockage deviennent trop importants. La méthode d'Arnoldi est donc redémarrée après avoir choisi un nouveau vecteur de départ v_1. Or tous les vecteurs et valeurs de Ritz qui nous intéressent n'ont pas nécessairement convergé et une grande partie de l'information risque d'être perdue entre les deux processus. Le choix de v_1 est donc capital. Supposons que nous soyons intéressés par les vecteurs de Ritz $y_1, \ldots, y_k$ et que nous voulions les voir converger. Le meilleur vecteur de départ v_1 serait alors celui pour lequel le sous-espace $\mathrm{K}_k(A, v_1)$ est exactement égal au sous-espace engendré par ces vecteurs de Ritz qui nous intéressent.

Plusieurs techniques ont été proposées pour tenter de résoudre ce problème : l'une d'elles consiste à appliquer la méthode d'Arnoldi par blocs. Le problème est que le degré des polynômes générés pour chacun des vecteurs du bloc initial est exactement égal à m/p si m est la dimension du sous-espace de Krylov et p celle du bloc. Généralement, cette dimension m/p est trop petite pour obtenir une précision suffisante des paires de Ritz rapidement. Une deuxième technique consiste à choisir v_1 comme combinaison linéaire des vecteurs de Ritz voulus. Cette solution soulève le problème du choix des poids dans la combinaison linéaire.

Morgan a montré dans [35] qu'il existait une unique combinaison linéaire des vecteurs de Ritz désirés et qui vérifie la propriété voulue. Toutes les autres combinaisons linéaires apportent nécessairement des erreurs et des imprécisions. Les poids de cette combinaison linéaire « idéale » sont en fait calculés de façon implicite dans l'*implicit shifted QR-iteration* de la méthode **IRA** [58] lorsque les *shifts* choisis sont exacts. Après avoir construit la base orthonormale formé par les vecteurs colonnes de la matrice V_m vérifiant la relation (4.2), la méthode **IRA** construit une nouvelle base orthonormale $\widehat{V}_k = [\widehat{v}_1, \ldots, \widehat{v}_k]$ dont les colonnes engendrent l'espace des vecteurs de Ritz désirés. Cette matrice satisfait :

$$A\widehat{V}_k = \widehat{V}_{k+1}\widehat{\widehat{H}}_k, \tag{4.9}$$

où $\widehat{\widehat{H}}_k$ est une matrice de Hessenberg supérieure de dimension $(k+1) \times k$ et $\mathrm{Span}\{\widehat{V}_k\} = \mathrm{K}_k(A, \hat{v}_1) = \{y_1, \ldots, y_k\}$ avec $\hat{v}_1$ le vecteur *idéal*, combinaison linéaire des vecteurs de Ritz $(y_i)_{1 \leq i \leq k}$. Cette relation est en fait une factorisation d'Arnoldi de longueur k. Les matrices $\widehat{V}_k$ et $\widehat{\widehat{H}}_k$ sont calculées grâce à $m-k$ *implicit shifted QR-steps* et sont en fait les mêmes matrices que nous aurions obtenues en utilisant le processus d'Arnoldi à partir du vecteur $\hat{v}_1$ pour obtenir une factorisation de longueur k. Leur construction grâce aux $m-k$ *implicit shifted QR-steps*, nous permet d'économiser k produits matrice vecteur avec la matrice A du processus d'Arnoldi et l'utilisation des *shifts* « exacts » dans le *QR-iteration* nous permet d'avoir la relation suivante :

$$\widehat{v}_{k+1} = \pm v_{m+1}. \tag{4.10}$$

Cette relation est démontrée dans [58].

Nous poursuivons la factorisation (4.9) de $m-k$ pas, cette fois-ci à l'aide du processus d'Arnoldi. Nous obtenons alors :

$$A\widehat{V}_m = \widehat{V}_{m+1}\widehat{\bar{H}}_m$$

avec $\mathrm{Span}\{\widehat{V}_m\} = \mathrm{K}_m(A, \hat{v}_1)$.

Cette relation nous permet de faire le lien avec notre système linéaire. Supposons que nous ayons choisi $v_1 = r_0$ et extrait de l'espace engendré par V_m la solution approchée x_m^F de la méthode **FOM**. Nous savons d'après la relation (1.20) que le résidu correspondant r_m^F est un multiple de v_{m+1}. Or $\hat{v}_{k+1} = \pm v_{m+1}$, ce qui nous permet d'extraire une nouvelle solution approchée du sous-espace $\mathrm{K}_m(A, \hat{v}_1)$ en résolvant :

$$\begin{aligned} \widehat{H}_m y_m^F &= \beta e_{k+1}^{(m)} \\ x_m^F &= x_m^F + \widehat{V}_m y_m^F \end{aligned}$$

avec $\beta = \hat{v}_{k+1}^T r_m^F$. Une fois de plus, nous avons que le nouveau résidu r_m^F est un multiple du vecteur $\hat{v}_{m+1}$ et le même procédé peut être appliqué. Nous calculons d'abord les paires de Ritz associées à la matrice $\widehat{H}_m$ et sélectionnons celles qui nous intéressent. Grâce à $m-k$ *implicit shifted QR-steps*, nous formons la nouvelle matrice $\widehat{V}_k$ dont les colonnes engendrent le sous-espace $\mathrm{K}_k(A, \hat{v}_1)$ avec $\hat{v}_1$ le vecteur *idéal*, combinaison linéaire des vecteurs de Ritz qui nous intéressent. Nous avons alors que le résidu r_m^F est un multiple de $\hat{v}_{k+1}$, ce qui nous permet, après avoir poursuivi la factorisation d'Arnoldi de k pas, jusqu'à ce qu'elle soit de longueur m, grâce au processus d'Arnoldi, de calculer la nouvelle solution approchée de **FOM**.

À l'aide d'une légère modification, nous adaptons cette technique aux deux méthodes supplémentaires **GMRES** et **MGMRES**.

Dans le paragraphe qui suit, nous détaillons cette technique appliquée à la méthode **FOM**, que nous appelons **IDFOM** pour **I**mplicitly Restarted and **D**eflated **FOM**.

4.2.2 Méthode implicitement déflatée appliquée à FOM

Dans un premier temps, nous considérons que, parmi l'ensemble des paires de Ritz calculées, aucune n'a encore convergé.

Sans déflation « réelle »

Le tout premier processus consiste en un processus de **FOM**. Nous calculons ensuite les vecteurs et valeurs propres (u_i, μ_i) de $\bar{H}_m$ et les paires de Ritz correspondantes $(y_i = V_m u_i, \mu_i)$ de A. Numérotons-les de telle sorte que $\mu_{k+1}, \ldots, \mu_m$ constituent les valeurs de Ritz qui ne nous intéressent pas, typiquement celles correspondant aux valeurs de Ritz les plus grandes en valeur absolue et $\mu_1, \ldots, \mu_k$ les valeurs de Ritz désirées que nous voudrions mieux approcher. Nous posons $p = m-k$ et procédons à p *implicit shifted QR-steps* sur la matrice H_m, en prenant les *shifts* égaux à $\mu_{k+1}, \ldots, \mu_m$. Pour le premier *shift* μ_{k+1}, nous obtenons :

$$H_m - \mu_{k+1} I_m = Q_1 R_1$$

et

$$H_m^1 = Q_1^T H_m Q_1 = R_1 Q_1 + \mu_{k+1} I_m.$$

Bien que la formule précédente soit écrite dans sa forme explicite, H_m^1 est calculée de façon implicite à l'aide la méthode QR implicite (voir [17, 22]). H_m étant Hessenberg supérieure, Q_1 l'est aussi, ainsi que H_m^1.
Si nous effectuons une translation à partir de l'équation (4.2), nous obtenons:

$$\begin{aligned} AV_m - \mu_{k+1}V_m - V_m(H_m - \mu_{k+1}I_m) &= h_{m+1,m}v_{m+1}e_m^{(m)T} \\ (A - \mu_{k+1}I_n)V_m - V_m(Q_1R_1) &= h_{m+1,m}v_{m+1}e_m^{(m)T} \\ (A - \mu_{k+1}I_n)(V_mQ_1) - (V_mQ_1)(R_1Q_1) &= h_{m+1,m}v_{m+1}e_m^{(m)T}Q_1 \\ A(V_mQ_1) - (V_mQ_1)(R_1Q_1 + \mu_{k+1}I_n) &= h_{m+1,m}v_{m+1}e_m^{(m)T}Q_1 \end{aligned} \tag{4.11}$$

et finalement

$$AV_m^1 - V_m^1 H_m^1 = h_{m+1,m}v_{m+1}e_m^{(m)T}Q_1 \tag{4.12}$$

avec $V_m^1 = V_mQ_1$. En appliquant le vecteur $e_1^{(m)}$ à droite de l'équation (4.11), nous obtenons

$$(A - \mu_{k+1}I_n)v_1 = r_{1,1}^1 v_1^1,$$

où $r_{i,j}^k$ sont définis comme étant les coefficients de la matrice R_k.
Répétons ce procédé jusqu'à ce que nous ayons appliqué p *shifts*. Comme cela a été fait précédemment, nous le faisons de façon implicite. L'équation (4.12)est alors transformée en l'équation

$$AV_m^p - V_m^p H_m^p = h_{m+1,m}v_{m+1}e_m^{(m)T}Q \tag{4.13}$$

avec $Q = Q_1 \ldots Q_p$,$V_m^p = V_mQ$ et $H_m^p = Q^T H_m Q$. Le vecteur $v_1^p = V_m^p e_1^{(m)}$ satisfait alors la propriété intéressante d'être un multiple de $\prod_{i=1}^p (A - \mu_{k+i}I_n)v_1$. Les matrices V_m^p et H_m^p, quant à elles, ont la forme suivante:

$$V_m^p = [\widehat{V}_{m-p}, \widetilde{V}_p] \tag{4.14}$$

$$H_m^p = \begin{pmatrix} \widehat{H}_{m-p} & \widetilde{M} \\ \widehat{\beta}e_1^{(p)}e_{m-p}^{(m-p)T} & \widetilde{H}_p \end{pmatrix}. \tag{4.15}$$

Les matrices Q_i étant chacune Hessenberg supérieure pour $i = 1, \ldots, p$, nous avons que $e_m^{(m)T}Q$ a la forme suivante:

$$e_m^{(m)T}Q = (0, \ldots, 0, \alpha_{m-p}, \ldots, \alpha_m). \tag{4.16}$$

Du fait que nous avons appliqué les *shifts* exacts, c'est-à-dire des *shifts* égaux à certaines valeurs propres de la matrice H_m, nous avons $\hat{\beta} = 0$ (voir [58]).
Si nous ne gardons que les $k = m - p$ premières colonnes de (4.13), l'équation devient:

$$A\widehat{V}_{m-p} - \widehat{V}_{m-p}\widehat{H}_{m-p} = \underbrace{(h_{m+1,m}\alpha_{m-p}v_{m+1})}_{w}e_{m-p}^{(m-p)T}. \tag{4.17}$$

En notant $\widehat{h}_{m-p+1,m-p} = \|w\|_2$ et $\widehat{v}_{m-p+1} = w/\widehat{h}_{m-p+1,m-p}$, nous obtenons

$$A\widehat{V}_{m-p} = \widehat{V}_{m-p}\widehat{H}_{m-p} + \widehat{h}_{m-p+1,m-p}\widehat{v}_{m-p+1}e_{m-p}^{(m-p)T}$$

qui est en fait l'équation suivante :

$$A\widehat{V}_k = \widehat{V}_k\widehat{H}_k + \widehat{h}_{k+1,k}\widehat{v}_{k+1}e_k^{(k)T}. \tag{4.18}$$

Cette dernière relation est en fait une factorisation d'Arnoldi de longueur k, pour les raisons suivantes :

- $\widehat{H}_k$ est une matrice de Hessenberg supérieure irréductible à sous-diagonale strictement positive
- $\widehat{V}_k$ est une base orthonormale
- $\widehat{V}_k^T\widehat{v}_{k+1} = 0$.

Nous avons donc construit une base orthonormale du sous-espace de Krylov $K_k(A, \widehat{v}_1)$, où $\widehat{v}_1$ est un multiple de $\prod_{i=1}^{p}(A - \mu_{k+i}I_n)v_1$. Sorensen a démontré dans [58], lemme (3.10), que $\widehat{v}_1$ est en fait une combinaison linéaire des vecteurs de Ritz désirés y_i pour $i = 1, \ldots, k$. C'est en fait l'*unique* combinaison linéaire des vecteurs $y_1, \ldots, y_k$ telle que :

$$\text{Span}\{\widehat{v}_1, A\widehat{v}_1, \ldots, A^{k-1}\widehat{v}_1\} = \text{Span}\{\widehat{V}_k\} = \text{Span}\{y_1, \ldots, y_k\}. \tag{4.19}$$

Cette relation a été démontrée par Morgan dans [35]. Nous utiliserons le même schéma de démonstration pour l'adaptation au cas de **GMRES**.

Nous complétons la factorisation d'Arnoldi (4.18) par p itérations du processus d'Arnoldi. Après avoir supprimé les accents circonflexes, nous obtenons la factorisation d'Arnoldi de longueur m suivante :

$$AV_m = V_mH_m + h_{m+1,m}v_{m+1}e_m^T. \tag{4.20}$$

Nous avons alors (voir [35]) :

$$\begin{aligned} \text{Span}\{V_m\} &= \text{Span}\{y_1, \ldots, y_k, v_{k+1}, Av_{k+1}, \ldots, A^{p-1}v_{k+1}\} && (4.21)\\ &= \text{Span}\{y_1, \ldots, y_k, Ay_i, \ldots A^p y_i\} && (4.22) \end{aligned}$$

pour $i = 1, \ldots, k$.

Afin d'itérer le procédé, nous mettons à jour les vecteurs ainsi : $x_0^F = x_m^F$, $y_0^F = y_m^F$, $r_0^F = r_m^F$. La nouvelle solution approchée de type **FOM** se calcule alors grâce à :

$$\begin{aligned} x_m^F &= x_0^F + V_m y_m^F && (4.23)\\ y_m^F &= (H_m^{-1})(-h'_{m+1,m}(y_0^F)_m)e_{k+1}^{(m)}, && (4.24) \end{aligned}$$

où $h'_{m+1,m}$ est le coefficient $(m+1) \times m$ de la matrice $\bar{H}_m$ du processus précédent.

L'algorithme sans déflation est résumé ci-dessous :

Algorithme 4.4 $[x_0, r_0, V_{m+1}, \bar{H}_m, iter]$ = *IDFOM(A,x₀,b,m,k,maxit,tol)*

1. *Initialisation :*

   (a) $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$

(b) 1 *Processus de FOM:*
$[x_0, r_0, V_{m+1}, \bar{H}_m]$ = *FOM(A,x_0,m)*

(c) *iter* = *iter* + 1, *si* $(\|r_0\|_2/\beta < tol)$ *ou* (*iter* = *maxit*) *alors arrêt*

2. *Calcul des paires de Ritz* $(y_i = V_m u_i, \mu_i)$ *où les couples* (u_i, μ_i) *satisfont* $H_m u_i = \mu_i u_i$. *Choix des p paires non désirées* $\mu_{k+1}, \ldots, \mu_m$.

3. *Calcul de* $\bar{H}_k$ *et* V_{k+1}, *les matrices résultantes des p pas de l'*`implicit shifted QR-iteration` *en prenant* $\mu_{k+1}, \ldots, \mu_m$ *comme shifts.*

4. *Compléter la factorisation d'Arnoldi à l'aide de p pas du processus d'Arnoldi*

5. *Calcul de la solution approchée:*

   (a) *Calcul de* $H_m \tilde{y} = V_m^T r_0$

   (b) $x_0 = x_0 + V_m \tilde{y}$

   (c) $r_0 = r_0 - A V_m \tilde{y}$

6. *Redémarrage:*

   (a) *iter* = *iter* + 1

   (b) *Si* $(\|r_0\|/\beta < tol)$ *ou* (*iter* = *maxit*) *alors arrêt, sinon aller en* 2

Nous prenons en compte à présent la déflation, c'est-à-dire le cas où des paires de Ritz ont convergé. Nous utilisons les techniques de déflation de Lehoucq et Sorensen [30] pour la méthode **IRA**, qui conduisent dans notre cas à une technique de déflation implicite déjà utilisée par Saad dans [45] avec la méthode d'Arnoldi, pour la recherche de valeurs propres.

Avec déflation «réelle»

Nous supposons tout d'abord que c'est la première fois parmi tous les processus effectués jusqu'à présent, que des paires de Ritz convergent. Il y en a l et nous les notons $(\tilde{y}_i, \tilde{\mu}_i)_{1 \le i \le l}$. Les paires $(\tilde{y}_i, \tilde{\mu}_i)_{1 \le i \le l}$ satisfont:

$$A\tilde{y}_i - \tilde{\mu}_i \tilde{y}_i = h_{m+1,m} v_{m+1} e_m^{(m)T} \tilde{u}_i,$$

où le terme $|h_{m+1,m} e_m^{(m)T} \tilde{u}_i|$ est considéré comme négligeable. Nous notons les autres paires de Ritz du processus courant qui n'ont pas convergé $(y_i, \mu_i)_{l_1+1 \le i \le m-l_2}$ où les valeurs l_1 et l_2 sont explicitées ci-dessous. Parmi celles-ci, ceux qui nous intéressent sont celles numérotées de $l_1 + 1$ à k.

Nous séparons les l paires de Ritz qui ont convergé en deux groupes: les l_1 premières qui nous intéressent et les $l_2 = l - l_1$ autres qui ne nous intéressent pas. Nous supposons que $l_1 < k$ dans un premier temps.

Nous allons à présent déflater les l_1 valeurs de Ritz désirées et les l_2 qui ne le sont pas. Les déflater signifie qu'après avoir appliqué l'*implicit shifted QR-iteration* de la méthode

IRA, les colonnes de la matrice $\widehat{V}_k$ engendreront le sous-espace suivant

$$\text{Span}\{\tilde{y}_1, \ldots, \tilde{y}_{l_1}, y_{l_1+1}, \ldots, y_k\},$$

où le sous-espace $\text{Span}\{\widehat{V}_{l_1}\} = \{\tilde{y}_1, \ldots, \tilde{y}_{l_1}\}$ est un sous-espace invariant et $y_{l_1+1}, \ldots, y_k$ sont les vecteurs de Ritz du processus courant qui n'ont pas convergé et que nous aimerions approcher plus précisément. Nous aurons alors $A\widehat{V}_{l_1} = \widehat{V}_{l_1}T$ où T est une matrice quasi-triangulaire supérieure. Une matrice quasi-triangulaire supérieure est une matrice triangulaire par blocs, dont chaque bloc diagonal est soit un bloc 1×1, soit un bloc 2×2. Les blocs 2×2 sur sa diagonale, correspondent, dans notre cas, aux valeurs propres conjuguées du sous-espace invariant. Si nous étendons la factorisation d'Arnoldi jusqu'à ce qu'elle soit de longueur m, nous aurons alors que chaque nouveau vecteur $(\widehat{v}_i)_{k+1 \leq i \leq m}$ de la base orthonormale ainsi créé est orthogonal à ce sous-espace invariant. C'est exactement la déflation implicite effectuée par Saad pour la résolution des valeurs propres dans [45]. Elle est ici appliquée à la résolution de systèmes linéaires.

La phase de déflation débute avec ce que Lehoucq et Sorensen appellent le processus *Lock* dans [30]. Nous commencons par effectuer la factorisation QR de la matrice $[\tilde{y}_1, \ldots, \tilde{y}_l]$ soit :

$$Q \begin{pmatrix} R_l \\ 0 \end{pmatrix} = [\tilde{y}_1, \ldots, \tilde{y}_l] = \tilde{Y}_l,$$

où Q est une matrice $m \times m$ et R_l une matrice triangulaire supérieure de dimension $l \times l$. Cette factorisation QR se fait en utilisant des matrices de Householder. Nous appliquons alors Q à droite de l'équation (4.20) et obtenons :

$$A(V_m Q) = (V_m Q)(Q^T H_m Q) + h_{m+1,m} v_{m+1} e_m^{(m)T} Q. \tag{4.25}$$

La matrice $Q^T H_m Q$ a alors la forme suivante :

$$Q^T H_m Q = \begin{pmatrix} T & B \\ 0 & C \end{pmatrix},$$

où la matrice T de dimension $l \times l$ est quasi-triangulaire supérieure et correspond au sous-espace invariant $V_m Q[e_1^{(m)}, \ldots, e_l^{(m)}]$ des l vecteurs de Ritz qui ont convergé. La matrice $Q^T H_m Q$ n'est plus Hessenberg supérieure car la matrice C de dimension $(m-l) \times (m-l)$ est de forme quelconque. À l'aide de la matrice P de dimension $(m-l) \times (m-l)$, produit de matrices de Householder, nous restaurons la forme Hessenberg de C et donc de $Q^T H_m Q$. Nous mettons à jour les matrices ainsi :

$$H_m = \begin{pmatrix} I_l & 0 \\ 0 & P^T \end{pmatrix} Q^T H_m Q \begin{pmatrix} I_l & 0 \\ 0 & P \end{pmatrix}$$

et

$$V_m = V_m Q \begin{pmatrix} I_l & 0 \\ 0 & P \end{pmatrix}.$$

Comme les paires de Ritz correspondant aux vecteurs $\tilde{y}_1, \ldots, \tilde{y}_l$ ont convergé, nous avons $e_m^{(m)T} Q = e_m^{(m)T} + w^T$ avec $\|w\|_2$ une valeur négligeable [30]. De plus l'application de la

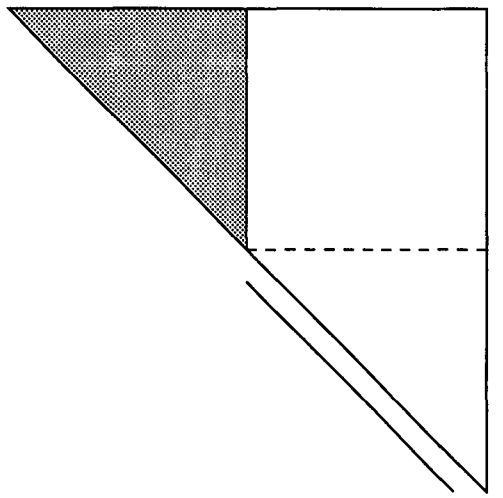


FIG. 4.1 – *La matrice H_m après avoir appliqué le processus Lock*

matrice P sur le vecteur $e_m^{(m)T}$ ne l'affecte pas et est égal à $e_m^{(m)T}$. La matrice H_m a, à présent, la même forme que la matrice de la figure 4.1, sa partie quasi-triangulaire supérieure étant de dimension $l \times l$. Nous avons encore $\text{Span}\{V_m Q[e_1^{(m)}, \ldots, e_l^{(m)}]\} = \{\tilde{y}_1, \ldots, \tilde{y}_l\}$.

Nous obtenons donc une factorisation d'Arnoldi de longueur m où le sous-espace invariant correspondant aux l paires de Ritz ayant convergé est mis en évidence.

Nous résumons ci-dessous la procédure **Lock** :

Algorithme 4.5 $[V_{m+1}, \bar{H}_m] = Lock(V_{m+1}, \bar{H}_m, \tilde{Y}_l)$

1. *Calcul de la factorisation QR :*

$$Q \begin{pmatrix} R_l \\ 0 \end{pmatrix} = \tilde{Y}_l$$

à l'aide des matrices de Householder. Q est de taille $m \times m$.

2. *Mise à jour de la factorisation :*
$H_m \leftarrow Q^T H_m Q$, $V_m \leftarrow V_m Q$

3. *Calcul de la matrice orthogonale P de dimension $(m-l) \times (m-l)$ en utilisant les matrices de Householder et qui restore la forme de Hessenberg supérieure de H_m*

4. *Mise à jour de la factorisation :*
$$H_m \leftarrow \begin{pmatrix} I_l & 0 \\ 0 & P^T \end{pmatrix} H_m \begin{pmatrix} I_l & 0 \\ 0 & P \end{pmatrix}, \; V_m \leftarrow V_m \begin{pmatrix} I_l & 0 \\ 0 & P \end{pmatrix}$$

Après avoir appliqué la procédure *Lock*, nous effectuons $m - (k + l_2)$ *implicit shifted QR-steps* en choisissant les *shifts* égaux aux $m - (k + l_2)$ valeurs de Ritz $\mu_{k+1}, \ldots, \mu_{m-l_2}$ qui ne nous intéressent pas. Nous obtenons la formule suivante :

$$A\widehat{V}_{k+l_2} = \widehat{V}_{k+l_2}\widehat{H}_{k+l_2} + \hat{h}_{k+l_2+1,k+l_2} v_{m+1} e_{k+l_2}^{(k+l_2)T}, \tag{4.26}$$

où la matrice $\widehat{H}_{k+l_2}$, de dimension $(k+l_2) \times (k+l_2)$, est de même forme que la matrice de la figure 4.1, sa partie triangulaire étant de dimension $l \times l$.

Afin de nous débarrasser des paires de Ritz qui ne nous intéressent pas, nous appliquons la phase appelée *Purge* dans [30]. Elle consiste tout d'abord à appliquer la phase *Lock* sur la dernière factorisation de longueur $k+l_2$ obtenue, de façon à ce que les matrices $\widehat{V}_{k+l_2}$ et $\widehat{H}_{k+l_2}$ résultantes vérifient :

$$\text{Span}\{\widehat{V}_{k+l_2}[e_1^{(k+l_2)}, \ldots, e_{l-l_1}^{(k+l_2)}]\} = \text{Span}\{\tilde{y}_{l_1+1}, \ldots, \tilde{y}_l\} \tag{4.27}$$

et

$$\text{Span}\{\widehat{V}_{k+l_2}[e_1^{(k+l_2)}, \ldots, e_l^{(k+l_2)}]\} = \text{Span}\{\tilde{y}_1, \ldots, \tilde{y}_l\}. \tag{4.28}$$

La matrice $\widehat{H}_{k+l_2}$ mise à jour, a la forme suivante :

$$\widehat{H}_{k+l_2} = \begin{pmatrix} \hat{T}_{l_2} & G_{l_2} \\ 0 & \widehat{H}_k \end{pmatrix},$$

où $\hat{T}_{l_2}$ correspond au sous-espace invariant $\text{Span}\{\tilde{y}_{l_1+1}, \ldots, \tilde{y}_l\}$ et $\widehat{H}_k$ est une matrice $k \times k$, dont la partie $l_1 \times l_1$ en haut à gauche est quasi-triangulaire supérieure.

Afin de transformer la matrice $\widehat{H}_{k+l_2}$ en une matrice diagonale par blocs, nous résolvons l'équation de Sylvester $Z\widehat{H}_k - \hat{T}_{l_2} Z = G_{l_2}$, où Z est une matrice de dimension $l_2 \times k$. Ainsi :

$$\widehat{H}_{k+l_2} \begin{pmatrix} I_{l_2} & Z \\ 0 & I_k \end{pmatrix} = \begin{pmatrix} I_{l_2} & Z \\ 0 & I_k \end{pmatrix} \begin{pmatrix} \hat{T}_{l_2} & 0 \\ 0 & \widehat{H}_k \end{pmatrix}$$

et l'équation (4.26) devient :

$$A\left(\widehat{V}_{k+l_2} \begin{pmatrix} I_{l_2} & Z \\ 0 & I_k \end{pmatrix}\right) = \left(\widehat{V}_{k+l_2} \begin{pmatrix} I_{l_2} & Z \\ 0 & I_k \end{pmatrix}\right) \begin{pmatrix} \hat{T}_{l_2} & 0 \\ 0 & \widehat{H}_k \end{pmatrix} + \hat{h}_{k+l_2+1,k+l_2} v_{m+1} e_{k+l_2}^{(k+l_2)T}.$$

En effectuant la factorisation QR de Z :

$$QR_k = \begin{pmatrix} Q_{l_2} \\ Q_k \end{pmatrix} R_k = \begin{pmatrix} Z \\ I_k \end{pmatrix}$$

et en ne gardant que les k dernières colonnes de l'équation précédente, celle-ci devient :

$$A(\widehat{V}_{k+l_2} Q) = (\widehat{V}_{k+l_2} Q) R_k \widehat{H}_k R_k^{-1} + \frac{1}{r_{k,k}^k} \hat{h}_{k+l_2+1,k+l_2} v_{m+1} e_k^{(k)T},$$

où $r_{k,k}^k$ est le $k^{\text{ième}}$ coefficient diagonal de la matrice R_k.

Nous mettons les données à jour ainsi :

$$\begin{cases} H_k = R_k \widehat{H}_k R_k^{-1} = R_k \widehat{H}_k Q_k \\ V_k = \widehat{V}_{k+l_2} Q \\ \\ h_{k+1,k} = \dfrac{1}{r_{k,k}^k} \hat{h}_{k+l_2+1,k+l_2}, \end{cases}$$

et obtenons finalement la factorisation d'Arnoldi suivante :

$$AV_k = V_k H_k + h_{k,k} v_{m+1} e_k^{(k)T},$$

où la matrice H_k est de même forme que sur la figure 4.1. La procédure *Purge* est détaillée ci-dessous :

Algorithme 4.6 $[V_{k+1}, \bar{H}_k] = Purge(\widehat{V}_{k+l_2+1}, \widehat{\bar{H}}_{k+l_2}, \tilde{Y}_{l_2})$

1. *Application de Lock :*
   $[\widehat{V}_{k+l_2+1}, \widehat{\bar{H}}_{k+l_2}] = Lock(\widehat{V}_{k+l_2+1}, \widehat{\bar{H}}_{k+l_2}, \tilde{Y}_{l_2})$
2. *Résolution de l'équation de Sylvester :*
   $$Z\widehat{H}_k - \widehat{T}_{l_2} Z = G_{l_2}$$
   où Z est une matrice de dimension $l_2 \times k$
3. *Calcul de la factorisation QR :*
   $$QR_k = \begin{pmatrix} Q_{l_2} \\ Q_k \end{pmatrix} R_k = \begin{pmatrix} Z \\ I_k \end{pmatrix}$$
   à l'aide des matrices de Householder. Q est de taille $(k + l_2) \times k$.
4. *Mise à jour de la factorisation :*
   $H_k \leftarrow R_k(\widehat{\bar{H}}_{k+l_2}[e_{l_2+1}^{(k+l_2)} \dots e_{k+l_2}^{(k+l_2)}]Q_k$, $V_k \leftarrow \widehat{V}_{k+l_2} Q$, $h_{k+1,k} = \widehat{h}_{k+l_2+1,k+l_2}/r_{k,k}^k$ *et* $v_{k+1} = v_{k+l_2+1}$

Nous mettons ensuite à jour $t = l_1$ la dimension du sous-espace invariant et complétons la factorisation à l'aide de p pas du processus d'Arnoldi, pour qu'elle soit de dimension m.

Supposons à présent que la dimension du sous-espace invariant soit $t < k$ et que dans le processus courant, u autres paires de Ritz aient convergé. Nous prenons alors en compte les $l = u + t$ paires de Ritz qui ont convergé et les séparons en deux groupes : les l_1 qui nous intéressent et les l_2 autres, avec l'hypothèse $l_1 < k$. Parmi les l_2, certaines peuvent appartenir au sous-espace invariant de dimension t du processus précédent. Ces paires de Ritz étaient intéressantes jusque là, mais ne le sont plus à présent. Les phase *Lock*, **IRA** et *Purge* s'effectuent alors comme précédemment mais sur de nouvelles données. La dimension t du sous-espace invariant est remis à jour en conséquence.

Regardons à présent le cas où nous avons $l_1 \geq k$. La taille du sous-espace invariant est t tel que $t < k$ et u autres paires de Ritz ont convergé dans le processus courant. Parmi les $l = u + t$ paires de Ritz qui ont convergé l_1 nous intéressent avec $l_1 \geq k$. Le procédé est alors simplifié, car nous obtenons un sous-espace invariant de la dimension désirée. Afin de le construire, nous appliquons l'algorithme 4.5 en prenant comme matrice $\tilde{Y}_l$ les k vecteurs de Ritz les plus intéressants parmi les l_1 désirés. Au sortir de l'algorithme,

nous ne gardons que les k premières colonnes de la factorisation d'Arnoldi, ce qui nous permet de ne garder que la partie corresondant au sous-espace invariant des k paires de Ritz les plus intéressantes ayant convergé. Nous ne procédons plus désormais et dans tous les processus suivants, aux phase de *Lock*, *implicit shifted QR-iteration* et *Purge*. Nous avons $AV_k = V_k T_k$ où la matrice T_k est quasi-triangulaire supérieure et orthogonalisons le résidu du processus précédent par rapport à ce sous-espace invariant pour former le vecteur v_{k+1} :

$$w = (I_n - V_k V_k^T)\frac{r_0}{\|r_0\|_2}$$
$$v_{k+1} = \frac{w}{\|w\|_2}.$$

Nous construisons le sous-espace de Krylov à l'aide de p pas du processus d'Arnoldi. Le sous-espace invariant reste le même jusqu'à, convergence de la solution. Dans l'ensemble des tests effectués, nous n'avons jamais atteint une déflation complète, c'est-à-dire t égal à k. Nous avons obtenu une déflation partielle, à savoir $t > 0$, mais généralement elle ne se produisait qu'à la fin de la procédure, lorsque la solution approchée avait presque convergé.

L'algorithme avec déflation est résumé ci-dessous :

Algorithme 4.7 $[x_0, r_0, V_{m+1}, \bar{H}_m, iter]$ = *IDFOM(A,x_0,b,m,k,maxit,tol)*

1. *Initialisation :*
   (a) $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$, $t = 0$
   (b) 1 *Processus de FOM :*
   $[x_0, r_0, V_{m+1}, \bar{H}_m]$ = *FOM(A,x_0,m)*
   (c) $iter = iter + 1$, *si* $(\|r_0\|_2/\beta < tol)$ *ou* $(iter = maxit)$ *alors arrêt*
2. *Si* $t < k$ *alors*
   (a) *Calcul des paires de Ritz* $(y_i = V_m u_i, \mu_i)$ *associées à la matrice* H_m *(celles du sous-espace invariant y compris)*
   (b) *Partage des paires de Ritz :*
      i. *Choix des* l_1 *paires de Ritz qui ont convergé et que nous désirons garder. En choisir au maximum k. Mettre à jour* $t = l_1$,
      ii. *Si* $t < k$, *partitionner les* $m - l_1$ *paires de Ritz restantes en :*
         - *les* l_2 *paires de Ritz qui ont convergé et que nous ne désirons pas garder,*
         - *les* $k - l_1$ *paires de Ritz qui n'ont pas convergé mais qui nous intéressent,*
         - *toutes les autres.*

      Sinon $l_2 = 0$.
      Fin Si
   (c) *Si* $l_1 + l_2 \neq 0$ *appliquer la phase* `Lock` :
   $[V_{m+1}, \bar{H}_m]$ = *Lock*$(V_{m+1}, \bar{H}_m, \tilde{Y}_l)$

(d) *Si* $t < k$

i. *Calcul de* $\bar{H}_{k+l_2}$ *et* V_{k+l_2+1} *les matrices résultante des* $m - k - l_2$ *pas de l'*`implicit shifted QR-iteration`,

ii. *Si* $l_2 \neq 0$ *appliquer la phase* `Purge` *:*
$[V_{k+1}, \tilde{H}_k] = Purge(\widehat{V}_{k+l_2+1}, \widehat{\bar{H}}_{k+l_2}, \tilde{Y}_{l_2})$.

Sinon calculer v_{k+1} *en orthonormalisant* r_0 *par rapport à* V_k $(AV_k = V_k T_k)$
Fin Si

Sinon calculer v_{k+1} *en orthonormalisant* r_0 *par rapport à* V_k $(AV_k = V_k T_k)$
Fin Si

3. *Compléter la factorisation d'Arnoldi de* $m - k$ *pas du processus d'Arnoldi*

4. *Calcul de la solution approchée :*

(a) *Calcul de* $H_m \tilde{y} = V_m^T r_0$

(b) $x_0 = x_0 + V_m \tilde{y}$

(c) $r_0 = r_0 - AV_m \tilde{y}$

5. *Redémarrage :*

(a) $iter = iter + 1$

(b) *Si* $(\|r_0\|/\beta < tol)$ *ou* $(iter = maxit)$ *alors arrêt, sinon aller en* 2.

Nous venons de voir une technique de déflation utilisant **IRA**, appliquée à l'algorithme de **FOM**. Nous voyons qu'à partir du même sous-espace de Krylov (4.21), nous pouvons améliorer aussi bien la solution approchée que la précision des paires de Ritz. Cette technique nous permet en outre d'atteindre et d'effectuer une déflation «réelle» implicite. Nous voyons dans la section suivante comment elle peut être adaptée à des méthodes de type **GMRES**.

4.3 Adaptation aux méthodes de type GMRES

Grâce à une légère modification de l'algorithme **IDFOM**, nous adaptons la technique de déflation vue dans la section précédente à deux autres méthodes: tout d'abord, la méthode **GMRES**, puis la méthode **MGMRES** qui se situe entre **FOM** et **GMRES**. Cette dernière sera explicitée lors de son utilisation avec la technique de déflation implicite.

4.3.1 GMRES

Pour adapter la technique vue précédemment à **GMRES**, nous n'allons plus considérer les paires de Ritz de H_m mais celles d'une autre matrice issue d'une projection oblique. Toutes les phases successives *Lock*, **IRA** et *Purge* s'appliquent à cette matrice. Nous obtenons l'équation (4.9) où $\hat{v}_1$ est *le* vecteur *idéal* combinaison linéaire des vecteurs de Ritz

désirés et associés à la matrice oblique. De la même façon que précédemment, le vecteur $\hat{v}_{k+1}$ est un muliple du résidu de **GMRES** du processus précédent, ce qui nous permet d'extraire une solution approchée de type **GMRES** de la factorisation d'Arnoldi du processus courant. La méthode résultante s'appelle **IDGMRES** pour **I**mplicitly Restarted and **D**eflated **GMRES**.

Nous considérons tout d'abord le cas où aucune paire de Ritz n'a convergé.

Sans déflation «réelle»

Nous exprimons le résidu de la méthode **IDGMRES**, r_m^{IDG}, en fonction de la matrice V_{m+1}. Il satisfait :

$$\begin{aligned} r_m^{IDG} &= V_{m+1}(V_{m+1}^T r_0 - \bar{H}_m y_m^{IDG}) && (4.29)\\ &= V_{m+1}Q_m^T(\tilde{\alpha}_{m+1}e_{m+1}^{(m+1)}) && (4.30)\\ &= V_{m+1}z_{m+1}^{IDG}, && (4.31) \end{aligned}$$

où $Q_m = \Omega_m \ldots \Omega_1$ est la matrice, produits des matrices de rotation de Givens, qui transforme $\bar{H}_m$ en une matrice triangulaire supérieure (voir chapitre 1). Nous notons $z_{m+1}^{IDG} = (\zeta_1^{IDG}, \ldots, \zeta_{m+1}^{IDG})^T$.

Proposition 4.1 *Tant que la solution n'a pas été trouvée, ζ_{m+1}^{IDG} n'est jamais nul.*

Preuve :

$$\zeta_{m+1}^{IDG} = s_m \tilde{\alpha}_{m+1},$$

où s_m est le sinus issu de la matrice de rotation de Givens Ω_m. Si $\tilde{\alpha}_{m+1}$ s'annule, $r_m^{IDG} = 0$ ce qui veut dire que la solution a déjà été trouvée. Si l'on suppose que $s_m = 0$ alors nous avons $h_{m+1,m} = 0$ et une fois encore, cela signifie que la solution a été trouvée et que la méthode a donc été arrêtée. Nous avons donc que, tant que la solution n'a pas été trouvée, ζ_{m+1}^{IDG} ne s'annule pas. ■

Cette proposition est valable aussi bien pour le tout premier processus où $r_0^{IDG} = \beta v_1$ que pour tous les autres où $r_0^{IDG} = V_{k+1}s_{k+1}$ avec s_{k+1} un vecteur appartenant à $\mathbb{R}^{k+1}$ tel que sa dernière composante soit non nulle, comme nous allons le voir dans la suite.

D'après la relation (4.31), nous pouvons exprimer v_{m+1} dans la base $((v_i)_{1\leq i\leq m}, r_m^{IDG})$:

$$v_{m+1} = \frac{1}{\zeta_{m+1}^{IDG}} r_m^{IDG} - \sum_{i=1}^{m} \frac{\zeta_i^{IDG}}{\zeta_{m+1}^{IDG}} v_i \qquad (4.32)$$

et la factorisation d'Arnoldi (4.2) devient :

$$\begin{aligned} AV_m &= V_m H_m + h_{m+1,m}\Big(\frac{1}{\zeta_{m+1}^{IDG}} r_m^{IDG} - \sum_{i=1}^{m} \frac{\zeta_i^{IDG}}{\zeta_{m+1}^{IDG}} v_i\Big)e_m^{(m)T} \\ &= V_m \underbrace{\begin{pmatrix} & & h_{1,m} - \frac{\zeta_1^{IDG}}{\zeta_{m+1}^{IDG}} h_{m+1,m} \\ & \bar{H}_{m-1} & \vdots \\ & & h_{m,m} - \frac{\zeta_m^{IDG}}{\zeta_{m+1}^{IDG}} h_{m+1,m} \end{pmatrix}}_{Hobl_m} + \frac{h_{m+1,m}}{\zeta_{m+1}^{IDG}} r_m^{IDG} e_m^{(m)T}. \qquad (4.33) \end{aligned}$$

$Hobl_m$ peut être vue comme la projection oblique de A sur Span$\{V_m\}$ orthogonalement à Span$\{W_m\}$, où les vecteurs w_i de W_m et les vecteurs v_i sont biorthonormaux. Nous avons en outre la condition supplémentaire que chaque vecteur w_i est orthogonal à r_m^{IDG} :

$$(w_i, v_j) = \delta_{i,j} \tag{4.34}$$
$$(w_i, r_m^{IDG}) = 0. \tag{4.35}$$

Une telle base de vecteurs w_i pourrait être $w_i = v_i - (\zeta_i^{IDG}/\zeta_{m+1}^{IDG})v_{m+1}$.
Nous obtenons :

$$Hobl_m = W_m^T A V_m = Hobl_m^0.$$

Comme cela a été fait pour **IDFOM** sur la matrice H_m, nous appliquons p *implicit shifted QR-steps* sur $Hobl_m$, en prenant comme *shifts* les valeurs propres $\mu_{k+1}, \ldots, \mu_m$ de la matrice $Hobl_m$, qui ne nous intéressent pas. Ces valeurs constituent donc des *shifts* «exacts». En gardant les notations du paragraphe précédent, nous obtenons :

$$Hobl_m^p = Q^T Hobl_m Q = Q_p^T \ldots Q_1^T Hobl_m Q_1 \ldots Q_p = R_p Q_p + \mu_{k+p} I_m$$

avec $Hobl_m^{p-1} - \mu_{k+p} I_m = Q_p R_p$. L'équation (4.33) devient :

$$A(V_m Q) = (V_m Q) Hobl_m^p + \frac{h_{m+1,m}}{\zeta_{m+1}^{IDG}} r_m^{IDG} e_m^T Q. \tag{4.36}$$

Q étant Hessenberg supérieure, $Hobl_m^p$ l'est aussi. Si nous partitionnons les matrices $V_m^p = V_m Q$ et $Hobl_m^p$ comme dans la section précédente, nous obtenons :

$$V_m^p = [\widehat{V}_{m-p}, \widetilde{V}_p]$$
$$Hobl_m^p = \begin{pmatrix} \widehat{Hobl}_{m-p} & \widetilde{M} \\ \widehat{\beta} e_1^{(p)} e_{m-p}^{(m-p)T} & \widetilde{Hobl}_p \end{pmatrix}.$$

Nous avons aussi

$$e_m^T Q = (0, \ldots, 0, \alpha_{m-p}, \ldots, \alpha_m).$$

Si nous ne gardons que les $k = m - p$ premières colonnes de (4.36), nous obtenons :

$$A\widehat{V}_k - \widehat{V}_k \widehat{Hobl}_k = \underbrace{(h_{m+1,m}\alpha_k r_m^{IDG} + \widehat{\beta}\widetilde{V}_p e_1^{(p)})}_{w} e_k^{(k)T}. \tag{4.37}$$

Proposition 4.2 *Si les* `shifts` *de l'*`implicit shifted QR-iteration` *sont pris égaux aux valeurs propres* $(\mu_{k+i})_{1 \le i \le p}$ *de la matrice* $Hobl_m$ *alors :*

- $\widehat{\beta} = 0$,
- *et* Span$\{\widehat{V}_k\}$ = Span$\{y_1, \ldots, y_k\}$,

où $(y_i)_{1 \le i \le k}$ *sont les vecteurs de Ritz associés aux valeurs de Ritz* $(\mu_i)_{1 \le i \le k}$ *de* $Hobl_m$ *qui ne sont pas prises en tant que* `shifts` *de l'*`implicit shifted QR-iteration`.

Preuve :

Nous démontrons d'abord la première affirmation.
$Hobl_m$ n'est pas singulière car elle peut s'exprimer comme le produit des deux matrices ci-dessous

$$\underbrace{\begin{pmatrix} 1 & & & -\zeta_1^{IDG}/\zeta_{m+1}^{IDG} \\ & \ddots & & \vdots \\ & & \ddots & \vdots \\ & & 1 & -\zeta_m^{IDG}/\zeta_{m+1}^{IDG} \end{pmatrix}}_{C} \times \bar{H}_m, \tag{4.38}$$

où C est une matrice $m \times (m+1)$ telle que $\mathrm{Ker}\{C\} = \mathrm{Span}\{z_{m+1}^{IDG}\}$.
Or $\mathrm{Ker}\{\bar{H}_m\} = \mathrm{Ker}\{H_m\} = \{0\}$ parce que la matrice H_m n'est pas singulière. Nous en déduisons que $\mathrm{Ker}\{Hobl_m\} = \{z \in \mathbb{R}^m / \ \bar{H}_m z = z_{m+1}^{IDG} \text{ ou } \bar{H}_m z = 0\}$. Résoudre l'équation $\bar{H}_m z = z_{m+1}^{IDG}$ est équivalent à résoudre $\bar{R}_m z = \tilde{\alpha}_{m+1} e_{m+1}^{(m+1)}$, mais avoir $\tilde{\alpha}_{m+1}$ non nul signifie qu'un tel z n'existe pas, et finalement que $\mathrm{Ker}\{Hobl_m\} = \{0\}$.

Comme μ_{k+1} est une valeur propre de $Hobl_m$, nous avons que la matrice $Hobl_m - \mu_{k+1} I_m$ est singulière. Mais $Hobl_m - \mu_{k+1} I_m = Q_1 R_1$ est aussi une matrice irréductible, dont les $m-1$ premières colonnes sont linéairement indépendantes. Nous avons donc $r_{i,i}^1 \neq 0$, pour $i = 1, \ldots, m-1$ et $r_{m,m}^1 = 0$, ce qui signifie que :

$$Hobl_m^1 = R_1 Q_1 + \mu_{k+1} I_m = \begin{pmatrix} \widehat{Hobl}_{m-1} & \widetilde{M} \\ 0 & \mu_{k+1} \end{pmatrix}.$$

La matrice $\widehat{Hobl}_{m-1}$ est aussi une matrice de Hessenberg irréductible. Nous construisons les matrices Q_1 et R_1 de la factorisation QR de telle sorte que les éléments de la sous-diagonale de $\widehat{Hobl}_{m-1}$ soient strictement positifs. Si nous appliquons de façon récursive ce que nous venons de démontrer pour $\widehat{Hobl}_m$ à la matrice $\widehat{Hobl}_{m-1}$, il s'ensuit que $\widehat{\beta} = 0$ et que $\widehat{Hobl}_p$ est une matrice triangulaire supérieure. Nous avons donc démontré la première affirmation.

Pour démontrer la seconde, nous devons d'abord prouver que chaque vecteur $Qe_j^{(m)}$ peut se mettre sous la forme $1/\tau_j \prod_{i=1}^{p} (Hobl_m - \mu_{k+i} I_m)$ multiplié par un vecteur, pour $j = 1, \ldots, m$.

- Pour $j = 1$:
  de l'équation $Hobl_m - \mu_{k+1} I_m = Q_1 R_1$, nous en déduisons que

  $$(Hobl_m - \mu_{k+1} I_m) e_1^{(m)} = r_{1,1}^1 Q_1 e_1^{(m)}$$

  et de $Hobl_m^j - \mu_{k+j+1} I_m = Q_j R_j$ que

  $$(Hobl_m^j - \mu_{k+j} I_m) e_1^{(m)} = r_{1,1}^j Q_j e_1^{(m)},$$

  ce qui est équivalent à l'équation suivante

  $$Q_{j-1}^T \ldots Q_1^T (Hobl_m - \mu_{k+j} I_m) Q_1 \ldots Q_{j-1} e_1^{(m)} = r_{1,1}^j Q_j e_1^{(m)}.$$

Finalement, nous obtenons

$$(Hobl_m - \mu_{k+j})Q_1 \dots Q_{j-1} e_1^{(m)} = r_{1,1}^j Q_1 \dots Q_{j-1} Q_j e_1^{(m)}.$$

Par récurrence, nous obtenons la propriété désirée pour $Qe_1^{(m)}$.

- Pour $j > 1$:
  si chaque vecteur $Q_1 e_l^{(m)}$ pour $l = 1, \dots, i-1$ peut se mettre sous la forme $(Hobl_m - \mu_{k+1} I_m) w_l$, il est alors évident que le vecteur $Q_1 e_i^{(m)}$ vérifie la même propriété.
  Supposons à présent que pour tout $i = 1, \dots, j-1$, les vecteurs $Q_1 \dots Q_i e_l^{(m)}$ pour $1 \leq l \leq m$ satisfassent la propriété désirée ainsi que les vecteurs $Q_1 \dots Q_j e_l^{(m)}$ pour $1 \leq l \leq k-1$, nous avons alors :

$$\begin{aligned} (Hobl_m^j - \mu_{k+j}) e_k^{(m)} &= r_{k,k}^j Q_j e_k^{(m)} + \textstyle\sum_{i=1}^{k-1} r_{i,k}^j Q_j e_i^{(m)} \\ (Hobl_m - \mu_{k+j}) Q_1 \dots Q_{j-1} e_k^{(m)} &= r_{k,k}^j Q_1 \dots Q_{j-1} Q_j e_k^{(m)} \\ &\quad + \textstyle\sum_{i=1}^{k-1} r_{i,k}^j Q_1 \dots Q_{j-1} Q_j e_i^{(m)}. \end{aligned}$$

Alors $Q_1 \dots Q_j e_k^{(m)}$ peut être mis sous la forme $\prod_{i=1}^{j} (Hobl_m - \mu_{k+i} I_m)$ multiplié par un vecteur. Par récurrence, nous avons que chaque vecteur $Qe_j^{(m)}$ pour $1 \leq j \leq m$ peut s'exprimer comme $\prod_{i=1}^{p} (Hobl_m - \mu_{k+i} I_m)$ multiplié par un vecteur.

Dénotons par $u_1, \dots, u_m$ les vecteurs propres de la matrice $Hobl_m$. Les vecteurs $Qe_i^{(m)}$ pour $1 \leq i \leq m$ ont donc été «purifiés» des vecteurs $u_{k+1}, \dots u_m$ et s'expriment uniquement à partir des vecteurs $u_1, \dots, u_k$. Puisque

$$Hobl_m Q(e_1^{(m)}, \dots, e_k^{(m)}) = Q(e_1^{(m)}, \dots, e_k^{(m)}) \widehat{Hobl}_k$$

nous avons

$$\dim(\mathrm{Span}\{Q(e_1^{(m)}, \dots, e_k^{(m)}\}) = k$$

et

$$\mathrm{Span}\{Q(e_1^{(m)}, \dots, e_k^{(m)})\} = \mathrm{Span}\{u_1, \dots, u_k\}$$

et finalement

$$\mathrm{Span}\{\widehat{V}_k\} = \mathrm{Span}\{y_1, \dots, y_k\}. \quad \blacksquare$$

Nous pouvons donc réécrire l'équation (4.37) ainsi :

$$A\widehat{V}_k = \widehat{V}_k \widehat{Hobl}_k + h_{m+1,m} \alpha_k r_m^{IDG} e_k^{(k)T}. \tag{4.39}$$

En notant w le vecteur r_m^{IDG}/β tel que $\beta = \|r_m^{IDG}\|_2$ et en l'orthogonalisant par rapport aux vecteurs colonnes de la matrice $\widehat{V}_k$, nous obtenons $\tilde{w} = w - \widehat{V}_k s_k$ avec

$$s_k = \widehat{V}_k^T w = \begin{pmatrix} \sigma_1 \\ \vdots \\ \sigma_k \end{pmatrix}.$$

Si nous appelons $\sigma_{k+1} = ||\tilde{w}||_2$ et $\widehat{v}_{k+1} = sign(\alpha_k)\ \tilde{w}/\sigma_{k+1}$, l'équation (4.39) se réécrit :

$$\begin{aligned} A\widehat{V}_k &= \widehat{V}_k\widehat{Hobl}_k + \\ &\quad h_{m+1,m}\alpha_k\beta(sign(\alpha_k)\sigma_{k+1}\widehat{v}_{k+1} + \textstyle\sum_{i=1}^{k}\sigma_i\widehat{v}_i)\ e_k^{(k)T} \\ &= \underbrace{\widehat{V}_k \begin{pmatrix} & \widehat{h_{obl}}_{1,k} + h_{m+1,m}\alpha_k\beta\sigma_1 \\ \widehat{Hobl}_{k-1} & \vdots \\ & \widehat{h_{obl}}_{k,k} + h_{m+1,m}\alpha_k\beta\sigma_k \end{pmatrix}}_{\widehat{H}_k} \\ &\quad + \underbrace{h_{m+1,m}|\alpha_k|\beta\sigma_{k+1}}_{\widehat{h}_{k+1,k+1}}\widehat{v}_{k+1}e_k^{(k)T}. \end{aligned}$$

Cette dernière équation est en fait une factorisation d'Arnoldi de longueur k. En effet :

- $\widehat{H}_k$ est une matrice de Hessenberg irréductible à sous diagonale strictement positive parce que la matrice $\widehat{Hobl}_k$ l'est,
- $[\widehat{V}_k, \widehat{v}_{k+1}]$ est une matrice orthonormale.

Nous pouvons compléter cette factorisation d'Arnoldi pour qu'elle soit de longueur m, grâce à p pas du processus d'Arnoldi. Nous obtenons alors, après avoir enlevé les accents circonflexes, la factorisation d'Arnoldi (4.2). Nous avons :

$$\begin{aligned} \text{Span}\{V_m\} &= \text{Span}\{v_1, \ldots, A^{m-1}v_1\} && (4.40) \\ &= \text{Span}\{y_1, \ldots, y_k, v_{k+1}, \ldots A^{p-1}v_{k+1}\}. && (4.41) \end{aligned}$$

Or v_{k+1} appartient au sous-espace $\text{Span}\{y_1, \ldots, y_k, r_m^{IDG}\}$ et d'après l'équation (4.33) $Ay_i = \mu_i y_i + h_{m+1,m}/\zeta_{m+1}^{IDG}(u_i)_m r_m^{IDG}$, chaque vecteur Ay_i appartient au sous-espace $\text{Span}\{y_i, r_m^{IDG}\}$. Nous avons alors que Av_{k+1} appartient à $\text{Span}\{y_1, \ldots, y_k, r_m^{IDG}, Ar_m^{IDG}\}$. Nous obtenons par récurrence :

$$\text{Span}\{V_m\} = \text{Span}\{y_1, \ldots, y_k, r_m^{IDG}, \ldots A^{p-1}r_m^{IDG}\}. \quad (4.42)$$

Nous avons aussi que le vecteur r_m^{IDG} appartient au sous-espace $\text{Span}\{y_i, Ay_i\}$ pour chaque i, ce qui par récurrence nous permet d'écrire :

$$\text{Span}\{V_m\} = \text{Span}\{y_1, \ldots, y_k, Ay_i, \ldots A^p y_i\} \quad (4.43)$$

pour $i = 1, \ldots, k$.

Comme nous l'avons fait pour la méthode **IDFOM**, nous pouvons à présent extraire une solution de type **GMRES** du sous-espace $\text{Span}\{V_m\}$, soit :

$$x_m^{IDG} = x_0 + V_m y_m^{IDG}$$

avec

$$y_m^{IDG} = \arg\min_{y\in\mathbb{R}^m} ||\beta(s_k^T, sign(\alpha_k)\sigma_{k+1}, 0\ldots, 0)^T - \bar{H}_m y||_2.$$

D'après la relation (4.40), la projection est faite sur un sous-espace de Krylov, ce qui fait de cette méthode une méthode de type **RM** à part entière. Par rapport à la méthode Augmented **GMRES** de Morgan, cette propriété nous permet de gagner en espace mémoire, puisque nous ne stockons plus les vecteurs $y_1, \ldots, y_k$. Nous voyons de plus, qu'à partir du même sous-espace (4.40), nous extrayons la solution **GMRES** et approchons également de façon plus précise les vecteurs propres

Avec déflation «réelle»

Lorsque des paires de Ritz convergent, nous appliquons le procédé que nous avons vu à la fin du paragraphe précédent, notamment les trois phases de *Lock*, *implicit shifted QR-iteration* et *Purge*. La seule différence réside dans le fait que les vecteurs et valeurs propres de $Hobl_m$ sont considérés au lieu de ceux de H_m et que tout le procédé est appliqué à cette matrice.

L'algorithme final est décrit ci-dessous :

Algorithme 4.8 $[x_0, r_0, V_m, \bar{H}_m, iter]$ = *IDGMRES(A,*x_0*,b,m,k,maxit,tol)*

1. *Initialisation :*
   (a) $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$, $t = 0$
   (b) 1 *Processus de GMRES :*
   $[x_0, r_0, V_{m+1}, \bar{H}_m]$ = *GMRES(A,*x_0*,m)*
   (c) $iter = iter + 1$, *si* $(\|r_0\|_2/\beta < tol)$ *ou* $(iter = maxit)$ *alors arrêt*
2. *Si* $t < k$ *alors*
   (a) *Calcul de* $Hobl_m$
   (b) *Calcul des paires de Ritz* $(y_i = V_m u_i, \mu_i)$ *associées à la matrice* $Hobl_m$ *(y compris celles du sous-espace invariant)*
   (c) *Partage des paires de Ritz :*
      i. *Choix des* l_1 *paires de Ritz qui ont convergé et que nous désirons garder. En choisir au maximum k. Mettre à jour* $t = l_1$,
      ii. *Si* $t < k$, *partitionner les* $m - l_1$ *paires de Ritz restantes en :*
         - *les* l_2 *paires de Ritz qui ont convergé et que nous ne désirons pas garder,*
         - *les* $k - l_1$ *paires de Ritz qui n'ont pas convergé mais qui nous intéressent,*
         - *toutes les autres.*

      Sinon $l_2 = 0$.
      Fin Si
   (d) *Si* $l_1 + l_2 \neq 0$ *appliquer la phase* `Lock` *:*
   $[V_{m+1}, \bar{H}_m]$ = *Lock(*$V_{m+1}, \bar{Hobl}_m, \tilde{Y}_l$*)*

(e) Si $t < k$

i. Calcul de $\bar{Hobl}_{k+l_2}$ *et* V_{k+l_2+1} *les matrices résultante des* $m-k-l_2$ *pas de l'*`implicit shifted QR-iteration`*,*

ii. Si $l_2 \neq 0$ *appliquer la phase* `Purge` *:*
$[V_{k+1}, \bar{Hobl}_k] = Purge(\widehat{V}_{k+l_2+1}, \widehat{\bar{Hobl}}_{k+l_2}, \tilde{Y}_{l_2})$,

iii. Calcul de v_{k+1} *et* H_k *à partir de* $Hobl_k$.

Sinon calculer v_{k+1} *en orthonormalisant* r_0 *par rapport à* V_k $(AV_k = V_k T_k)$.
Fin Si

Sinon calculer v_{k+1} *en orthonormalisant* r_0 *par rapport à* V_k $(AV_k = V_k T_k)$
Fin Si

3. *Compléter la factorisation d'Arnoldi de* $m-k$ *pas du processus d'Arnoldi*

4. *Calcul de la solution approchée :*

(a) Résoudre $\tilde{y} = \arg\min \|V_{m+1}^T r_0 - \bar{H}_m y\|_2$

(b) $x_0 = x_0 + V_m \tilde{y}$

(c) $r_0 = r_0 - AV_m \tilde{y}$

5. *Redémarrage :*

(a) $iter = iter + 1$

(b) Si $(\|r_0\|/\beta < tol)$ *ou* $(iter = maxit)$ *alors arrêt, sinon aller en 2.*

4.3.2 MGMRES

Dans ce paragraphe, nous introduisons une nouvelle méthode **IDMGMRES** qui se situe mathématiquement entre **IDFOM** et **IDGMRES**. Nous calculons une solution approchée de telle sorte que la norme du résidu correspondant soit minimale au sens de la norme 2, avec comme contrainte supplémentaire que ce résidu soit orthogonal au résidu du processus précédent.

Nous détaillons tout d'abord la méthode elle-même, à savoir **MGMRES**, à laquelle nous associons ensuite la technique de redémarrage et de déflation implicite. La méthode finale est **IDMGMRES** pour Implicitly Restarted and Deflated **MGMRES**.

MGMRES

Nous calculons une solution approchée de la forme $x_m^{MG} = x_0 + V_m y_m^{MG}$ telle que :

$$\begin{aligned} y_m^{MG} \;=\; \arg \;& \min \|r_0 - AV_m y\|_2 \\ & y \in \mathbb{R}^m \\ & (r_0, r_0 - AV_m y) = 0, \end{aligned}$$

ce qui est équivalent à résoudre

$$\begin{aligned} y_m^{MG} \quad = \quad \arg \quad & \min \|\beta e_1^{(m+1)} - \bar{H}_m y\|_2^2. \\ & y \in \mathbb{R}^m \\ & \textstyle\sum_{i=1}^m h_{1,i} y_i = \beta. \end{aligned}$$

Cette équation n'est rien d'autre qu'un problème de moindres carrés linéaire avec contraintes. La solution de ce problème, notée (y_m^{MG}, λ^{MG}) appartenant à $\mathbb{R}^m \times \mathbb{R}$, satisfait :

$$\left\{ \begin{array}{rcl} \bar{H}_m^T \bar{H}_m y_m^{MG} & = & \bar{H}_m^T \, \beta(1+\lambda^{MG}) e_1^{(m+1)} \\ \displaystyle\sum_{i=1}^m h_{1,i} \, (y_m^{MG})_i & = & \beta. \end{array} \right. \tag{4.44}$$

La première condition de (4.44) se traduit par :

$$y_m^{MG} = (1+\lambda^{MG}) \; R_m^{-1} (Q_m \beta e_1^{(m+1)})_{1:m} = (1+\lambda^{MG}) R_m^{-1} (\gamma_1, \ldots, \gamma_m)^T,$$

où Q_m est la matrice produit de rotations de Givens (1.18), qui transforme la matrice $\bar{H}_m$ en la matrice triangulaire supérieure $\bar{R}_m$. Les coefficients γ sont définis par l'équation (1.19) du chapitre 1. Cette dernière relation signifie aussi que :

$$y_m^{MG} = (1+\lambda^{MG}) \; y_m^G \tag{4.45}$$

avec y_m^G le vecteur correspondant de la méthode **GMRES**. La deuxième condition de (4.44) implique que :

$$1 + \lambda^{MG} = \beta / \sum_{i=1}^m h_{1,i} \, (y_m^G)_i. \tag{4.46}$$

La paire (y_m^{MG}, λ^{MG}) est donc aisément calculable et son utilisation n'induit que très peu de changement dans le code originel de la méthode **GMRES**. En fait, nous ne calculons jamais le terme λ^{MG} et mettons à jour le vecteur y_m^{MG} à partir de y_m^G de la façon suivante :

$$y_m^{MG} = \left(\; \beta / \textstyle\sum_{i=1}^m h_{1,i} \, (y_m^G)_i \; \right) \; y_m^G. \tag{4.47}$$

L'algorithme de **MGMRES** est décrit ci-dessous :

Algorithme 4.9 $[x_0, r_0, V_{m+1}, \bar{H}_m, iter]$ = *MGMRES(A,x_0,b,m,maxit,tol)*

1. *Initialisation :* $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, $iter = 0$
2. *Construction du sous-espace de Krylov :* $[V_{m+1}, \bar{H}_m]$ = *Arnoldi(A,r_0,m)*
3. *Calcul de la solution approchée :*
   (a) $\tilde{y} = \arg\min_y \|\beta e_1^{(m+1)} - \bar{H}_m y\|_2$
   (b) $\alpha = \beta / \sum_{i=1}^m h_{1,i} \tilde{y}_i$ *et* $\tilde{y} = \alpha \tilde{y}$
   (c) $x_0 = x_0 + V_m \tilde{y}$

(d) $r_0 = r_0 - AV_m\tilde{y}$

4. Redémarrage :

(a) $iter = iter + 1$

(b) si $(\|r_0\|_2/\beta < tol)$ *ou si* $(iter = maxit)$ *alors arrêt sinon aller en* 2

D'un point de vue polynomial, la méthode **MGMRES** signifie que nous construisons un résidu $r_m^{MG} = r_0 - V_m y_m^{MG}$ qui vérifie les équations suivantes :

$$\begin{cases} (AV_m)^T(r_m^{MG} - \tilde{\lambda} r_0) = 0 \\ r_0^T r_m^{MG} = 0, \end{cases} \tag{4.48}$$

soit encore :

$$\begin{cases} r_m^{MG} - \tilde{\lambda} r_0 \perp AK_m(A, r_0) \\ r_0^T r_m^{MG} = 0. \end{cases} \tag{4.49}$$

Nous associons à présent cet algorithme à la technique de redémarrage et de déflation implicite.

IDMGMRES

De la même façon que pour la méthode **MGMRES**, nous calculons tout d'abord le vecteur y_m^{IDG} de la méthode **IDGMRES**, puis mettons à jour le vecteur y_m^{IDMG} de **IDMGMRES**, par une opération similaire à l'équation (4.45).

Le vecteur initial r_0 de chaque processus s'exprime à présent ainsi :

$$r_0 = V_{m+1} \underbrace{(\tau_1, \ldots, \tau_{k+1}, 0 \ldots, 0)^T}_{t_{m+1}^T}, \tag{4.50}$$

où le coefficient τ_{k+1} n'est jamais nul.

La solution $(y_m^{IDMG}, \lambda^{IDMG})$ satisfait à présent les équations suivantes :

$$\begin{cases} \bar{H}_m^T \bar{H}_m y_m^{IDG} = \bar{H}_m t_{m+1} \\ 1 + \lambda^{IDMG} = \|t_{m+1}\|^2 / \sum_{j=1}^{k+1} \tau_j \sum_{i=1}^{m} h_{j,i} (y_m^{IDG})_i \\ y_m^{IDMG} = (1 + \lambda^{IDMG}) y_m^{IDG}. \end{cases} \tag{4.51}$$

D'après ce qui a été dit dans le paragraphe précédent, le redémarrage implicite n'est possible que si

$$r_m^{IDMG} = V_{m+1} z_{m+1}^{IDMG}$$

tel que $\zeta_{m+1}^{IDMG} \neq 0$ avec $z_{m+1}^{IDMG} = (\zeta_1^{IDMG}, \ldots, \zeta_{m+1}^{IDMG})^T$. Or

$$\zeta_{m+1}^{IDMG} = (1 + \lambda^{IDMG})\ \zeta_{m+1}^{IDG}.$$

Du fait de la proposition 4.1, ζ_{m+1}^{IDG} ne s'annule jamais à moins que la solution de **IDGMRES** n'ait été trouvée. Cela veut également dire que la méthode **IDMGRES** a trouvé la solution. En effet, nous avons alors $t_{m+1} = \bar{H}_m y_m^{IDG}$ et $1 + \lambda^{IDMG} = 1$, ce qui implique que $x_m^{IDMG} = x_m^{IDG}$ est la solution du système linéaire. Quand un tel cas apparaît, nous ne construisons pas la matrice $Hobl_m$, puisque le procédé est arrêté.

Nous savons en outre que le coefficient λ^{IDMG} n'est jamais égal à -1. En effet, dans le cas contraire, nous aurions $\sum_{j=1}^{k+1} \tau_j \sum_{i=1}^{m} h_{j,i}(y_m^{IDG})_i = \infty$ ou bien $t_{k+1} = 0$, ce qui est impossible.

Afin que les équations (4.40-4.43) soient satisfaites pour la méthode **IDMGMRES**, les hypothèses de la proposition 4.2 doivent être remplies, c'est-à-dire que la matrice $Hobl_m$ issue de l'équation (4.33) où l'exposant **IDG** a été remplacé par l'exposant **IDMG** ne doit pas être singulière. Pour cela, nous exprimons la matrice $Hobl_m$ comme le produit des matrices C définie par la relation (4.38) et $\bar{H}_m$ où $\mathrm{Ker}\{C\} = \{z_{m+1}^{IDMG}\}$. Résoudre $\bar{H}_m z = z_{m+1}^{IDMG}$ est équivalent à résoudre

$$\bar{R}_m z = (\psi_1, \ldots, \psi_{m+1})^T$$

où $\psi_{m+1} = (1 + \lambda^{IDMG})\ \tilde{\alpha}_{m+1}$. Or $\tilde{\alpha}_{m+1} \neq 0$ et $\lambda^{IDMG} \neq -1$, ce qui signifie qu'un vecteur z satisfaisant l'équation précédente ne peut exister. La matrice $Hobl_m$ n'est donc pas singulière et l'algorithme de déflation peut lui être appliqué.

L'implantation de **IDMGMRES** est la même que celle de **IDGMRES**, excepté que le calcul de y_m^{IDMG} nécessite de le multiplier par un scalaire.

Cette très légère modification induit des accélérations parfois importantes.

4.4 Comparaison des coûts de calculs et de stockage

Nous comparons à présent les méthodes **GMRES**, **FOM AGMRES**, **IDFOM**, **IDGMRES**, **IDMGMRES**, mais aussi **Bi-CGSTAB** et **QMR**, qui font partie des méthodes testées, d'un point de vue coût de stockage et coût des calculs. Nous comparons les méthodes redémarrées pour une même dimension m de sous-espace de Krylov construit et les méthodes non redémarrées comme **Bi-CGSTAB** et **QMR** pour une itération effectuée, qui permet de passer du calcul de x_k à celui de x_{k+1}.

4.4.1 Coûts de stockage

Comme nous supposons que les matrices concernées sont de grande taille, nous faisons l'hypothèse que m est très petit devant n. Nous ne tiendrons donc pas compte des tableaux de taille de l'ordre de m.

Les méthodes **GMRES** et **FOM** requièrent un coût de stockage comparable. Il en est de même pour les méthodes **IDFOM**, **IDGMRES** et **IDMGMRES**. Toutes ces méthodes seront donc considérées ensemble pour ce qui est de leur coût de stockage.

Elles nécessitent le stockage de la matrice V_{m+1} de dimension $n \times (m+1)$ ainsi que 3

vecteurs supplémentaires de dimension n, le second membre b, la solution approchée x et le résidu correspondant r.

En ce qui concerne la méthode **AGMRES**, nous considérons deux implantations différentes, pour lesquelles nous construisons le même sous-espace $S_{m,k}(A, r_0, y_1, \ldots, y_k)$.

La première implantation que nous dénotons **AGMRES+** nécessitent $m + 4 + 2k$ vecteurs de stockage :

- les 3 vecteurs b, x et r,
- les $m + 1$ vecteurs colonnes de la matrice V_{m+1},
- les k vecteurs de Ritz $y_1, \ldots, y_k$ qui permettent de former la matrice $W_m = [V_{m-k}, y_1, \ldots, y_k]$,
- mais aussi les k vecteurs $Ay_1, \ldots, Ay_k$.

Le fait de stocker les vecteurs $(Ay_i)_{1 \leq i \leq k}$ nous permet de réduire le nombre de produits matrice vecteur avec la matrice A. En effet, puisque y_i est un vecteur de Ritz, il est de la forme $W_m u_i$ où u_i est un vecteur propre de H_m. Il satisfait donc l'équation :

$$Ay_i = AW_m u_i = V_{m+1} \bar{H}_m u_i.$$

Et, au lieu d'effectuer k produits matrice vecteur avec A, nous calculons k produits matrice vecteur avec la matrice V_{m+1}, ce qui est beaucoup moins coûteux. Cela oblige en contrepartie à stocker les k vecteurs résultants, qui seront utilisés dans le processus suivant.

Nous appelons **AGMRES** la seconde implantation de la même méthode, qui ne stocke pas les vecteurs $(Ay_i)_{1 \leq i \leq k}$, mais calcule les k produits matrice vecteur à la place. Elle permet donc de sauvegarder la place de k vecteurs de dimension n et nécessite au total le stockage de $m + 4 + k$ vecteurs.

Le stockage de l'ensemble des méthodes testées est résumé ci-dessous.

Stockage	FOM	AGMRES	AGMRES+	Bi-CGSTAB	QMR
Vecteur de dimension n	$m + 4$	$m + 4 + k$	$m + 4 + 2k$	8	10

Les nouvelles méthodes nécessitent donc un coût de stockage comparable à **GMRES** et **FOM**. Elle nécessitent en fait quelques tableaux supplémentaires de dimension $m \times m$, qui sont négligeables. Les méthodes **AGMRES** et **AGMRES+**, en revanche, nécessitent beaucoup plus de stockage que toutes les autres méthodes testées.

Les méthodes **Bi-CGSTAB** et **QMR**, quant à elles, ont un coût qui ne dépend pas de m, car ce ne sont pas des méthodes redémarrrées. Leur stockage n'est pas onéreux.

4.4.2 Coûts de calculs

Dans ce paragraphe, nous comparons les coûts de calculs pour les différentes méthodes testées.

Comme nous venons de le faire pour les coûts de stockage, nous faisons aussi l'hypothèse que $m \ll n$ pour le calcul de la complexité en temps. Nous ne considérons pas les opérations d'ordre m, qui seront effectuées de façon redondante sur les processeurs de la machine parallèle.

Les méthodes **GMRES** et **FOM** ont une complexité égale. Les méthodes **IDFOM**, **IDGMRES** et **IDMGMRES** ont également une complexité égale et sont regroupées toutes les trois.

Pendant un processus, **GMRES** et **FOM** effectuent :

- dans le processus d'Arnoldi :
  - $(m+1)(m+2)/2$ produits scalaires de dimension n,
  - $m(m+1)/2$ opérations triadiques sur des vecteurs de dimension n,
  - m produits matrice vecteur avec A,
- lors du calcul de la solution approchée :
  - 2 produits matrice vecteur de dimensions $(n \times m)$ par m.

Les méthodes **IDFOM**, **IDGMRES** et **IDMGMRES** nécessitent k produits matrice-vecteur de moins que les méthodes **FOM** et **GMRES**, ceci grâce à la méthode **IRA**. De la même façon, comme la matrice V_{k+1} est construite grâce à l'*implicit shifted QR-iteration*, ceci nous permet d'économiser $k(k+1)/2$ produits matrice vecteur de dimension n ainsi que les $(k-1)k/2$ opérations triadiques de dimension n correspondantes. Ces gains sont importants, puisqu'ils nous permettent, pour une même taille de sous-espace, de diminuer les produits matrice vecteur, mais aussi les produits scalaires, source d'une perte de temps dans le processus d'Arnoldi (voir chapitre 3). C'est essentiellement la méthode **IRA** qui nous permet de réduire leur nombre. Ces méthodes nécessitent en outre :

- 2 produits matrice vecteur de dimensions $(n \times m)$ par m, dans le calcul de la solution approchée, comme c'est le cas pour **FOM** et **GMRES**,
- $2(m-k)$ produits matrice vecteur de dimensions $(n \times s)$ par s, si la phase *Lock* a lieu, c'est-à-dire si des paires de Ritz ont convergé : s est de l'ordre de m ou k, selon le nombre de paires de Ritz qui ont convergé,
- $2(m+1)$ opérations triadiques de dimension n, lors de l'*implicit shifted QR-iteration*,
- $2m-k$ produits matrice vecteur de dimensions $(n \times s)$ par s, si la phase *Purge* a lieu, c'est-à-dire si des paires de Ritz non désirées ont convergé.

Il faut souligner que les phases *Lock* et *Purge* sont assez rares et n'apparaissent que dans les derniers processus, lorsque la solution approchée a pratiquement convergé. Ils ne doivent pas être pris en compte pour la majorité des processus.

La méthode **AGMRES+** effectue à peu près le même nombre d'opérations que **FOM** et **GMRES** au sein du processus d'Arnoldi, excepté qu'elle fait k produits matrice vecteur avec la matrice A en moins. En plus de ceci, elle effectue les opérations suivantes :

- pour former la matrice $W_m^T A^T W_m$:
  - $m-k$ produits matrice vecteur de dimensions $(n \times m)$ par m,
  - $(m-k) \times k$ produits scalaires de dimension n,
- pour calculer les vecteurs $(y_i)_{1 \leq i \leq k}$ et $(Ay_i)_{1 \leq i \leq k}$:
  - $2k$ produits matrice vecteur de dimensions $(n \times m)$ par m.

La méthode **AGMRES** effectue, par rapport à **AMGRES+**, k produits matrice vecteur avec la matrice A de plus et k produits matrice vecteur de dimensions $(n \times m)$ par m de moins. Au lieu de stocker les vecteurs $(Ay_i)_{1 \leq i \leq k}$ calculés à moindre coût dans le processus précédent, la méthode **AGMRES** fait le calcul réel du produit matrice vecteur avec la matrice A dans le processus courant.

Toutes les opérations sont résumées ci-dessous :

Calcul	FOM	IDFOM	AGMRES / AGMRES+
Produit mat-vec avec A	m	$m-k$	m / $m-k$
DDOT de longueur n	$\frac{(m+1)(m+2)}{2}$	$\frac{(m+1)(m+2)-k(k+1)}{2}$	$\frac{(m+1)(m+2)}{2} + (m-k)k$
DAXPY de longueur n	$\frac{m(m+1)}{2}$	$\frac{(m+1)(m+6)-(k-1)k}{2}$	$\frac{m(m+1)}{2}$
Produit matrice vecteur de dimensions $(n \times s)$ par s	2	1) 2 2) $2+2m-2k$ 3) $2+4m-3k$	$2+m$ / $2+m+k$

Nous voyons donc que pour une même dimension du sous-espace, les méthodes implicitement redémarrées et déflatées ainsi que **AGMRES+** permettent de réduire le nombre des produits matrice vecteur, qui sont l'une des opérations les plus coûteuses. Les méthodes implicitement déflatées permettent en outre de diminuer les produits scalaires du processus d'Arnoldi, qui constituent une perte de temps importante en environnement parallèle, et fait l'objet du chapitre 3. Ces gains sont compensés par quelques produits matrice matrice de taille $(n \times s) \times s$, qui sont, en fait, effectués au travers de calculs locaux n'induisant aucune communication. Lorsqu'aucune déflation n'apparaît, ce qui est le cas le plus fréquent, ces produits matrice matrice sont au nombre de 2 et correspondent en fait à la mise à jour du vecteur solution et du résidu correspondant, comme pour **FOM** et **GMRES**. Dans ce cas, il n'y a pas de coût compensant la diminution du nombre des produits matrice vecteur et des produits scalaires. Dans le cas où, à la fois, les phases

Lock et *Purge* ont lieu, elles induisent un coût de $(2+4m-3k) \times n(2s-1)$ opérations, ce qui est supérieur au $k(k+1)(2n-1)/2$ opérations des $k(k+1)/2$ produits scalaires que nous n'effectuons pas. Mais elles n'induisent aucune communication, qui faite en séquence dans le processus d'Arnoldi est source de perte de temps. De plus, nous n'avons pas pris en compte le coût des k produits scalaires que nous ne faisons pas.

Les méthodes **AGMRES**, quant à elles, induisent des produits scalaires supplémentaires par rapport aux méthodes implicitement redémarrées et surtout doivent faire un compromis entre le stockage et le calcul. La méthode Augmented **GMRES** la moins coûteuse en stockage est cependant plus onéreuse que toutes les autres méthodes prises en compte.

Nous avons donc finalement, au vu de ce tableau, que la complexité des méthodes implicitement redémarrées et déflatées est inférieure à celles des autres méthodes et plus adaptée aux machines parallèles. À nombre de processus égal pour atteindre la convergence, elles doivent permettre de converger plus rapidement en temps d'exécution. Ces méthodes sont donc prometteuses pour les machines parallèles, en comptant que, en nombre de processus pour converger, elles se comportent aussi bien que les méthodes **FOM** et **GMRES**.

En ce qui concerne les méthodes **Bi-CGSTAB** et **QMR**, leur coût en calcul est répertorié, pour une itération, dans le tableau suivant:

Calcul	Bi-CGSTAB	QMR
Produit mat-vec avec A	2	1
Produit mat-vec avec A^T	0	1
DDOT de longueur n	6	3
DAXPY de longueur n	6	6

Si nous comparons $m/2$ itérations des méthodes **Bi-CGSTAB** et **QMR**, qui correspondent à un nombre égal de produits matrice vecteur d'un processus des méthodes **FOM** et **GMRES**, nous nous apercevons que leur complexité est nettement inférieure. Elles sont donc excessivement rapides en environnement parallèle.

4.5 Résultats numériques

Les tests n'ont été effectués qu'en Matlab et visent uniquement à analyser le comportement numérique des nouvelles méthodes. Il est en effet nécessaire, avant de passer à la parallélisation d'une méthode de s'assurer, en séquentiel, qu'elle présente un comportement numérique intéressant par rapport aux méthodes existantes. La parallélisation de cette méthode fera l'objet d'un travail ultérieur.

Chaque test suit le même schéma: le second membre est pris égal au vecteur $(1,\ldots,1)^T$ et le vecteur initial est le vecteur nul. La procédure de test est arrêtée dès que $\|r\|_2/\beta <$ tol avec $tol = 1.0e-9$ ou bien lorsque le nombre maximum de processus $maxproc = 200$ est atteint. Dans les tables suivantes, m est la dimension du sous-espace

d'Arnoldi, c'est-à-dire la longueur de la factorisation d'Arnoldi, et k est le nombre de vecteurs de Ritz pris en compte dans chaque processus des méthodes suivantes : **IDFOM**, **IDGMRES**, **IDMGMRES**, **AMGRES** et **AGMRES+**. Comme des valeurs propres conjuguées peuvent survenir, ce nombre peut en fait varier entre k et $k+1$ suivant le processus. Il est remis à jour à chaque redémarrage. Les termes *Proc* et *Reduct.* montrent respectivement le nombre de processus effectués et la réduction de la norme résiduelle atteinte $\|r\|_2/\beta$ lorsque la procédure s'arrête. Le terme *Matvec.* correspond au nombre total de produits matrice vecteur avec la matrice A effectués pendant toute la procédure. Le paramètre *tolvp* est la tolérance utilisée dans le critère de convergence des paires de Ritz. Nous considérons qu'une paire de Ritz (y_i, μ_i) a convergé si elle satisfait :

$$\frac{\|Ay_i - \mu_i y_i\|_2}{\|y_i\|_2} < tolvp$$

Nous avons pris *tolvp* égale à $1.0e-06$.

Bien que nos méthodes soient dédiées à des problèmes de grande dimension, nous ne présentons ici que des tests de matrices de dimensions relativement faibles. L'hypothèse faite dans la section précédente que $m \ll n$ n'est pas vérifiée. Mais comme nous nous intéressons au comportement numérique des méthodes, nous nous concentrons essentiellement sur les opérations les plus coûteuses, détaillées dans le paragraphe précédent, lorsque les matrices sont de grandes dimensions. Nous mesurons le coût des méthodes en nombre de processus et de produits matrice vecteur avec la matrice A.

Nous présentons trois matrices. La première est utilisée dans l'article de Morgan [34] pour comparer sa méthode Augmented **GMRES** à la méthode **GMRES**. Les deux autres proviennent de la collection Harwell-Boeing [11].

Pour chacun des tests effectués, nous comparons les méthodes **IDFOM**, **IDGMRES** et **IDMGMRES** avec les méthodes **GMRES**, **FOM**, **AGMRES** et **AGMRES+**, pour une dimension m de sous-espace comparable. Celle-ci est prise égale successivement aux valeurs suivantes $20, 30, 40$ et 50, tandis que k est égal à $1, 3$ ou 6. Nous avons aussi fait des comparaisons avec les méthodes **Bi-CGSTAB** et **QMR**. Cette dernière est implantée dans sa version originelle, à savoir sans *look-ahead* et avec des produits matrice vecteur avec A^T. Pour ces deux dernières méthodes, le nombre maximal d'itérations est fixé à 2000, tout comme pour les versions non redémarrées de **FOM** et **GMRES**.

Dans la suite, nous appelons $nb_{methode}$ le nombre de produit matrice vecteur avec la matrice A (y compris ceux avec la matrice A^T si c'est le cas) effectué par la méthode *methode* pour converger.

Première matrice : la première matrice montre les problèmes posés par le redémarrage des méthodes **FOM** et **GMRES** lorsque la matrice définie positive a ses cinq plus petites valeurs propres d'ordre de grandeur légèrement différent de toutes les autres. La matrice est bidiagonale, de dimension 1000 avec les valeurs 0.1 sur toute sa diagonale supérieure. Les valeurs sur sa diagonale principale sont les suivantes : 0.1, 0.2, 0.3, 0.4, 0.5 pour les cinq premières valeurs, 6 à 1000 pour les autres, avec un pas de 1.

La méthode **GMRES** non redémarrée converge en 229 itérations et la méthode **FOM** non redémarrée en 231.

m	Perf.	GMRES	FOM
20	Proc	200	200
	Matvec	4001	4001
	Reduct.	2.1929E-02	3.9344E-00
30	Proc	200	200
	Matvec	6001	6001
	Reduct.	2.0120E-02	1.4008E-03
40	Proc	200	200
	Matvec	8001	8001
	Reduct.	2.0594E-02	1.7539E+19
50	Proc	200	182
	Matvec	10001	9064
	Reduct.	2.0077E-02	8.6687E-10

TAB. 4.1 – *Matrice EX1, tol =10E-9, maxproc = 200, Precon = no.*

Perf.	QMR	Bi-CGSTAB
Proc	227	399
Matvec A	228	799
Matvec A^T	227	0
Reduct.	8.9750E-10	7.4079E-10

TAB. 4.2 – *Matrice EX1, tol =10E-9, maxit = 2000, Precon = no.*

Comme nous pouvons le voir dans la table 4.1, **GMRES** ne converge pour aucun test, tandis que **FOM** ne converge que pour $m = 50$. En fait, la norme de leurs résidus stagne. Ceci est dû au redémarrage et à la difficulté pour ces méthodes d'approcher de façon correcte les valeurs propres les plus proches de zéro.

Les méthodes **Bi-CGSTAB** et **QMR** (voir table 4.2) convergent très rapidement en nombre d'itérations, tout comme les versions non redémarrées de **GMRES** et **FOM**. Elles effectuent cependant deux fois plus de produits matrice vecteur au total.

D'après la table 4.3, nous pouvons dire :

- les méthodes déflatées réduisent considérablement le nombre de processus à partir de $k = 3$ seulement et le nombre de processus diminue lorsque k augmente, pour m fixé,

- à l'exception de **IDFOM**, la valeur de k égale à 1 est trop petite pour que les méthodes déflatées approchent suffisamment bien les valeurs propres proches de zéro,

- lorsque k augmente, le nombre de produits matrice vecteur de toutes les méthodes déflatées décroît. Choisir $k = 3$ au lieu de $k = 1$ conduit à une convergence rapide, à la place d'un procédé qui stagne. Les valeurs propres les plus proches de zéro sont en fait approchées de plus en plus rapidement et plus précisément lorsque k augmente,

m/k		AGMRES	AGMRES+	IDFOM	IDGMRES	IDMGMRES
20/1	Proc	200	200	200	200	200
	Matvec	4200	3801	3802	3802	3712
	Reduct.	2.0254E-02	2.0254E-02	7.1219E-01	2.0250E-02	4.0363E-02
20/3	Proc	168	172	200	96	78
	Matvec	3516	2912	3404	1633	1327
	Reduct.	9.9770E-10	9.9918E-10	6.7017E-07	9.8248E-10	5.4268E-10
20/6	Proc	20	20	19	19	26
	Matvec	397	269	272	268	366
	Reduct.	9.0890E-10	9.0890E-10	7.8643E-10	9.4811E-10	2.4981E-10
30/1	Proc	200	200	200	200	200
	Matvec	6200	5801	5802	5802	5703
	Reduct.	2.0077E-02	2.0077E-02	7.1017E-08	2.0077E-02	2.1759E-02
30/3	Proc	29	29	17	23	88
	Matvec	898	784	448	616	2345
	Reduct.	8.5696E-10	8.5629E-10	9.9454E-10	9.7928E-10	3.1020E-11
30/6	Proc	11	11	11	11	11
	Matvec	318	253	256	252	269
	Reduct.	8.8821E-10	8.8821E-10	8.3377E-10	8.7220E-10	9.4485E-10
40/1	Proc	200	200	81	200	200
	Matvec	8200	7801	3119	7802	7802
	Reduct.	2.0077E-02	2.0077E-02	8.4790E-10	2.077E-02	2.2873E-02
40/3	Proc	11	11	12	10	15
	Matvec	412	373	423	371	557
	Reduct.	8.9534E-10	8.9565E-10	9.5688E-10	8.5029E-10	3.1323E-11
40/6	Proc	8	8	8	8	8
	Matvec	293	249	253	248	278
	Reduct.	9.4717E-10	9.6941E-10	7.7630E-10	9.7923E-10	5.9823E-10
50/1	Proc	200	200	78	200	200
	Matvec	10200	9801	3770	9802	9703
	Reduct.	2.0076E-02	2.0076E-02	8.5044E-10	2.0077E-02	5.4974E-02
50/3	Proc	7	7	13	7	9
	Matvec	356	330	577	314	426
	Reduct.	3.6822E-11	3.6447E-11	9.2228E-10	9.7071E-10	2.4308E-10
50/6	Proc	6	6	6	6	7
	Matvec	274	244	248	246	315
	Reduct.	9.2912E-10	9.3046E-10	8.9910E-10	9.4345E-10	1.2899E-13

TAB. 4.3 – *Matrice EX1, tol = 10E-9, tolvp = 10E-6, maxproc = 200, Precon = no.*

mais pas suffisamment pour qu'une réelle déflation ait lieu. Si l'on prend k plus grand encore, le procédé ne converge pas beaucoup plus vite en nombre de produits matrice vecteur et parfois même ralentit, mais la déflation a alors réellement lieu. Par exemple, choisir k égal à $m/2$ ne réduit pas réellement le nombre de produits matrice vecteur, mais à la fin de la procédure, lorsque la méthode a convergé, les paires de Ritz associées aux plus petites valeurs de Ritz ont aussi convergé, alors qu'aucune déflation n'a eu lieu pour les tests de la table 4.3,

- **AGMRES** et **AGMRES+** convergent en un nombre de processus identique,

$$nb_{AGMRES} \geq nb_{AGMRES+},$$

mais pas en nombre de produits matrice vecteur, puisque **AGMRES+** effectue moins de produits matrice vecteur par processus. Cependant, **AGMRES+** nécessite davantage de stockage,

- **IDGMRES** converge en un nombre de processus inférieur ou égal à celui de **AGMRES+**. Chaque processus de **IDGMRES** étant moins coûteux qu'un processus de **AGMRES+**, ces performances en environnements parallèles seront meilleures. En ce qui concerne le nombre de produits matrice vecteur, nous avons excepté pour $m = 50$ et $k = 6$:

$$nb_{AGMRES+} \geq nb_{IDGMRES},$$

- nous avons pour $k = 6$ et toutes les valeurs m testées :

$$nb_{\text{AGMRES}} \geq nb_{\text{IDFOM}} \geq nb_{\text{IDGMRES}},$$

- la méthode **IDMGMRES** ne converge pas plus rapidement que **IDGMRES**, en nombre de processus et de produits matrice vecteur,

- la convergence de la méthode **IDGMRES** est plus régulière que celle de **IDFOM**, pour laquelle la convergence est imprévisible,

- pour $m \geq 40$ et $k = 3$ ou 6, le nombre de produits matrice vecteur de **AGMRES**, **AGMRES+**, **IDGMRES** et **IDMGMRES** pour converger est inférieur au nombre de produits matrice vecteur des méthodes **Bi-CGSTAB** et **QMR**,

- le nombre minimal de produits matrice vecteur pour converger est atteint par **AGMRES+** pour $m = 50, k = 6$ en 244 produits matrice vecteur et **IDGMRES** pour $m = 50$ et $k = 6$ en 246 produits matrice vecteur. D'autres tests ont montré que pour $m = 40$ et $k = 10$, **IDGMRES** converge en 237 produits matrice vecteur, ce qui est presque ce qu'effectue la méthode **GMRES** non redémarrée pour converger.

Comme cela avait déjà été montré dans [34], les techniques de déflation apportent une réelle accélération par rapport aux méthodes correspondantes. Pour cette matrice, **IDGMRES** est la méthode qui se comporte le mieux en réduction du nombre de processus et presqu'aussi bien que **GMRES** non redémarrée, tandis que **GMRES** stagne. Un

m	Perf.	GMRES	FOM
20	Proc	200	200
	Matvec	4001	4001
	Reduct.	1.6845E-06	1.3962E-05
30	Proc	144	164
	Matvec	4306	4902
	Reduct.	9.9701E-10	9.3368E-10
40	Proc	85	95
	Matvec	3364	3791
	Reduct.	9.9711E-10	9.8299E-10
50	Proc	58	61
	Matvec	2873	3047
	Reduct.	9.9772E-10	9.8951E-10

TAB. 4.4 – *Matrice SHERMAN1, tol =10E-9, maxproc = 200, Precon = no.*

Perf.	QMR	Bi-CGSTAB
Proc	564	433
Matvec A	565	867
Matvec A^T	564	0
Reduct.	9.8652E-10	9.8153E-10

TAB. 4.5 – *Matrice SHERMAN1, tol =10E-9, maxit = 2000, Precon = no.*

processus de la méthode **IDGMRES** étant moins coûteux que celui de **FOM**, **GMRES** et **AGMRES+**, de bons temps d'exécution sur machines parallèles peuvent être attendus.

Deuxième matrice : la deuxième matrice est non symétrique, de dimension 1000. C'est la matrice Sherman1 de la collection Harwell-Boeing.

Nous pouvons voir dans les tables 4.4 et 4.5 que les méthodes **QMR** et **Bi-CGSTAB** convergent beaucoup plus rapidement que les méthodes **GMRES** et **FOM**. La méthode non redémarrée **GMRES** converge cependant en 376 produits matrice vecteur et la version non redémarrée de **FOM** en 380.

Au regard de la table 4.6, nous pouvons faire les remarques suivantes :

- les méthodes déflatées réduisent considérablement le nombre de processus total par rapport aux méthodes correspondantes dès $k = 1$. Lorsque k augmente pour m fixé, le nombre de processus diminue. Comme la complexité d'un processus diminue avec k pour les méthodes implicitement déflatées et **AGMRES+**, de bons temps sont attendus sur machines parallèles,

- pour chaque méthode de déflation et chaque m fixé, le nombre total de produits matrice vecteur diminue lorsque k augmente,

m/k	Perf.	AGMRES	AGMRES+	IDFOM	IDGMRES	IDMGMRES
20/1	Proc	142	142	149	141	69
	Matvec	2964	2682	2825	2665	1313
	Reduct.	9.9311E-10	9.9312E-10	9.3415E-10	9.9642E-10	8.5251E-10
20/3	Proc	122	122	128	122	62
	Matvec	2550	2068	2178	2073	1058
	Reduct.	9.9160E-10	9.9158E-10	8.8339E-10	9.9634E-10	7.3851E-10
20/6	Proc	101	101	108	100	58
	Matvec	2101	1406	1510	1405	819
	Reduct.	9.9509E-10	9.9509E-10	8.9183E-10	9.9579E-10	2.3213E-10
30/1	Proc	64	64	66	63	37
	Matvec	1957	1831	1902	1827	1075
	Reduct.	9.9158E-10	9.9158E-10	9.5594E-10	9.9958E-10	9.3685E-10
30/3	Proc	53	53	56	52	30
	Matvec	1615	1409	1494	1402	814
	Reduct.	9.9520E-10	9.9522E-10	9.8049E-10	9.9012E-10	9.7897E-10
30/6	Proc	42	42	44	41	31
	Matvec	1281	999	1051	987	751
	Reduct.	9.8583E-10	9.8583E-10	9.9522E-10	9.9988E-10	5.3735E-10
40/1	Proc	37	37	39	37	26
	Matvec	1485	1413	1504	1409	1016
	Reduct.	9.9607E-10	9.9554E-10	9.7896E-10	9.9460E-10	2.1861E-10
40/3	Proc	30	30	33	29	22
	Matvec	1213	1099	1206	1056	818
	Reduct.	9.9314E-10	9.9045E-10	9.0914E-10	9.9965E-10	4.1344E-10
40/6	Proc	26	26	27	25	23
	Matvec	1022	852	911	848	789
	Reduct.	9.8497E-10	9.8500E-10	9.9776E-10	9.8840E-10	4.4913E-10
50/1	Proc	26	26	27	26	18
	Matvec	1283	1230	1284	1232	884
	Reduct.	9.8014E-10	9.9709E-10	8.9342E-10	9.8834E-10	8.9597E-10
50/3	Proc	21	21	22	21	17
	Matvec	1052	973	1037	973	803
	Reduct.	9.8191E-10	9.9889E-10	9.7640E-10	9.8813E-10	6.5222E-10
50/6	Proc	18	18	19	18	17
	Matvec	890	777	818	771	755
	Reduct.	9.8258E-10	9.7531E-10	9.7884E-10	9.8049E-10	3.4586E-10

TAB. 4.6 – *Matrice SHERMAN1, tol = 10E-9, tolvp = 10E-6, maxproc = 200, Precon = no.*

- comme dans le cas précédent, **AGMRES+** converge en un même nombre de processus que la méthode **AGMRES**, mais bien sûr en un nombre de produits matrice vecteur moindre,

- **IDGMRES** converge en un nombre de processus très proche mais inférieur ou égal à celui de **AGMRES+**. Nous avons:

$$nb_{AGMRES+} \simeq nb_{IDGMRES},$$

- **IDMGMRES** est le méthode convergeant toujours avec un nombre de processus le plus petit par rapport aux autres méthodes, ce qui laisse présager de bons temps d'exécution sur machines parallèles,

- excepté pour **IDFOM** $m = 40, 50$ et $k = 1$, nous avons:

$$nb_{\text{FOM}} \geq nb_{\text{GMRES}} \geq nb_{\text{AGMRES}} \geq nb_{\text{IDFOM}} \geq nb_{\text{IDGMRES}} \geq nb_{\text{IDMGMRES}},$$

- la méthode **IDMGMRES** converge avec un nombre de produits matrice vecteur inférieur à celui des méthodes **QMR** et **Bi-CGSTAB** lorsque $m = 20, k = 6$ et $m = 30, 40, 50$ et $k \geq 3$.

- les méthodes **IDGMRES** et **AGMRES+** convergent plus rapidement en nombre de produits matrice vecteur que les méthodes **QMR** et **Bi-CGSTAB** pour $m = 40, 50$ et $k = 6$,

- le nombre minimal de produits matrice vecteur avec A est atteint par la méthode **IDMGMRES** pour $m = 30, k = 6$ en 751 produits matrice vecteur avec A et pour $m = 50, k = 6$ en 755 produits matrice vecteur.

Pour cette matrice, la méthode **IDMGMRES** est plus performante que la méthode **IDGMRES**. Nous sommes encore loin des résultats des versions non redémarrées des méthodes **GMRES** et **FOM**, mais **IDGMRES** converge tout de même en un nombre de produits matrice vecteur inférieur à celui des méthodes **QMR** et **Bi-CGSTAB**. En ce qui concerne les autres techniques de déflation, elles convergent plus rapidement que les méthodes dont elles sont dérivées. Les méthodes **AGMRES+** et **AGMRES** diminuent autant le nombre de processus que les méthodes implicitement redémarrées, mais étant de complexité plus grande, elle convergeront en un temps d'exécution supérieur.

Troisième matrice : la troisième matrice est une matrice non symétrique de dimension 680. C'est la matrice fs6801 de la collection Harwell-Boeing. C'est une matrice définie positive dont les valeurs propres sont comprises entre $1.0E-9$ et $1.0E-13$.

Comme nous pouvons le voir dans les tables 4.7 et 4.8, cette matrice est difficile pour les méthodes **QMR** et **Bi-CGSTAB**, mais pas pour **GMRES** et surtout **FOM**. Dans leur version non redémarrée, **GMRES** et **FOM** convergent en 106 produits matrice vecteur.

La table 4.9 nous montre:

- les méthodes déflatées ne réduisent pas toujours le nombre de processus par rapport aux méthodes correspondantes,

m	Perf.	GMRES	FOM
20	Proc	68	49
	Matvec	1356	976
	Reduct.	9.9477E-10	7.8822E-10
30	Proc	29	26
	Matvec	870	763
	Reduct.	9.9638E-10	8.9748E-10
40	Proc	15	14
	Matvec	587	560
	Reduct.	9.3015E-10	8.5277E-10
50	Proc	9	8
	Matvec	443	400
	Reduct.	8.7179E-10	8.7473E-10

TAB. 4.7 – *Matrice FS6801, tol =10E-9, maxproc = 200, Precon = no.*

Perf.	QMR	Bi-CGSTAB
Proc	2000	771
Matvec A	2001	1543
Matvec A^T	2000	0
Reduct.	6.1434E-09	9.0315E-10

TAB. 4.8 – *Matrice FS6801, tol =10E-9, maxit = 2000, Precon = no.*

m/k	Perf.	AGMRES	AGMRES+	IDFOM	IDGMRES	IDMGMRES
20/1	Proc	70	68	68	59	44
	Matvec	1464	1278	1294	1118	837
	Reduct.	9.9242E-10	9.8981E-10	7.8114E-10	9.8577E-10	9.3283E-10
20/3	Proc	80	78	74	76	54
	Matvec	1664	1327	1249	1293	919
	Reduct.	9.9040E-10	9.9672E-10	9.4999E-10	9.9858E-10	5.1104E-10
20/6	Proc	89	88	93	97	68
	Matvec	1923	1231	1269	1270	932
	Reduct.	9.9803E-10	9.8281E-10	8.0788E-10	9.7798E-10	3.7984E-10
30/1	Proc	27	27	29	28	22
	Matvec	834	782	823	798	640
	Reduct.	9.8240E-10	9.8220E-10	9.839E-10	9.9871E-10	6.3592E-10
30/3	Proc	29	29	29	28	28
	Matvec	871	762	781	748	759
	Reduct.	9.9980E-10	9.9438E-10	8.3553E-10	9.9013E-10	8.8989E-10
30/6	Proc	32	31	33	32	29
	Matvec	996	741	766	756	695
	Reduct.	9.9107E-10	9.733E-10	8.8954E-10	9.8547E-10	4.7059E-10
40/1	Proc	15	15	15	15	14
	Matvec	613	585	581	575	548
	Reduct.	9.2482E-10	9.2482E-10	7.7028E-10	9.8492E-10	2.9212E-10
40/3	Proc	15	15	15	15	14
	Matvec	609	555	556	541	522
	Reduct.	9.9669E-10	9.9670E-10	8.8744E-10	9.8112E-10	6.2157E-10
40/6	Proc	17	17	17	17	16
	Matvec	667	551	574	547	544
	Reduct.	9.9130E-10	9.9130E-10	9.4862E-10	9.8785E-10	6.5677E-10
50/1	Proc	10	10	9	10	9
	Matvec	485	467	424	459	443
	Reduct.	9.4869E-10	9.4869E-10	9.0454E-10	9.9525E-10	5.9862E-10
50/3	Proc	9	9	9	9	10
	Matvec	452	422	421	423	470
	Reduct.	9.8758E-10	9.8758E-10	8.9576E-10	9.2485E-10	1.4496E-10
50/6	Proc	10	10	10	10	10
	Matvec	493	432	436	438	442
	Reduct.	9.3673E-10	9.3673E-10	8.2046E-10	9.3356E-10	5.6090E-10

TAB. 4.9 – *Matrice FS6801, tol = 10E-9, tolvp = 10E-6, maxproc = 200, Precon = no.*

- le nombre de processus des méthodes déflatées ne diminue pas forcément lorsque k grandit,
- les méthodes **FOM** et **GMRES** convergent en un nombre de produits matrice vecteur inférieur à celui de **QMR** et **Bi-CGSTAB**. Il en est de même pour toutes les méthodes déflatées,
- **FOM** est déjà très performante et la méthode **IDFOM** ne fait jamais mieux,
- la méthode **AGMRES+** converge avec à peu près le même nombre de processus que la méthode **AGMRES**, mais en moins de produits matrice vecteur. Elle est compétitive avec **IDGMRES**. Nous avons :

$$nb_{AGMRES+} \simeq nb_{IDGMRES},$$

- les méthodes **IDGMRES** et **AGMRES+** effectuent moins de produits matrice vecteur pour converger que **GMRES**, pour tous les cas testés, excepté pour $m = 50$ et $k = 1$,
- pour tous les cas où $m < 50$, **IDMGMRES** converge en un nombre de produits matrice vecteur inférieur à ceux de **FOM**, **GMRES**, **AGMRES**, **AGMRES+**, **IDFOM** et **IDGMRES**,
- le nombre minimal de produits matrice vecteur est atteint par **FOM** pour $m = 50$ en 408 produits matrice vecteur.

FOM est très performante pour cette matrice et **IDFOM** ne peut converger en un nombre inférieur de processus et de produits matrice vecteur. Les méthodes **IDGMRES** et **AGMRES+** , quant à elles, effectuent moins de produit matrice vecteur pour converger que la méthode **GMRES**, mais toujours en un nombre supérieur à celui de **FOM**. Parmi les méthode déflatées, seule **IDMGMRES** est plus efficace puisqu'elle parvient à converger presque pour tous les cas, en un nombre de processus inférieur à celui de **GMRES** et un nombre de produits matrice vecteur inférieur à celui de **FOM**, qui effectue déjà peu de produits matrice vecteur.

4.6 Conclusion

Nous avons mis en avant dans ce chapitre différentes techniques liées à la déflation permettant de limiter la perte d'information dû au redémarrage des méthodes **FOM** et **GMRES**. Ces techniques s'articulent autour de deux approches : l'approche préconditionnement et l'approche enrichissement du sous-espace de Krylov. Nous avons présenté une nouvelle technique d'enrichissement du sous-espace de Krylov, s'aidant de la méthode **IRA** pour redémarrer les méthodes **FOM** et **GMRES**. Cette technique permet par rapport aux techniques d'enrichissement du sous-espace de Krylov existantes :

- d'effectuer un enrichissement du sous-espace de Krylov de façon naturelle, permettant ainsi à la fois de calculer les solutions **FOM** et **GMRES** et de mieux approcher les paires de Ritz nous aidant dans la déflation,

- d'appliquer les techniques de déflation existantes pour la recherche de valeurs propres dans la méthode d'Arnoldi,
- de réduire le coût de stockage : il est alors égal à celui des méthodes **FOM** et **GMRES**,
- de réduire la complexité en temps d'un processus.

Les résultats numériques ont montré que les méthodes convergeaient en un nombre de processus égal, voire inférieur à celui des méthodes d'enrichissement du sous-espace de Krylov existantes, lesquelles réduisent déjà nettement le nombre de processus par rapport aux méthodes **FOM** et **GMRES**. Un processus de ces méthodes étant moins coûteux en environnement séquentiel, mais surtout parallèle, des gains en temps d'exécution sur machines parallèles sont attendus. Cela fait l'objet d'une étude qui est en cours.

Nous avons également adapté cette technique à une troisième méthode **MGMRES**, qui se situe entre **FOM** et **GMRES**. Son implantation nécessite une modification de coût négligeable par rapport à la méthode **GMRES**. Utilisée avec la technique de déflation que nous présentons dans ce chapitre, elle permet de diminuer notablement le nombre de processus par rapport aux méthodes **FOM** et **GMRES**. Un processus étant de même coût qu'un processus **GMRES**, elle devrait nous permettre d'obtenir des gains substanciels sur machines parallèles.

Conclusion et perspectives

Nous avons montré dans cette thèse qu'il était possible d'accélérer les méthodes **FOM**(m) et **GMRES**(m) en environnements parallèles.

Nous avons proposé, tout d'abord dans le chapitre 3, des méthodes permettant de remédier aux problèmes des produits scalaires faits en séquence dans le processus d'Arnoldi des méthodes **FOM**(m) et **GMRES**(m), en choisissant un produit scalaire discret. Nous avons alors obtenu deux méthodes l'une de type **RQO** l'autre de type **RQM**, qui permettent d'effectuer un produit matrice matrice au lieu d'une succession de produits scalaires. Sans préconditionnement, ces méthodes permettent d'obtenir un gain substantiel en temps d'exécution par rapport aux méthodes **FOM**(m) et **GMRES**(m). Ceci est d'autant plus vrai que la matrice est très creuse. Nous avons vu que ces méthodes étaient particulièrement performantes pour des dimensions de sous-espace de Krylov peu élevées. Préconditionnée, ces méthodes n'ont pas donné d'aussi bons résultats.

Ces méthodes présentent l'intérêt supplémentaire qu'elles peuvent être utilisées comme moyen d'approcher rapidement une solution dont nous ne désirons pas avoir une trop grande précision. C'est le cas notamment dans **FGMRES**, où l'on peut enrichir le sous-espace de Krylov par le calcul approché de la solution du système $Ax = v_i$, où v_i est un vecteur orthonormal de la base de Krylov.

Dans le chapitre 4, nous avons proposé une technique de déflation et de redémarrage implicites, permettant de ne pas perdre la totalité des informations entre deux redémarrages des méthodes **FOM**(m) et **GMRES**(m). Les méthodes résultantes dénomées **IDFOM** et **IDGMRES** ont les avantages suivants : elles sont de complexité inférieure à celle des méthodes **FOM**(m) et **GMRES**(m) et ont un coût de stockage égal ; elles permettent d'enrichir le sous-espace de Krylov de façon naturelle et de tirer profit des techniques de déflation existantes pour la méthode **QR**. Comparées aux méthodes **GMRES**(m) et **FOM**(m), les résultats numériques faits en séquentiel ont montré une nette réduction du nombre total d'itérations. Nous avons également comparé ces méthodes à d'autres techniques d'enrichissement du sous-espace de Krylov : les résultats obtenus pour les méthodes **IDFOM** et **IDGMRES** sont dans l'ensemble meilleurs et au pire équivalents. De complexité inférieure, nous espérons obtenir de très bonnes accélérations en environnements parallèles.

Suite au travail qui a été fait dans cette thèse, nous aimerions le poursuivre sur différents plans. Tout d'abord, un travail de parallélisation des méthodes **IDFOM** et **IDGMRES** est en cours. Nous comptons effectuer des tests sur machines parallèles et voire de façon pratique, si nos espérances, quant à l'accélération du temps total d'exécution, sont fondées.

Nous aimerions également trouver des préconditionnements plus adaptés aux méthodes **ADOP**. De bons candidats sont les préconditionnements polynomiaux qui tireraient partie de l'approximation des valeurs propres que nous calculons à chaque pas des méthodes **ADOP**.

Nous aimerions également faire le lien entre les méthodes **ADOP** et la déflation utilisée dans les méthodes **IDFOM** et **IDGMRES**. Ceci peut être fait en gardant les valeurs de Ritz approchées par les méthodes **ADOP**, d'un processus à l'autre, lorsqu'elles ont suffisamment convergé.

Enfin, nous aimerions utiliser le produit scalaire discret développé dans le chapitre 3 pour développer de nouvelles accélérations polynomiales et de nouveaux préconditionneurs polynomiaux. Nous calculerions ainsi le polynôme minimal au sens de la norme induite par le produit scalaire discret et que nous utiliserions comme polynôme de préconditionnement ou d'accélération (voir chapitre 2) d'une méthode de Krylov. Le calcul de ce polynôme serait simplifié par rapport aux techniques existantes et son calcul de coût plus faible.

Bibliographie

[1] J. Baglama, D. Calvetti, G. H. Golub, and L. Reichl. Adaptively preconditioned GMRES algorithms. *SIAM J. Sci. Comput.*, à paraître.

[2] G. Bergère. *Contribution à une programmation parallèle hétérogène de méthodes numériques hybrides.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 1999.

[3] C. Brezinski. Hybrid procedures and semi-iterative methods for linear systems. Technical Report ANO-340, Université des Sciences et Technologies de Lille, Laboratoire d'Analyse Numérique et d'Optimisation, 1995.

[4] C. Brezinski and M. Redivo-Zaglia. Hybrid procedures for solving linear systems. *Numer. Math.*, pages 1–19, 1994.

[5] C. Brezinski and M Redivo-Zaglia. Transpose-free Lanczos-type algorithms for nonsymmetric linear systems. Technical Report ANO-377, Université des Sciences et Technologies de Lille, Laboratoire d'Analyse Numérique et d'Optimisation, 1997.

[6] K. Burrage and J. Erhel. On the performance of various adaptive preconditioned GMRES strategies. Technical Report 1081, IRISA, France, 1997.

[7] D. Calvetti and L. Reichel. Adaptive Richardson iteration based on leja points. *J. Comput. Appl. Math.*, 71 :267–286, 1996.

[8] D. Calvetti and L. Reichel. An adaptive Richardson iteration method for indefinite linear systems. *Numer. Algo.*, 12 :125–149, 1996.

[9] T. F. Chan and H. A. van der Vorst. Approximate and incomplete factorizations. *Parallel Numerical Algorithms*, 4 :167–202, 1997.

[10] A. Chapman and Y. Saad. Deflated and augmented Krylov subspace techniques. *Numer. Linear Algebra Appl.*, 4 :43–66, 1997.

[11] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, 1992.

[12] M. Eiermann and O. Ernst. Preliminary version : On some recurrent theorems concerning Krylov subspace methods. Technical report, Technische Universität, Bergakademie Freiberg, Freiberg, 1998.

[13] J. Erhel, K. Burrage, and B. Pohl. Restarted GMRES preconditioned by deflation. *J. Comp. Appl. Math.*, 69 :303–318, 1996.

[14] A. Essai. *Méthode hybride parallèle et méthodes pondérées pour la résolution des systèmes linéaires.* PhD thesis, Laboratoire d'Analyse numrérique et d'Optimisation, Université des Sciences et Technologies de Lille, 1999.

[15] R. Fletcher. Conjugate gradient methods for indefinite systems in Numerical Analysis. *G. A. Watson ed., LNM 506, Springer Verlag, Berlin*, pages 73–89, 1976.

[16] M. J. Flynn. Some computer organization and their effectiveness. *IEEE Trans. on Computers*, C-21 :948–960, 1972.

[17] J. G. Francis. The QR transformation : A unitary analogue to the LR transformation, parts i and ii. *Comput. J.*, 4 :265–272,332–345, 1961.

[18] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.*, 14(2) :470–482, 1993.

[19] R. W. Freund and M. Malhotra. A block-QMR algorithm for non-hermitian linear systems with multiple right-hand sides. Technical Report SCCM-96-01, Stanford University, Computer Sciences Dept., 1996.

[20] R. W. Freund and N. M. Nachtigal. QMR : a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60 :315–339, 1991.

[21] W. Gautschi. On generating orthogonal polynomials. *SIAM J. Scient. Stat. Comput.*, 3 :289–317, 1982.

[22] G.H. Golub and C. F. van Loan. *Matrix Computations.* The John Hopkins University Press, Baltimore, 1989.

[23] M. R. Hestenes and E. L. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Natl. Bur. Stand.*, Section B, 49 :409–436, 1952.

[24] M. Heyouni. *Méthode de Hessenberg généralisée et applications.* PhD thesis, Laboratoire d'Analyse numrérique et d'Optimisation, Université des Sciences et Technologies de Lille, 1996.

[25] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report 96-036, Departement of Computer Science and Civil Engineering, University of Minnesota, Minnesota, USA, 1996.

[26] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Paral. Distrib. Comput.*, à paraître.

[27] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Natl. Bur. Stand.*, 45 :255–282, 1950.

[28] C. Le Calvez and B. Molina. Implicitly restarted and deflated FOM and GMRES. Technical Report as-184, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 1998.

[29] C. Le Calvez and Y. Saad. Modified Krylov acceleration for parallel environments. *Numer. Appl. Math.*, à paraître.

[30] R. B. Lehoucq and D. C. Sorensen. Deflation techniques for an implicitly restarted Arnoldi iteration. *SIAM J. Matrix Anal. Appl.*, 17(4):789–821, 1996.

[31] M. P. I. FORUM. MPI: a message passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, 1994.

[32] T. A. Manteuffel. The Tchebychev iteration for nonsymmetric linear systems. *Numer. Math.*, 38:307–327, 1977.

[33] T. A. Manteuffel. Adaptative procedure for extimating parameters for the nonsymmetric Tchebyshev iteration. *Numer. Math.*, 31:183–208, 1978.

[34] R. B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM J. Matrix Anal. Appl.*, 4(16):1154–1171, 1995.

[35] R. B. Morgan. On restarting the Arnoldi method for large nonsymmetric eigenvalue problems. *Math. Comput.*, 65:1213–1230, 1996.

[36] R. B. Morgan. Implicitly restarted GMRES and Arnoldi methods for nonsymmetric systems of equations. *SIAM J. Matrix Anal. Appl.*, à paraître.

[37] L. Reichel N. M. Nachtigal and L. Trefethen. A hybrid GMRES algorithm for nonsymmetric linear systems. *SIAM J. Matrix Anal. Appl.*, 13:796–825, 1992.

[38] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra Appl.*, 29:293–322, 1980.

[39] M. Gutknecht R. Freund and N. Nachtigal. An implementaion of the look-ahead Lanczos algorithm for non-hermitian matrices. *SIAM J. Sci. Stat. Comput.*, 14:137–158, 1993.

[40] J. K. Reid. On the method of conjugate gradients for the solution of large sparse systems of linear equations. *Large Sparse Sets of Linear Equations, Academic Press*, pages 231–254, 1971.

[41] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Math. Comput.*, 37:105–126, 1981.

[42] Y. Saad. Practical use of some Krylov subspace methods for solving indefinite and nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 5:203–228, 1984.

[43] Y. Saad. Least-squares polynomials in the complex plane and their use for solving nonsymmetric linear systems. *SIAM J. Numer. Anal.*, 24:155–169, 1987.

[44] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[45] Y. Saad. *Numerical Methods for Large Eigenvalue Problem.* Halstead Press, New York, 1992.

[46] Y. Saad. A parallel multi-elimination ILU preconditionner for general sparse matrices. Technical Report UMSI 92/241, Supercomputing Institute, University of Minnesota, Minnesota, USA, 1992.

[47] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Scient. Stat. Comput.*, 14:461–469, 1993.

[48] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numer. Linear Algebra Appl.*, 1:387–402, 1994.

[49] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.

[50] Y. Saad. Analysis of augmented Krylov subspace methods. *SIAM J. Matrix Anal. Appl.*, 18(2):435–449, 1997.

[51] Y. Saad and A. Malevsky. P-SPARSLIB: A portable library of distributed memory sparse iterative solvers. Technical Report UMSI 95-180, Supercomputing Institute, University of Minnesota, Minnesota, USA, 1995.

[52] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[53] Y. Saad and K. Wu. Design of an iterative solution module for a parallel sparse matrix library (P_SPARSLIB). In W. Schönauer, editor, *Proceedings of IMACS conference, Georgia, 1994*, 1995.

[54] Y. Saad and K. Wu. DQGMRES: A direct quasi-minimal residual algorithm based on incomplete orthogonalization. *Numer. Linear Algebra Appl.*, 3:329–343, 1996.

[55] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h. A deflated version of the conjugate gradient algorithm. Technical Report UMSI 98/97, Supercomputing Institute, University of Minnesota, Minnesota, USA, 1998.

[56] H. Sadok. CMRH: A new method for solving nonsymmetric linear systems based on the Hessenberg reduction algorithm. *soumis.*

[57] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10(1):36–52, 1989.

[58] D. C. Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13:327–341, 1992.

[59] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 2(13):631–644, 1992.

[60] H. A. Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *J. Comput. Appl. Math.*, 48:327–341, 1993.

[61] L. Zhou and H. F. Walker. Residual smoothing techniques for iterative methods. *SIAM J. Sci. Comput.*, 15(2):297–312, 1994.