

Résolution de systèmes linéaires

Enseignement spécialisé "éléments finis" – S6133/5

Christophe Bovet (christophe.bovet@onera.fr)

Onera – The French Aerospace Lab F-92322 Chatillon, France, christophe.bovet@onera.fr

Paris, le 23 novembre 2017



Cette œuvre est mise à disposition sous licence Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 3.0 France.

Pour voir une copie de cette licence, visitez

<http://creativecommons.org/licenses/by-nc-sa/3.0/fr/> ou écrivez à Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

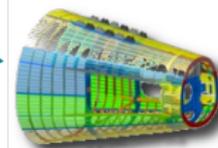
Contexte



Modélisation

$$\begin{cases} \mathbf{f}_{acc} = \mathbf{f}_{int} + \mathbf{f}_{ext} \\ \boldsymbol{\sigma} = \mathbf{f}(\boldsymbol{\epsilon}) \end{cases}$$

Discrétisation



Problème linéaire ?

oui

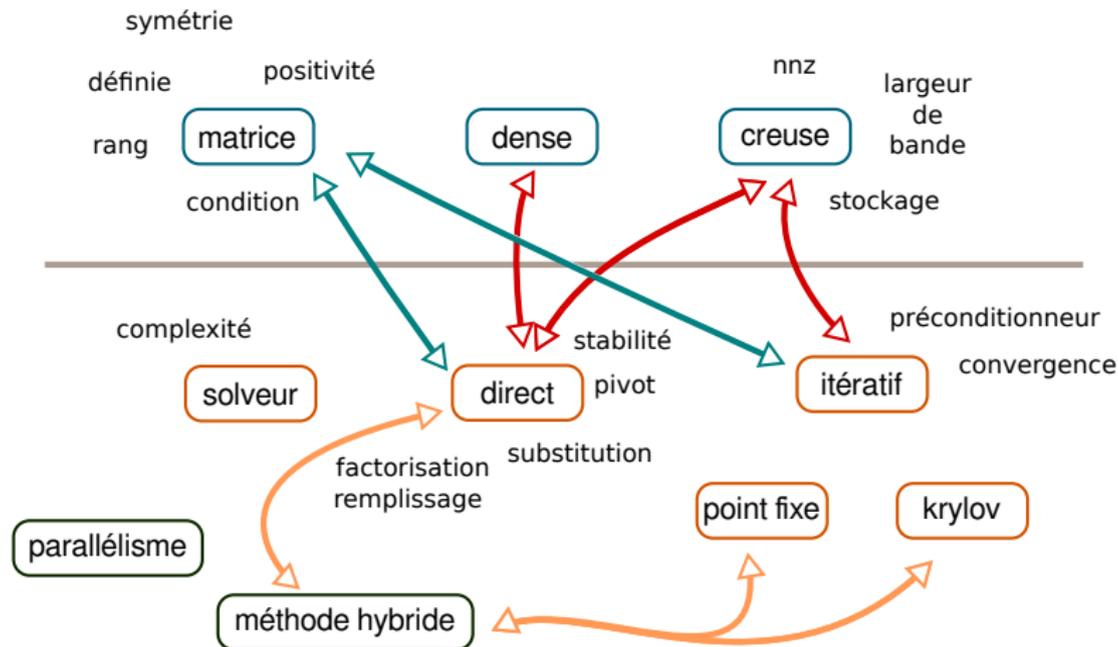
non

Système linéaire $\mathbf{Ax} = \mathbf{b}$

Méthode
de Newton

Résoudre efficacement les grands systèmes linéaires issus de la discrétisation EF est indispensable

Schéma synoptique

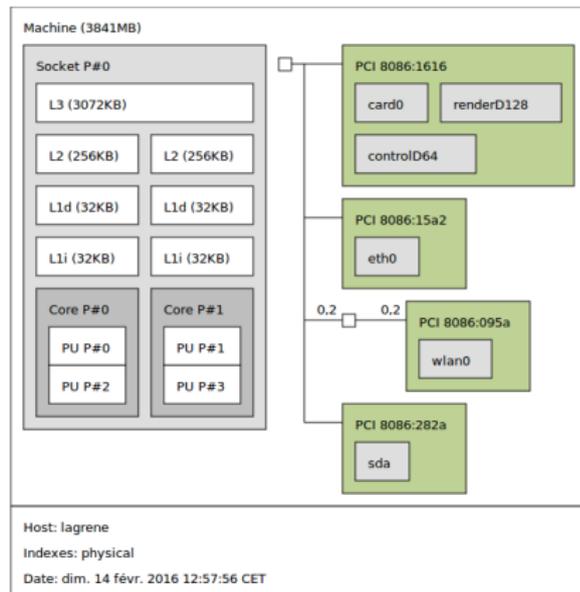


Objectifs

- ▶ Avoir un aperçu des méthodes de résolutions existantes
- ▶ Connaître les avantages & défauts des \neq méthodes
- ▶ Être sensibilisé aux problématiques du monde réel : algèbre linéaire numérique (complexité, stabilité & précision, parallélisme, ...)
- ▶ Avoir des notions des mots clefs du schéma précédent

Architecture d'un ordinateur

- ▶ **Unités de calculs :**
noeud / processeur / cœur
- ▶ **Mémoire :**
cache L1/L2/L3, RAM,
disque dur
- ▶ **Communication :**
bus / réseau (gigabit,
infiniband)



Architecture d'un ordinateur

Mémoire partagée (symmetric multiprocessing)

Échange rapide (accès direct)

Besoin de protéger les données

(mutex/semaphore)

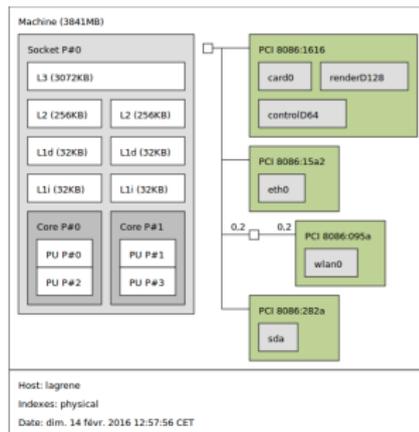
⇒ multithreading, protocole OpenMP

Mémoire séparée

Données protégées

Il faut s'occuper d'échanger les données.

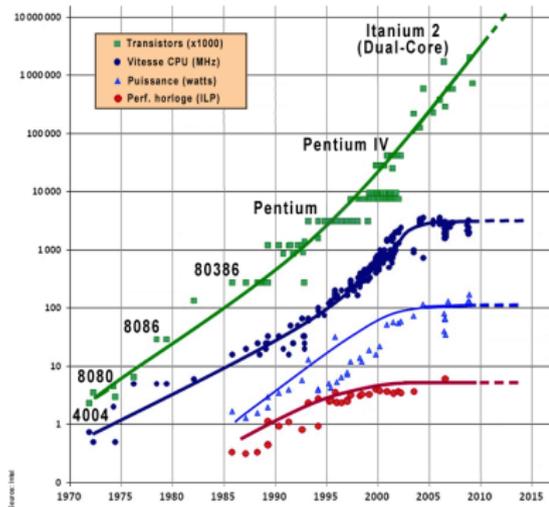
⇒ Protocole MPI (message passing interface)



Architectures actuelles

Architectures mixtes plusieurs nœuds en réseau avec plusieurs cœurs à mémoire partagée.

Constat sur les architectures



Source :

www.astrosurf.com/luxorion/loi-moore.html

Flop/s = floating-point operation per second, (*péta* = 10^{15})

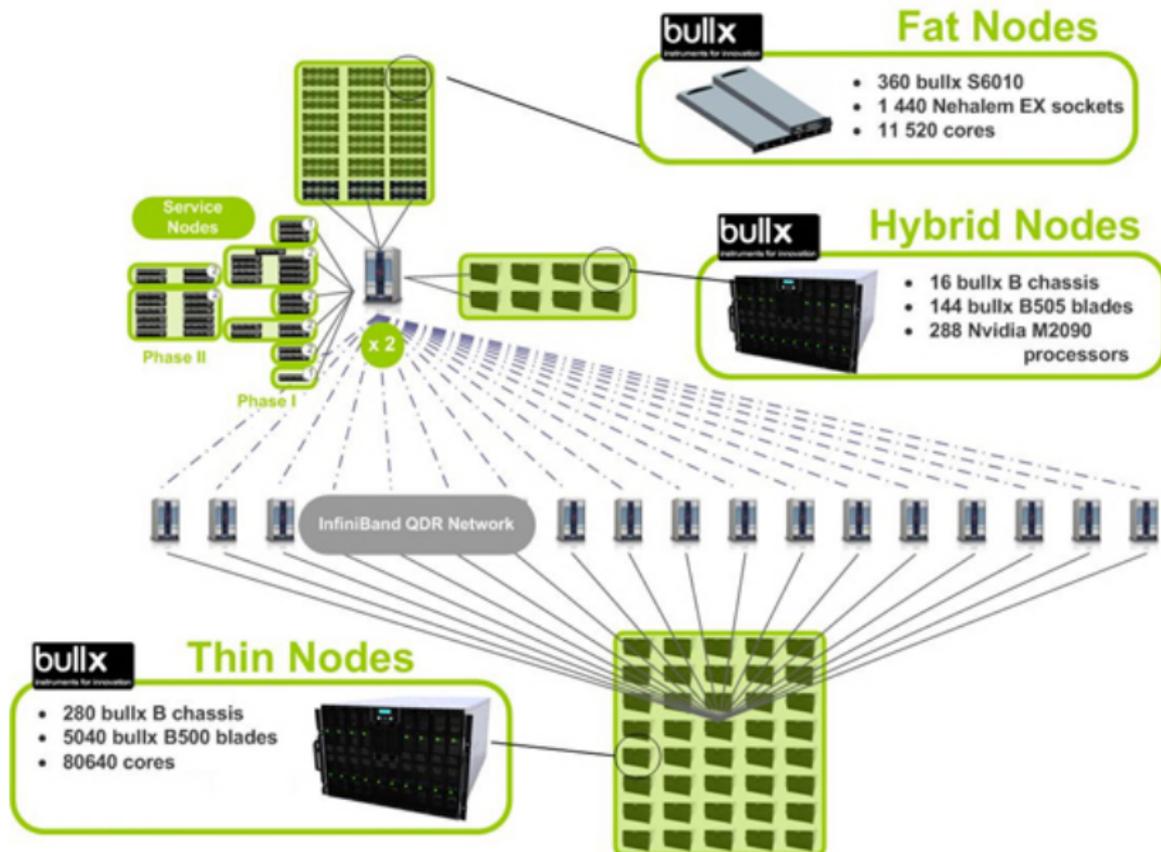
Flop/s théorique = cœurs × fréquence × (Flop/cycle)



Source : www.top500.org

Présentation d'un Cluster

Le calculateur Curie du TGCC



Présentation d'un Cluster

Le calculateur Curie du TGCC

Fat nodes

360 BullX-S6010

Intel NH EX 2,26 Ghz

11 520 cœurs

128 cœurs/nœud

512 Go/nœud

105 TFlops

Thin nodes

5040 BullX B500

Intel Sandy Bridge 2,7Ghz

80 640 cœurs

16 cœurs/nœuds

4 Go/cœurs

1 740 TFlops



Plan

- 1 Quelques rappels d'algèbre linéaire
 - Contexte, vocabulaire et notations
 - Élimination de Gauss
- 2 Méthodes directes
 - Factorisation LU
 - Calculs à virgule flottante
 - Autres factorisations classiques
 - Systèmes creux
- 3 Aperçu des méthodes itératives
 - Méthodes itératives stationnaires
 - Méthodes itératives de type Krylov
- 4 Aperçu d'une méthode hybride
 - Méthodes de décomposition de domaine

Plan

- 1 Quelques rappels d'algèbre linéaire
 - Contexte, vocabulaire et notations
 - Élimination de Gauss
- 2 Méthodes directes
- 3 Aperçu des méthodes itératives
- 4 Aperçu d'une méthode hybride

Contexte

- ▶ Équivalence système linéaire & calcul matriciel :

$$\begin{array}{ccccccc} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 & & & & & & \\ \vdots & \vdots & & \vdots & \vdots & \Leftrightarrow & \mathbf{Ax} = \mathbf{b} \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n & & & & & & \end{array}$$

- ▶ \mathbf{A} et \mathbf{b} (2^{nd} membre, *right hand side*) sont donnés
- ▶ On suppose \mathbf{A} réelle d'ordre n et inversible ($\det(\mathbf{A}) \neq 0$)
- ▶ On cherche à calculer $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, comment faire ?
- ▶ Vocabulaire : \mathbf{A} est dite **dense** s'il y a peu de a_{ij} nuls sinon elle est creuse

Vocabulaire et notations

Rappels

- ▶ \mathbf{A} est symétrique si $\mathbf{A}^T = \mathbf{A}$
- ▶ \mathbf{A} est positive si $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0 \quad \forall \mathbf{x} \in \mathbb{R}^n$
- ▶ \mathbf{A} est définie si pour $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x}^T \mathbf{A} \mathbf{x} = 0 \Rightarrow \mathbf{x} = \mathbf{0}$
- ▶ Norme matricielle :

$$\|\mathbf{A}\| = \max_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \max_{\|\mathbf{x}\|\neq 0} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}$$

- ▶ Condition de \mathbf{A} (inversible) (*condition number*) :

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| = \left(\max_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| \right) / \left(\min_{\|\mathbf{y}\|=1} \|\mathbf{A}\mathbf{y}\| \right)$$

On a toujours $\kappa(\mathbf{A}) \geq 1$

Élimination de Gauss

- Opération sur les lignes jusqu'à obtenir un syst. tri. sup. \Rightarrow succession de systèmes lin. $(\mathbf{A}^{(k)}, \mathbf{b}^{(k)})_k$ on pose $\mathbf{A}^{(1)} = \mathbf{A}$ $\mathbf{b}^{(1)} = \mathbf{b}$

- On pose $m_{i1} = a_{i1}^{(1)} / a_{11}^{(1)}$, on réalise ligne $i - m_{i1}$ ligne 1 (hyp $a_{11} \neq 0$.)

$$\begin{array}{l}
 a_{11}^{(1)} x_1 + a_{12}^{(1)} x_2 + \cdots + a_{1n}^{(1)} x_n = b_1^{(1)} \\
 a_{21}^{(1)} x_1 + a_{22}^{(1)} x_2 + \cdots + a_{2n}^{(1)} x_n = b_2^{(1)} \\
 \vdots \\
 a_{n1}^{(1)} x_1 + a_{n2}^{(1)} x_2 + \cdots + a_{nn}^{(1)} x_n = b_n^{(1)}
 \end{array}
 \Rightarrow
 \begin{array}{l}
 a_{11}^{(1)} x_1 + a_{12}^{(1)} x_2 + \cdots + a_{1n}^{(1)} x_n = b_1^{(1)} \\
 a_{22}^{(2)} x_2 + \cdots + a_{2n}^{(2)} x_n = b_2^{(2)} \\
 \vdots \\
 a_{n2}^{(2)} x_2 + \cdots + a_{nn}^{(2)} x_n = b_n^{(2)}
 \end{array}$$

- À l'étape k

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}} \quad i, j = k + 1, \dots, n$$

$$\mathbf{b}_i^{(k+1)} = \mathbf{b}_i^{(k)} - m_{ik} \mathbf{b}_k^{(k)} \quad i = k + 1, \dots, n \quad a_{kk}^{(k)} \text{ est appelé } \mathbf{pivot}$$

- À l'étape n on a un système tri. sup., $\mathbf{A}^{(n)} \mathbf{x} = \mathbf{Ux} = \mathbf{b}^{(n)}$

Élimination de Gauss : changement de pivot

- ▶ Si $a_{kk}^{(k)} = 0$, il faut permuter des lignes, on parle de changement de pivot

$$\begin{array}{cccc}
 a_{11}^{(1)} x_1 + a_{12}^{(1)} x_2 + \cdots + a_{1n}^{(1)} x_n = b_1^{(1)} & & a_{11}^{(1)} x_1 + a_{12}^{(1)} x_2 + \cdots + a_{1n}^{(1)} x_n = b_1^{(1)} \\
 a_{21}^{(1)} x_1 + a_{22}^{(1)} x_2 + \cdots + a_{2n}^{(1)} x_n = b_2^{(1)} & \Rightarrow & a_{22}^{(2)} x_2 + \cdots + a_{2n}^{(2)} x_n = b_2^{(2)} \\
 \vdots & & \vdots & & \vdots \\
 a_{n1}^{(1)} x_1 + a_{n2}^{(1)} x_2 + \cdots + a_{nn}^{(1)} x_n = b_n^{(1)} & & a_{n2}^{(2)} x_2 + \cdots + a_{nn}^{(2)} x_n = b_n^{(2)}
 \end{array}$$

- ▶ Remarque : la condition $a_{ij} \neq 0$ n'est pas suffisante pour empêcher l'apparition de pivots nuls.
- ▶ Complexité algorithmique : $\simeq 2/3n^3 + n^2 + o(n^2)$ flop

Résolution de systèmes triangulaires

- ▶ On arrive au système triangulaire supérieur

$$\begin{aligned}u_{11}x_1 + u_{12}x_2 + \cdots + u_{1n}x_n &= b_1 \\u_{22}x_2 + \cdots + u_{2n}x_n &= b_2 \\&\vdots \\u_{nn}x_n &= b_n\end{aligned}$$

- ▶ Résolution par substitution rétrograde (*backward subst., row version*)

$$\begin{aligned}x_n &= \frac{b_n}{u_{nn}} \\x_i &= \frac{1}{u_{ii}} \left(b_i - \sum_{j=i+1}^n u_{ij}x_j \right), \quad i = n-1, \dots, 1\end{aligned}$$

- ▶ Complexité algorithmique : n^2 flop

Plan

- 1 Quelques rappels d'algèbre linéaire
- 2 **Méthodes directes**
 - Factorisation LU
 - Calculs à virgule flottante
 - Autres factorisations classiques
 - Systèmes creux
- 3 Aperçu des méthodes itératives
- 4 Aperçu d'une méthode hybride

Élimination de Gauss & factorisation **LU**

Theorem

Soit **A** inversible, l'élimination de Gauss donne

$$\mathbf{PA} = \mathbf{LU}$$

où **P** est une matrice de permutation, **L** est triangulaire inférieure à diagonale unitaire et **U** est triangulaire supérieure.

► Remarque : Matrice de permutation

$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Rightarrow \mathbf{P} \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} = \begin{pmatrix} l_2 \\ l_1 \\ l_3 \end{pmatrix}$$

Élimination de Gauss & factorisation LU

- ▶ Remarque : suivant les propriétés de \mathbf{A} , l'absence de pivot nul est garantie (matrice à diagonale dominante, matrice SDP)
- ▶ En l'absence de pivot nul, l'élimination de Gauss donne

$$\mathbf{A} = \mathbf{LU} \quad \text{avec} \quad \mathbf{L} = \begin{pmatrix} 1 & & & \\ m_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ m_{n1} & \dots & m_{nn-1} & 1 \end{pmatrix} \quad \text{et} \quad \mathbf{U} = \begin{pmatrix} u_{11} & \dots & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{pmatrix}$$

- ▶ On pose $\mathbf{M}_k = \mathbf{I}_n - \mathbf{m}_k \mathbf{e}_k^\top$ avec $\mathbf{m}_k = [0, \dots, 0, m_{k+1,k}, \dots, m_{n,k}]^\top$
- ▶ On remarque $\mathbf{M}_k^{-1} = \mathbf{I}_n + \mathbf{m}_k \mathbf{e}_k^\top$
- ▶ Démonstration

Élimination de Gauss & factorisation **LU**

En pratique, on veut souvent résoudre plusieurs second membres. La substitution à la volée de **b** n'est pas souhaitée

- 1 Calcul de la factorisation **LU**

$$LU = PA$$

Alors

$$LUx = PAx = Pb$$

- 2 Résolution d'un système tri. inf. à diagonale unitaire

$$Ly = Pb \quad (\text{descente, } forward \text{ substitution})$$

- 3 Résolution d'un système tri. supérieur

$$Ux = y \quad (\text{remontée, } backward \text{ substitution})$$

Élimination de Gauss & factorisation **LU**

Theorem (Existence et unicité)

Soit la matrice $\mathbf{A} \in \mathbb{R}^{n \times n}$. La factorisation **LU** existe et est unique ssi toutes les sous-matrices \mathbf{A}_i d'ordre $i = 1, \dots, n - 1$ sont inversibles.

- ▶ Si \mathbf{A} est inversible, on peut toujours se ramener au cas ci-dessus grâce à des permutations.

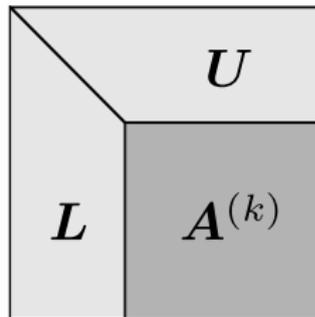
Complexité

- ▶ Coût du calcul de **LU** $\simeq 2/3n^3 + o(n^3)$
- ▶ Coût de résolution d'un système triangulaire $\simeq n^2$
- ▶ Coût de stockage, en dense il suffit d'un vecteur suppl. pour \mathbf{P} .

Élimination de Gauss & factorisation LU

Algorithme LU de base sans changement de pivot

```
def lu_kji(A):  
    """ Factorisation LU (ver. kji) sans pivot """  
    n, m = A.shape  
    if n != m:  
        raise ValueError("Only square matrix allowed")  
    for k in xrange(0, n):  
        if abs(A[k,k]) < 1.0e-20:  
            raise ArithmeticError("Null pivot")  
        A[k+1:,k] /= A[k,k] # i = k+1:n  
        for j in xrange(k+1,n):  
            A[k+1:,j] -= A[k+1:, k] * A[k,j]  
    return A
```



Remarques : raisonnement simplifié

- ▶ Arithmétique exacte (spécifités du calcul sur ordinateur)
- ▶ Algèbre linéaire dense (spécifités des EDP)

Élimination de Gauss et calculs à virgule flottante

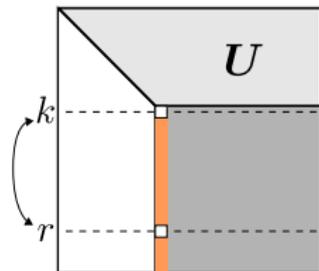
Pivot partiel $PA = LU$

- ▶ Exemple ($k \ll 1$)

$$\mathbf{A} = \begin{bmatrix} k & 10 \\ 10 & 10 \\ 0 & 01 \end{bmatrix} \quad \mathbf{x}_{\text{ex}} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \Rightarrow \quad \mathbf{b} = \begin{bmatrix} k+1 \\ 11 \\ 1 \end{bmatrix}$$

- ▶ Au lieu de se satisfaire de $a_{kk}^{(k)} \neq 0$, on permute la ligne i qui maximise $|a_{ik}^{(k)}|, i \geq k$

- ▶ Requiert $(n - k)$ tests
- ▶ N'affecte pas les lignes de \mathbf{U} déjà factorisées

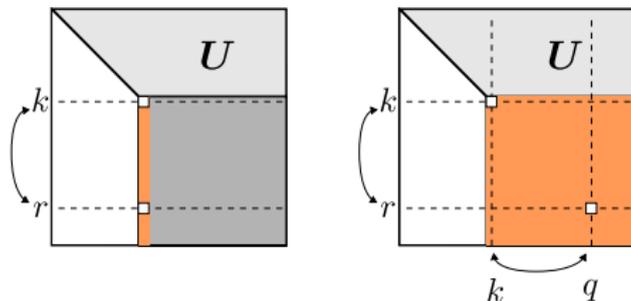


Élimination de Gauss et calculs à virgule flottante

Pivot total $PAQ = LU$

- ▶ Si le pivot partiel n'est pas suffisant, il existe le pivotage total (ou complet).

$$PAQ^T = LU$$



- ▶ Requiert $(n - k)^2$ tests

Calculs à virgule flottante : le scaling

- ▶ La présence de grands pivots ne suffit pas à garantir la précision de la solution.
- ▶ Exemple :

$$\mathbf{A} = \begin{bmatrix} -10^9 & 10^8 & 10^8 \\ 10^8 & 10^{-6} & 0 \\ 10^8 & 0 & 1 \end{bmatrix} \quad \mathbf{x}_{\text{ex}} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \Rightarrow \mathbf{b} = \mathbf{Ax} \quad \mathbf{x}_{\text{num}} = \begin{bmatrix} 1. \\ 0.9981378 \\ 1.0018622 \end{bmatrix}$$

- ▶ Le scaling consiste à utiliser des matrices diagonales $\mathbf{D}_1, \mathbf{D}_2$ inversible pour uniformiser les ordres de grandeurs

$$\mathbf{Ax} = \mathbf{b} \Leftrightarrow \begin{cases} \mathbf{D}_1 \mathbf{A} \mathbf{D}_2 \mathbf{x}^* = \mathbf{D}_1 \mathbf{b} \\ \mathbf{x} = \mathbf{D}_2 \mathbf{x}^* \end{cases}$$

Calculs à virgule flottante : le raffinement itératif

- ▶ Pour les matrices mal conditionnées, la méthode du raffinement itératif permet d'améliorer la qualité de la solution

- ▶ Exemple :

Algorithme 1 : Raffinement itératif

Données : ϵ tolérance utilisateur, \mathbf{A} , \mathbf{b}

Entrées : $\mathbf{x}^{(0)} = \text{gauss}(\mathbf{A}, \mathbf{b})$, $i = 0$

répéter

$$\mathbf{r}^{(i)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(i)}$$

$$\mathbf{z} = \text{gauss}(\mathbf{A}, \mathbf{r}^{(i)})$$

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \mathbf{z}$$

jusqu'à $\|\mathbf{z}\| \leq \epsilon \|\mathbf{x}^{(i+1)}\|$

$$\mathbf{A} = \begin{bmatrix} a & b & c \\ a & b & c \\ a & b & c \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

- ▶ Solution exacte :

$$\mathbf{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Autres factorisations classiques

- ▶ Variante de la factorisation **LU** :

$$\mathbf{LDU}^* = \mathbf{PA}$$

Avec **D** diagonale et **U**^{*} tri. sup. à diagonale unitaire.

- ▶ Ces factorisations ne profitent pas des propriétés de **A**.
- ▶ Si **A** est symétrique \Rightarrow factorisation de Crout (pivotage symétrique)

$$\mathbf{LDL}^T = \mathbf{PAP}^T$$

Coût du calcul $\simeq 1/3n^3$

- ▶ Si **A** est SDP, tous les termes $\mathbf{D}_{ii} > 0 \Rightarrow$ factorisation de Cholesky.

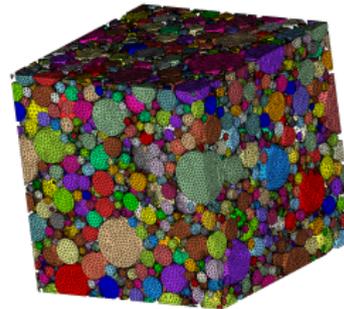
$$\mathbf{L}_c \mathbf{L}_c^T = \mathbf{A}$$

Coût du calcul $\simeq 1/3n^3$, pas besoin de pivoter *a priori*.

Est-ce que la complexité compte ?

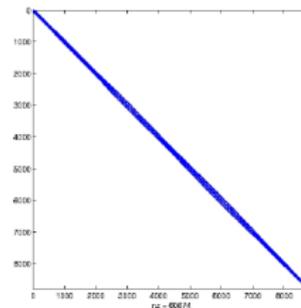
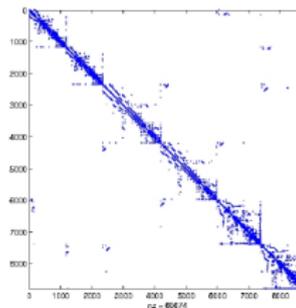
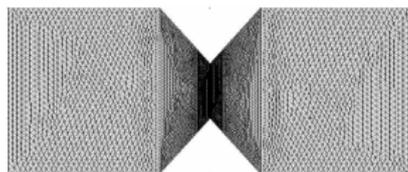
- En **dense**, coût d'une facto **LU** et temps de calcul

n	Flop/s de l'ordinateur		
	10^9	10^{12}	10^{15}
10^4	10 min	1 sec	$1 \mu s$
10^6	20 ans	7 mois	10 min
10^8	20 ans



$\geq 10^8$ dofs

- Heureusement les systèmes sont souvent **creux**



Source : Analyse des solides déformables par la méthode des éléments finis, Bonnet, Frangi

Remarques & références

▶ Remarques

- ▶ Sans info. supplémentaire, les méthodes solve utilisent une **LU**
- ▶ Si vous connaissez les propriétés de **A**, utilisez les bonnes méthodes !
- ▶ Ne codez pas vos propres méthodes (sauf si ça vous amuse)

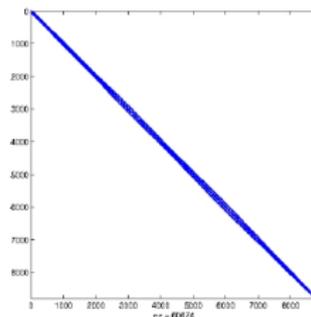
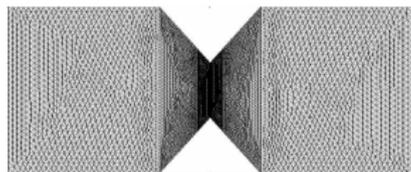
▶ Quelques références

- ▶ Langages interprétés | Python Scipy : `scipy.linalg`
Octave / Matlab : `lu`
- ▶ Bibliothèques compilées
Eigen <http://eigen.tuxfamily.org/>
GNU Scientific library <http://www.gnu.org/software/gsl/>
LAPACK <http://www.netlib.org/lapack/>

Voir aussi http://en.wikipedia.org/wiki/Comparison_of_linear_algebra_libraries

Systèmes creux

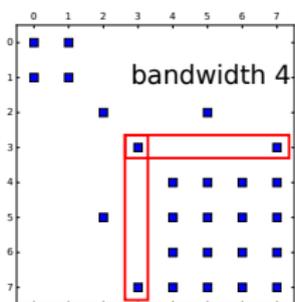
- ▶ Les systèmes provenant d'EDPs sont souvent creux
- ▶ On parle de systèmes creux quand le nombre de valeur non nulle de \mathbf{A} est en $O(n)$
- ▶ Exploiter le caractère creux du système permet de diminuer considérablement le coût de calcul d'une factorisation
- ▶ Certaines matrices creuses ont des structures particulières comme les matrices bandes ou blocs



Source : Analyse des solides déformables par la méthode des éléments finis, Bonnet, Frangi

Matrices bandes

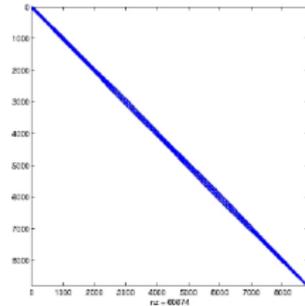
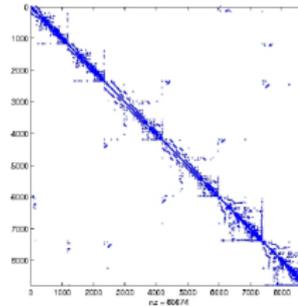
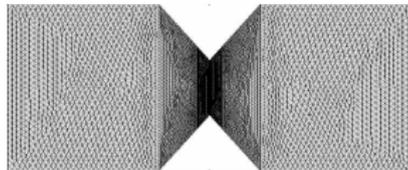
- ▶ \mathbf{A} a une largeur de bande inférieure p si $a_{ij} = 0$ quand $i > j + p$
- ▶ \mathbf{A} a une largeur de bande supérieure q si $a_{ij} = 0$ quand $j > i + q$
- ▶ Propriété :
Soit \mathbf{A} inversible., la factorisation LU existe et \mathbf{L} a une largeur de bande inférieur p , \mathbf{U} a une largeur de bande supérieure q



- ▶ On peut ne stocker que la bande
- ▶ Calcul uniquement des valeurs dans la bande
- ▶ Complexité de la factorisation en $O(2npq)$
- ▶ Complexité des substitutions en $O(np)$ et $O(nq)$

Stockage creux

- Pour une matrice bande, on stocke la bande dans une matrice dense



Source : Analyse des solides déformables par la méthode des éléments finis, Bonnet, Frangi

- Dans le cas général, \neq stockages sont possibles (COO, CSR, CSC, ...)
- Exemple du stockage COO (i, j, a_{ij})

$$i = [1 \ 1 \ 2 \ 3 \ 4]$$

$$j = [1 \ 2 \ 2 \ 3 \ 2]$$

$$a_{ij} = [11 \ 45 \ 6 \ 22 \ 3]$$

$$n = 4$$

$$m = 4$$

$$\text{nnz} = 5$$

$$\mathbf{A} = \begin{bmatrix} 11 & 45 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 22 & 0 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

- nnz nombre de valeurs non nulles

Factorisation et stockage creux

- ▶ On considère la factorisation LU de la matrice **A** creuse
- ▶ Les matrices **L** et **U** sont en général creuses également
- ▶ Elles sont par contre (beaucoup) plus remplies

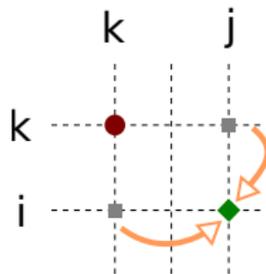
$$nnz(\mathbf{A}) \ll nnz(\mathbf{L}) \quad \text{et} \quad nnz(\mathbf{A}) \ll nnz(\mathbf{U})$$

- ▶ Le remplissage ou *fill-in* est la différence de nnz entre **A** et **L** ou **U**
- ▶ Le *fill-in* augmente la mémoire nécessaire et le coût de la factorisation
- ▶ En plus de toutes les problématiques vues précédemment (changement de pivots, scaling, ...), il faut veiller à minimiser ce remplissage

Source du *fill-in*

- Retour sur l'élimination de Gauss

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}}$$



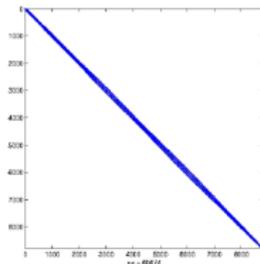
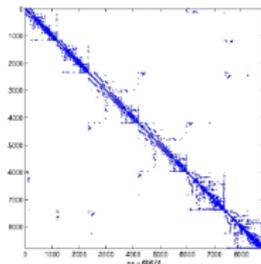
- À l'étape $k + 1$, $a_{ij}^{(k+1)} \neq 0$ si $\begin{cases} a_{ij}^{(k)} \neq 0 \\ a_{ik}^{(k)} \neq 0 \text{ et } a_{kj}^{(k)} \neq 0 \text{ (structural fill-in)} \end{cases}$

- Exemple de la matrice flèche

$$\begin{pmatrix} x & x & x & x \\ x & x & & \\ x & x & & \\ x & x & & \end{pmatrix} \Rightarrow \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix} \quad \text{mais} \quad \begin{pmatrix} x & & x \\ & x & x \\ & & x & x \\ x & x & x & x \end{pmatrix} \Rightarrow \begin{pmatrix} * & & * \\ & * & * \\ & & * & * \\ * & * & * & * \end{pmatrix}$$

Analyse symbolique

- ▶ Permuter des lignes et des colonnes de \mathbf{A} permet de réduire significativement le remplissage
- ▶ On factorise $\mathbf{A}^* = \mathbf{P}_1 \mathbf{A} \mathbf{Q}_1$ où \mathbf{P}_1 et \mathbf{Q}_1 sont des matrices de permutations.
- ▶ La phase d'**analyse symbolique** vise à trouver \mathbf{P}_1 et \mathbf{Q}_1 pour minimiser le remplissage
- ▶ Cette phase s'appuie uniquement sur le profile de \mathbf{A}
- ▶ Elle fait appel à de notions complexes de théorie des graphes
- ▶ Exemple : algorithme de type Cuthill–McKee pour minimiser la largeur de bande de \mathbf{A}^*



Complexité en $O(m \log(m) nnz)$ avec m le plus haut degré du graphe.

Factorisation et stockage creux

- ▶ Les permutations ont un rôle double
 - ▶ Garantir la stabilité et limiter la propagation des erreurs d'arrondis
 - ▶ Limiter le remplissage
- ▶ La factorisation de $\mathbf{A}^* = \mathbf{P}_1 \mathbf{A} \mathbf{Q}_1$ peut nécessiter des changements de pivots
- ▶ Il faut veiller à ne augmenter trop le remplissage

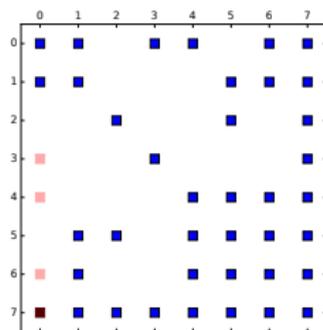
$$\mathbf{LU} = \mathbf{P}_2 \mathbf{A}^* \mathbf{Q}_2 = \mathbf{P}_2 \mathbf{P}_1 \mathbf{A} \mathbf{Q}_1 \mathbf{Q}_2$$

- ▶ Pivots candidats (*threshold pivot*) :
 au lieu de choisir

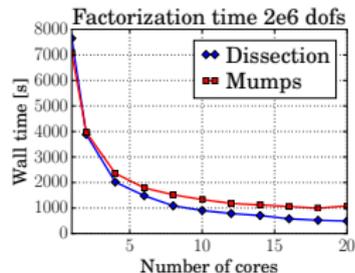
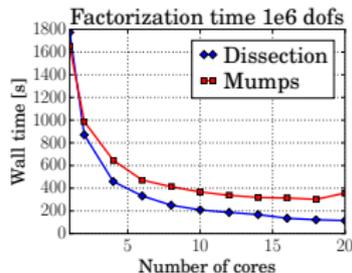
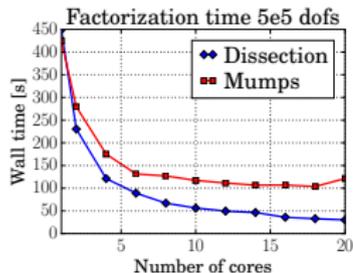
$$p = \operatorname{argmax}_{i \geq k} (|a_{ik}^{(k)}|)$$

on choisit le pivot qui minimise le *fill-in*
 parmi les pivots candidats vérifiant :

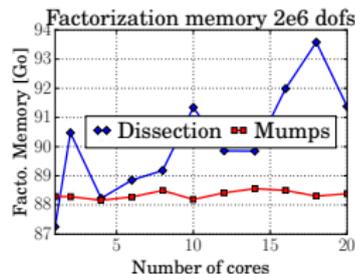
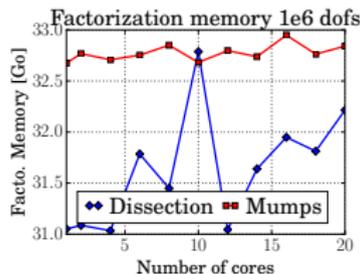
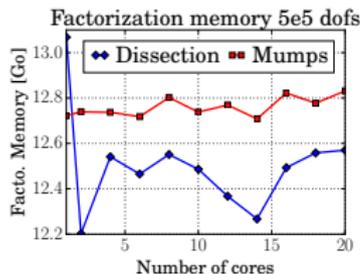
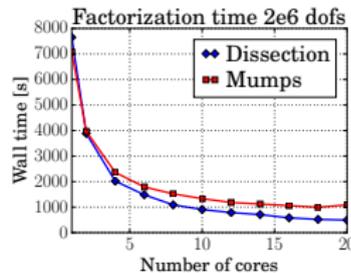
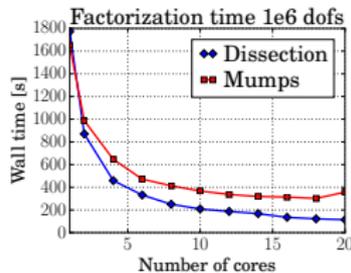
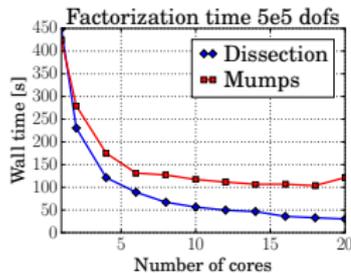
$$|a_{ik}^{(k)}| \geq \tau | \max_{pk} a_{pk}^{(k)} |$$



Exemple : cube élastique linéaire (c3d20)



Exemple : cube élastique linéaire (c3d20)



Synthèse

Les solveurs directs sont

- ▶ des variantes de l'élimination de Gauss
- ▶ ils fonctionnent en 3 phases

Fact. symbolique \Rightarrow Fact. numérique \Rightarrow descente-remontée

- ▶ robustes : ils fournissent la solution exacte (arithmétique exacte) en un nombre fini d'opération (qui peut être grand)
- ▶ coûteux en mémoire (les ressources croissent fortement avec la taille du problème)
- ▶ le conditionnement de \mathbf{A} influe uniquement sur la qualité de la solution (remèdes : pivot, scaling, raffinement itératif)

Remarques & références

- ▶ Remarques
 - ▶ Si vous connaissez les propriétés de \mathbf{A} , utilisez les bonnes méthodes !
 - ▶ Ne codez pas vos propres méthodes (sauf si c'est votre métier)
- ▶ Quelques références
 - ▶ Langages interprétés
 - Python Scipy :
`scipy.sparse & scipy.sparse.linalg`
 - Matlab :
<http://fr.mathworks.com/help/matlab/sparse-matrices.html>
 - ▶ Librairies compilées
 - MULTifrontal Massively Parallel sparse direct Solver (MUMPS)
<http://mumps.enseiht.fr/>
 - Supernodal LU (SuperLU)
<http://crd-legacy.lbl.gov/~xiaoye/SuperLU/>
 - Parallel Sparse matrix package (PaStiX)
<http://pastix.gforge.inria.fr/>

Plan

- 1 Quelques rappels d'algèbre linéaire
- 2 Méthodes directes
- 3 Aperçu des méthodes itératives**
 - Méthodes itératives stationnaires
 - Méthodes itératives de type Krylov
- 4 Aperçu d'une méthode hybride

Pourquoi utiliser des méthodes itératives ?

- ▶ Les solveurs directs sont robustes mais très gourmand en mémoire
- ▶ La parallélisation est possible mais pas évidente (échange de complément de Schur)
- ▶ Pour les très gros problèmes il faut penser aux méthodes itératives ou hybrides
- ▶ **En dense**, un produit MV en $O(n^2)$ \Rightarrow uniquement utile pour le creux
- ▶ Solveurs itératifs presque *embarrassingly parallel*

$$\mathbf{Ax} \Rightarrow \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$$

Méthodes itératives stationnaires

Principe

- ▶ On souhaite résoudre $\mathbf{Ax} = \mathbf{b}$ avec \mathbf{A} très grande et creuse
- ▶ Construire une suite de vecteurs tels que $\lim_{k \rightarrow +\infty} \mathbf{x}_k = \mathbf{A}^{-1}\mathbf{b} = \mathbf{x}^*$
- ▶ Soit \mathbf{x}_k l'approximation au pas k

$$\mathbf{x}^* = \mathbf{x}_k + \mathbf{e}_k \quad (\text{erreur})$$

$$\mathbf{Ae}_k = \mathbf{Ax}^* - \mathbf{Ax}_k := \mathbf{r}_k \quad (\text{résidu}) \quad \text{d'où} \quad \mathbf{x}^* = \mathbf{x}_k + \mathbf{A}^{-1}\mathbf{r}_k$$

- ▶ Idée : remplacer \mathbf{A} par une matrice proche mais facilement inversible
- ▶ Une méthode itérative est convergente ssi la suite $(\mathbf{x}_n)_n \rightarrow \mathbf{x}^* \forall \text{init. } \mathbf{x}_0$
- ▶ Choix du critère d'arrêt $\left\{ \begin{array}{l} \|\mathbf{r}_k\| \leq \epsilon \|\mathbf{r}_0\| \quad (\text{résidu}) \\ \|\mathbf{x}_{k+1} - \mathbf{x}_k\| \leq \epsilon \|\mathbf{x}_k\| \quad (\text{stagnation}) \end{array} \right.$

Méthodes itératives stationnaires

Principe

- ▶ Soit \mathbf{M} une matrice inversible qui
 - ▶ est une bonne approximation de \mathbf{A}
 - ▶ soit facile à calculer
 - ▶ permette de résoudre facilement le système $\mathbf{Mz} = \mathbf{r}$
- ▶ \mathbf{M} est appelé **préconditionneur**
- ▶ Au lieu de résoudre $\mathbf{x}^* = \mathbf{x}_k + \mathbf{A}^{-1}\mathbf{r}_k$, on itère :

$$\begin{aligned} \mathbf{r}_k &= \mathbf{b} - \mathbf{A}\mathbf{x}_k \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{M}^{-1}\mathbf{r}_k \end{aligned}$$

- ▶ Trois étapes :
calcul de résidu \Rightarrow résolution du pb préconditionné \Rightarrow maj solution

Préconditionnements de type décomposition

- ▶ Décomposer la matrice $\mathbf{A} = \mathbf{M} - \mathbf{N}$ où \mathbf{M} est « facilement inversible »
- ▶ Suivant le choix de \mathbf{M} on obtient \neq méthodes avec des propriétés de convergence différentes
- ▶ Exemples : si $\mathbf{A} = \mathbf{D} - \mathbf{E} - \mathbf{F}$ avec $\left. \begin{array}{l} \mathbf{D} \text{ diagonal} \\ \mathbf{E} \text{ tri. inf. stricte} \\ \mathbf{F} \text{ tri. sup. stricte} \end{array} \right\}$
- ▶ $\mathbf{M} = \mathbf{D} \Rightarrow$ méthode de Jacobi (convergence garantie si A est à diag. dom. stricte)
- ▶ $\mathbf{M} = \mathbf{D} - \mathbf{E} \Rightarrow$ méthode de Gauss-Seidel (convergence garantie si A est SPD)

Méthodes itératives de type Krylov

- ▶ Avec les méthodes itératives stationnaires précédentes

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{M}^{-1}\mathbf{r}_{k-1} = \mathbf{x}_{k-1} + \mathbf{z}_{k-1}$$

- ▶ $\mathbf{z}_k = Q_k(\mathbf{M}^{-1}\mathbf{A})\mathbf{z}_0$ où Q_k est le polynôme $Q_k(X) = (1 - X)^k$

$$Q_k \in \mathcal{K}_k(\mathbf{M}^{-1}\mathbf{A}, \mathbf{z}_0) = \text{Vect}(\mathbf{z}_0, (\mathbf{M}^{-1}\mathbf{A})\mathbf{z}_0, \dots, (\mathbf{M}^{-1}\mathbf{A})^{k-1}\mathbf{z}_0)$$

- ▶ $\mathcal{K}_k(\mathbf{M}^{-1}\mathbf{A}, \mathbf{z}_0)$ est l'espace de Krylov généré par le couple $(\mathbf{M}^{-1}\mathbf{A}, \mathbf{z}_0)$
- ▶ Les méthodes de type Krylov ajoutent une contrainte sur \mathbf{z}_k tel que

$$\begin{cases} \mathbf{x}_k \in \mathbf{x}_0 + \mathcal{K}_k(\mathbf{M}^{-1}\mathbf{A}, \mathbf{z}_0) \\ \mathbf{z}_k \perp \mathcal{K}_k(\mathbf{M}^{-1}\mathbf{A}, \mathbf{z}_0) \end{cases}$$

Méthodes itératives de type Krylov

- ▶ Les méthodes de type Krylov ajoutent une contrainte sur \mathbf{r}_m tel que

$$\begin{cases} \mathbf{x}_m \in \mathbf{x}_0 + \mathcal{K}_m(\mathbf{A}, \mathbf{r}_0) \\ \mathbf{r}_m \perp \mathcal{K}_m(\mathbf{A}, \mathbf{r}_0) \end{cases}$$

- ▶ Suivant les propriétés de \mathbf{A} , le choix du type l'orthogonalité permet de définir plusieurs approches (CG, GMRes, OrthoDir, BiCG, CGS, ...)
- ▶ Si \mathbf{A} est SDP le Gradient Conjugué (CG) est la meilleure solution

$$\begin{cases} \mathbf{x}_m \in \mathbf{x}_0 + \mathcal{K}_m(\mathbf{K}, \mathbf{r}_0) \\ \mathbf{r}_m \perp \mathcal{K}_m(\mathbf{K}, \mathbf{r}_0) \end{cases}$$

- ▶ Avec le CG, \mathbf{x}_k minimise la \mathbf{A} -norme de l'erreur sur l'espace de Krylov.
- ▶ Dans le contexte parallèle, l'expression de la contrainte nécessite des communications globales

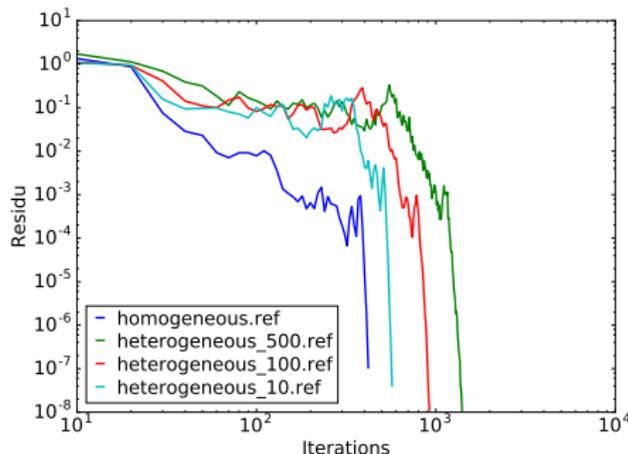
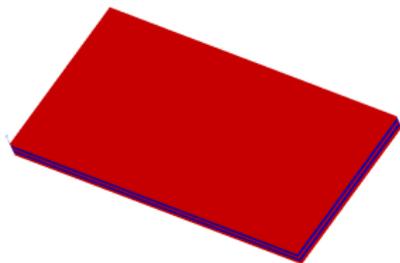
Quelques exemples

► Matrice symétrique

```
def example_generator(n=100, k=1e3, off=1.):  
    """ Generate an example and plot the results """  
    b = np.random.rand((n))  
    L = np.random.rand((n*n)); L.shape = n, n  
    A = - (L + L.transpose())  
    A += k * np.diag(np.random.rand(n) + off)  
  
    P = A.diagonal()  
def prec(r, z):  
    z[:] = r/P; return z
```

Exemple provenant d'une EDP et conditionnement

- ▶ Plaque lamifiée élastique linéaire $n \simeq 21\,000$,
- ▶ Solveur : Gradient conjugué
- ▶ Préconditionneur : factorisation de cholesky incomplète



Synthèse

- ▶ Les méthodes itératives nécessitent très peu de mémoire
- ▶ Elles peuvent donc résoudre de très gros problèmes
- ▶ Choisir un bon préconditionneur n'est pas toujours évident
- ▶ La vitesse de convergence dépend du conditionnement de l'opérateur à résoudre
- ▶ Les méthodes de Krylov convergent généralement plus rapidement
- ▶ Elles requièrent cependant plus de communications

Plan

- 1 Quelques rappels d'algèbre linéaire
- 2 Méthodes directes
- 3 Aperçu des méthodes itératives
- 4 **Aperçu d'une méthode hybride**
 - Méthodes de décomposition de domaine

Méthodes hybrides

- ▶ Les méthodes modernes combinent méthodes directes et itératives
- ▶ Objectif : combiner les avantages de chaque approche
- ▶ Exemples :
 - ▶ Méthodes itératives par blocs
solveur itératif stationnaire
méthodes directes sur les sous-blocs
 - ▶ Méthodes itératives avec préconditionneur par facto. incomplète
 - ▶ Méthode de type décomposition de domaine
solveurs directs pour les problèmes locaux
solveurs de Krylov pour équilibrer l'interface
 - ▶ Méthodes multi-grilles
solveurs itératifs stationnaires
préconditionneur avec une méthode DD

Méthodes hybrides

- ▶ Les méthodes modernes combinent méthodes directes et itératives
- ▶ Objectif : combiner les avantages de chaque approche
- ▶ Exemples :
 - ▶ Méthodes itératives par blocs
solveur itératif stationnaire
méthodes directes sur les sous-blocs
 - ▶ Méthodes itératives avec préconditionneur par facto. incomplète
 - ▶ **Méthode de type décomposition de domaine**
solveurs directs pour les problèmes locaux
solveurs de Krylov pour équilibrer l'interface
 - ▶ Méthodes multi-grilles
solveurs itératifs stationnaires
préconditionneur avec une méthode DD

Méthodes de décomposition de domaine

- ▶ Système global

$$\mathbf{K}\mathbf{u} = \mathbf{f} \quad \text{avec} \quad \mathbf{K} \text{ SDP}$$

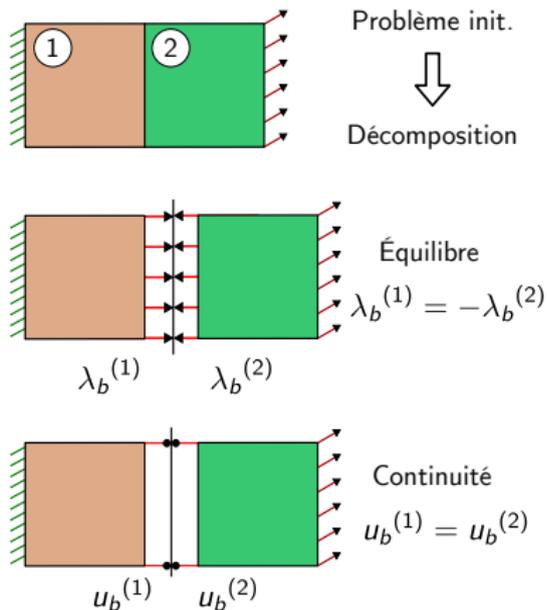
- ▶ Décompo. sans recouvrement

$$\mathbf{K}^{(s)}\mathbf{u}^{(s)} = \mathbf{f}^{(s)} + \boldsymbol{\lambda}^{(s)}, \quad s = 1, 2$$

$$\boldsymbol{\lambda}_b^{(1)} + \boldsymbol{\lambda}_b^{(2)} = \mathbf{0}$$

$$\mathbf{u}_b^{(1)} - \mathbf{u}_b^{(2)} = \mathbf{0}$$

- ▶ Problème global \Leftrightarrow
 - Équilibre locaux
 - Équilibre interface
 - Continuité interface



Décomposition en deux sous domaines

On suppose les K_{ii}^s inversibles.

$$\begin{pmatrix} K_{ii}^{(1)} & 0 & K_{ib}^{(1)} \\ 0 & K_{ii}^{(2)} & K_{ib}^{(2)} \\ K_{bi}^{(1)} & K_{bi}^{(2)} & K_{bb}^{(1)} + K_{bb}^{(2)} \end{pmatrix} = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ K_{bi}^{(1)} & K_{ii}^{(1)-1} & K_{bi}^{(2)} & K_{ii}^{(2)-1} & I \end{pmatrix} \begin{pmatrix} K_{ii}^{(1)} & 0 & 0 \\ 0 & K_{ii}^{(2)} & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} I & 0 & K_{ii}^{(1)-1} & K_{ib}^{(1)} \\ 0 & I & K_{ii}^{(2)-1} & K_{ib}^{(2)} \\ 0 & 0 & I & \end{pmatrix}$$

Complément de Schur

$$\begin{aligned} S &= K_{bb} - K_{bi}^{(1)} K_{ii}^{(1)-1} K_{ib}^{(1)} - K_{bi}^{(2)} K_{ii}^{(2)-1} K_{ib}^{(2)} \\ &= \left(K_{bb}^{(1)} - K_{bi}^{(1)} K_{ii}^{(1)-1} K_{ib}^{(1)} \right) + \left(K_{bb}^{(2)} - K_{bi}^{(2)} K_{ii}^{(2)-1} K_{ib}^{(2)} \right) \end{aligned}$$

Décomposition en deux sous domaines

$$\mathbf{K} \mathbf{u} = \mathbf{f}$$

$$\begin{pmatrix} \mathbf{K}_{ii}^1 & 0 & 0 \\ 0 & \mathbf{K}_{ii}^2 & 0 \\ 0 & 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{I} & 0 & \mathbf{K}_{ii}^{1-1} & \mathbf{K}_{ib}^1 \\ 0 & \mathbf{I} & \mathbf{K}_{ii}^{2-1} & \mathbf{K}_{ib}^2 \\ 0 & 0 & \mathbf{I} & \end{pmatrix} \begin{pmatrix} \mathbf{u}_i^1 \\ \mathbf{u}_i^2 \\ \mathbf{u}_b \end{pmatrix} = \begin{pmatrix} \mathbf{I} & 0 & 0 \\ 0 & \mathbf{I} & 0 \\ -\mathbf{K}_{bi}^1 \mathbf{K}_{ii}^{1-1} & -\mathbf{K}_{bi}^2 \mathbf{K}_{ii}^{2-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{f}_i^1 \\ \mathbf{f}_i^2 \\ \mathbf{f}_b \end{pmatrix}$$

Algorithme 2 : Méthode de décomposition de domaine, principe de base

Factorisation parallèle de $\mathbf{K}_{ii}^{(s)}$

Calcul parallèle des RHS condensé $\mathbf{b}^s = \mathbf{f}_b^s - \mathbf{K}_{bi}^s \mathbf{K}_{ii}^{s-1} \mathbf{f}_i^s$ puis assemblage avec son voisin

Résolution du système d'interface $\mathbf{S} \mathbf{u}_b = \mathbf{b}$ via un solveur itératif

Descente remontée parallèle de $(\mathbf{K}_{ii}^{(s)})^{-1} \Rightarrow \mathbf{u}_i^s = \mathbf{K}_{ii}^{s-1} (\mathbf{f}_i^s - \mathbf{K}_{bi}^s \mathbf{u}_b)$

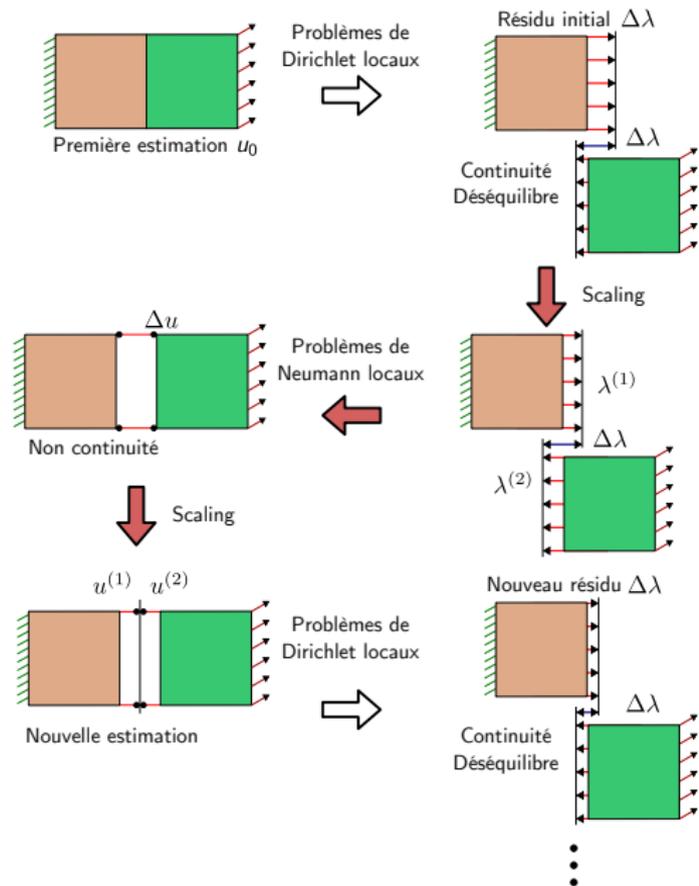
Méthode Balancing Domain Decomposition (BDD)

- ▶ L'opérateur est une somme de compléments de Schur locaux

$$\mathbf{S}u_b = (\mathbf{S}^{(1)} + \mathbf{S}^{(2)})u_b$$

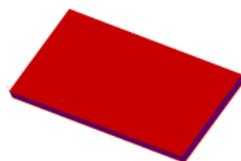
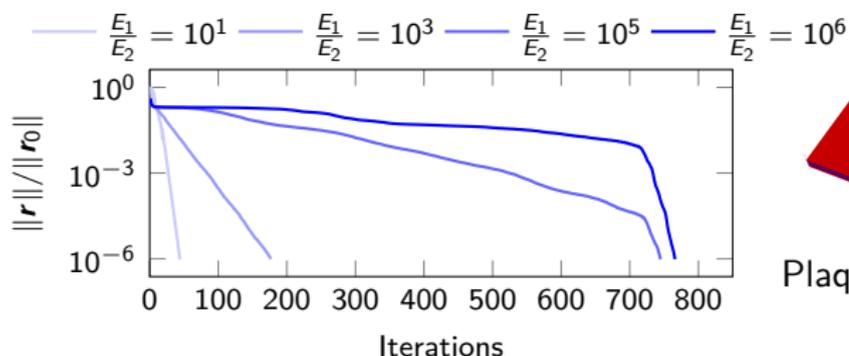
- ▶ Pour le préconditionneur, on approche l'inverse de la somme par la somme des inverses

$$\mathbf{M} = (\mathbf{S}^{(1)+} + \mathbf{S}^{(2)+})$$



Méthode BDD

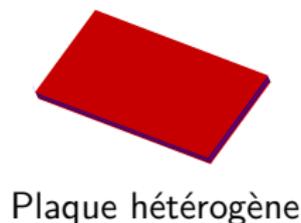
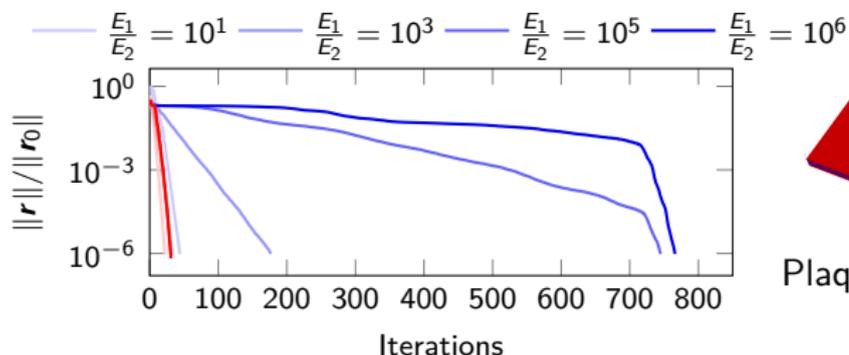
- ▶ Les méthodes DD classiques restent sensibles au conditionnement du problème
- ▶ Des solutions existent (multipreconditionnement, augmentation)



Plaque hétérogène

Méthode BDD

- ▶ Les méthodes DD classiques restent sensibles au conditionnement du problème
- ▶ Des solutions existent (**multipreconditionnement**, augmentation)



Étude d'extensibilité faible hétérogène (élasticité linéaire)

Paramètres $\tau = 0.01$, $E_1/E_2 = 10^6$, $H/h = 29$, (Occigen cluster, 4Gb/core)

Adaptive Multipréconditionné FETI

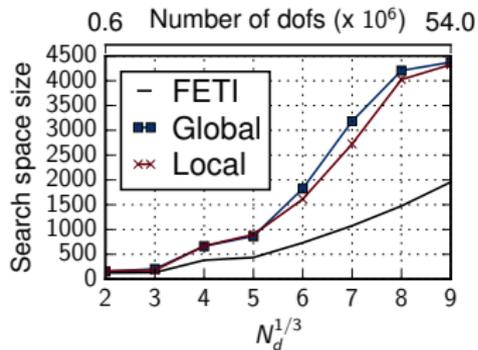
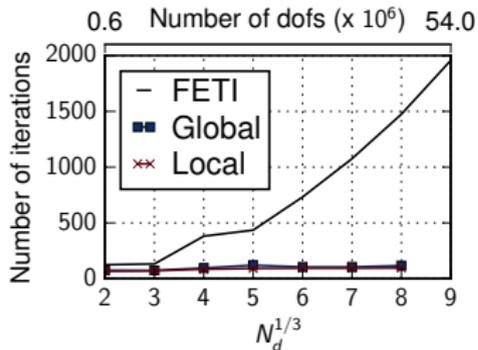
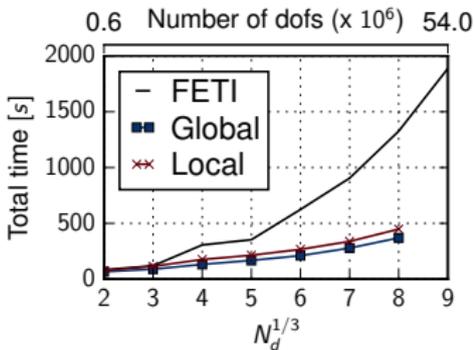
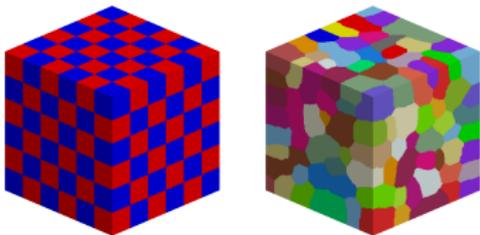
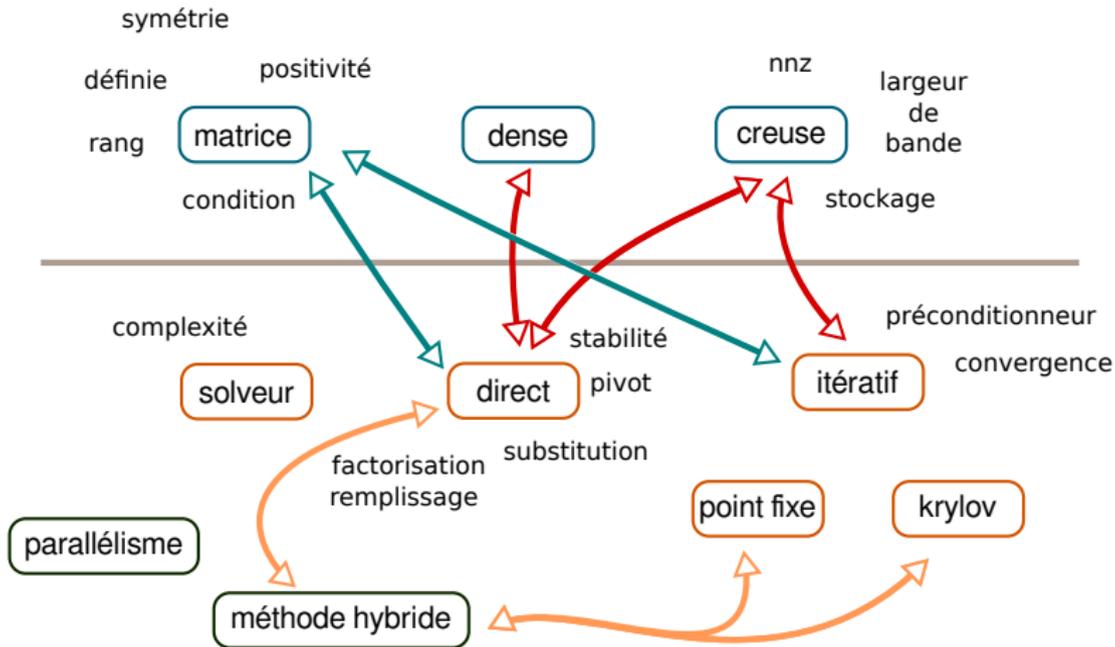


Schéma synoptique



Références

- ▶ Méthodes numériques en général
 - ▶ Quarteroni, A. M., Sacco, R., & Saleri, F. (2008). Méthodes numériques : algorithmes, analyse et applications. Springer Science & Business Media.
- ▶ Algèbre linéaire dense
 - ▶ Golub & van Loan : Matrix Computations, 3rd ed., Johns Hopkins, 1996.
- ▶ Solveurs directs creux
 - ▶ I. Duff, A. Erisman, J. Reid : Direct Methods for Sparse Matrices, Oxford University Press, 1986.
 - ▶ T. Davis : Direct Methods for Sparse Linear Systems, SIAM, 2006.
- ▶ Solveurs de Krylov
 - ▶ Y. Saad : Iterative Methods for Sparse Linear Systems, 2nd ed., pp. 103–128, SIAM, 2003.
 - ▶ Van der Vorst, H. A. (2003). Iterative Krylov methods for large linear systems (Vol. 13). Cambridge University Press.