Résolution numérique de systèmes linéaires

Benoit Fabrèges

04 avril 2019

Quelques références

- A. Quarteroni, R. Sacco, F. Saleri, **Méthodes numériques : Algorithmes, analyse et applications**.
  Niveau L3, avec des rappels. Disponible en pdf sur le site de la bibliothèque.
- P. G. Ciarlet, **Introduction à l'analyse numérique matricielle et à l'optimisation**.
  Niveau L3-M1. Première partie sur les systèmes linéaires.
- G. H. Golub, C. F. Van Loan, **Matrix computations**.
  Complet, plus compliqué.

Sommaire

Sommaire

Rappel sur les matrices

Quelques définitions :

- Une matrice A d'ordre n est dite ***inversible*** s'il existe une matrice B d'ordre n telle que $AB = BA = I_n$ (I_n est la matrice identité d'ordre n).
  Exemple : matrice de rotation d'angle θ

$$A = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \qquad B = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

- On appelle ***transposée*** d'une matrice $A \in \mathbb{R}^{n\times m}$ la matrice $m \times n$, notée A^T, obtenue en échangeant les lignes et les colonnes de A.
  Exemple : pour une matrice générale de taille 2×3

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}, \qquad A^T = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{pmatrix}$$

Rappel sur les matrices

- On a les relations suivantes :

$$\left(A^T\right)^T = A, \quad (A+B)^T = A^T + B^T,$$
$$(AB)^T = B^T A^T, \quad \left(A^{-1}\right)^T = \left(A^T\right)^{-1} = A^{-T}$$

- Pour une matrice complexe $A \in \mathbb{C}^{n\times m}$, la matrice $\overline{A}^T$ s'appelle la matrice ***adjointe*** de A et est notée A^*.
  On retrouve les propriétés de la transposée avec la matrice adjointe dans le cas complexe.

Rappel sur les matrices

- Une matrice $A \in \mathbb{R}^{n\times n}$ est dite ***symétrique*** si $A = A^T$ et ***antisymétrique*** si $A = -A^T$. Elle est dite ***orthogonale*** si $AA^T = A^TA = I$, c'est-à-dire $A^{-1} = A^T$.
- Une matrice $A \in \mathbb{C}^{n\times n}$ est dite ***hermitienne*** ou ***autoadjointe*** si $A = A^*$. Elle est dite ***normale*** si $AA^* = A^*A$ et ***unitaire*** si $AA^* = A^*A = I$, c'est-à-dire $A^{-1} = A^*$.
  Exemple :

$$A = \begin{pmatrix} 3 & i & 1-5i \\ -i & -2 & 5 \\ 1+5i & 5 & 10 \end{pmatrix}$$

Déterminant d'une matrice

On appelle ***déterminant*** d'une matrice le scalaire suivant :

$$\det A = \sum_{\pi \in P} \operatorname{sign}(\pi) a_{1\pi_1} a_{2\pi_2} \dots a_{n\pi_n},$$

où P est l'ensemble des $n!$ multi-indices obtenus par permutation de $(1, \dots, n)$. Le coefficient $\operatorname{sign}(\pi)$ vaut 1 si on fait un nombre pair de transpositions, -1 sinon.

On a les relations suivantes :

- $\det A = \det A^T$, $\det A^* = \overline{\det A}$
- $\det AB = \det A \det B$
- $\det A^{-1} = (\det A)^{-1}$
- $\det \alpha A = \alpha^n \det A$

Déterminant d'une matrice... à la main

- Soit A une matrice d'ordre n.
- On note A_{ij} la matrice obtenue en éliminant la i-ème ligne et la j-ième colonne de A. On appelle son déterminant le ***mineur*** associé au coefficient a_{ij} de A.
- Le ***cofacteur*** de A_{ij} est défini par $\Delta_{ij} = (-1)^{i+j} \det A_{ij}$.
- On a alors la relation de récurrence suivante :

$$\det A = \begin{cases} a_{11} & \text{si } n = 1 \\ \displaystyle\sum_{j=1}^{n} \Delta_{ij} a_{ij} & \text{pour } n > 1 \end{cases}$$

- Une formule pour l'inverse d'une matrice :

$$A^{-1} = \frac{1}{\det A} C^T, \quad c_{ij} = \Delta_{ij}$$

Déterminant d'une matrice... exemple

On considère la matrice de rotation en 3D suivante :

$$A = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

En développant par rapport à la dernière ligne :

$$\begin{aligned} \det A &= 0 \times (-1)^{3+1} \det \begin{pmatrix} -\sin\theta & 0 \\ \cos\theta & 0 \end{pmatrix} \\ &\quad + 0 \times (-1)^{3+2} \det \begin{pmatrix} \cos\theta & 0 \\ \sin\theta & 0 \end{pmatrix} \\ &\quad + 1 \times (-1)^{3+3} \det \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \\ &= \det \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \\ &= 1 \end{aligned}$$

Rang et noyau d'une matrice

Soit A une matrice rectangulaire de taille $m \times n$.

- On appelle ***déterminant extrait d'ordre*** q, le déterminant de n'importe quelle matrice d'ordre q obtenue à partir de A en éliminant $m - q$ lignes et $n - q$ colonnes.
- Le ***rang*** de A, noté $\mathrm{rg}(A)$, est l'ordre maximum des déterminants extraits non nuls de A.
- Le rang de A est la dimension de l'image de A :

$$\mathrm{Im}(A) = \{y \in \mathbb{R}^m : y = Ax \text{ pour } x \in \mathbb{R}^n\}.$$

- Le noyau de A est le sous-espace vectoriel défini par :

$$\mathrm{Ker}(A) = \{x \in \mathbb{R}^n : Ax = 0\}.$$

Relations sur le rang et le noyau d'une matrice

Soit A une matrice rectangulaire de taille $m \times n$:

- $\mathrm{rg}(A^*) = \mathrm{rg}(A)$,
- $\mathrm{rg}(A) + \dim(\mathrm{Ker}(A)) = n$

Si A est une matrice carrée de $\mathbb{C}^{n\times n}$, les relations suivantes sont équivalentes :

(i) A est inversible

(ii) $\det A \neq 0$

(iii) $\mathrm{Ker}(A) = 0$

(iv) $\mathrm{rg}(A) = n$

(v) Les colonnes et les lignes de A sont linéairement indépendantes.

Cas particulier : les matrices diagonales et diagonales par blocs

Les matrices diagonales sont des matrices dont tous les éléments sont nuls, sauf, peut-être, ceux sur la diagonale.

- Le déterminant des matrices diagonales est le produit des éléments diagonaux.
- L'inverse d'une matrice diagonale est diagonale dont les coefficients sont les inverses de ceux de la matrice initiale.

Les matrices diagonales par blocs sont des matrices de la forme $D = \mathrm{diag}(D_1, D_2, \ldots, D_k)$ où les D_i sont des matrices carrées, éventuellement de taille différentes.

- Le déterminant d'une matrice diagonale par blocs est le produit des déterminants des blocs diagonaux.
- Exemple : la matrice de rotation en 3D précédente !

Cas particulier : les matrices triangulaires

Les matrices triangulaires sont des matrices carrées de la forme :

$$L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}, \text{ ou } U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}$$

- Le déterminant d'une matrice triangulaire est le produit de ses éléments diagonaux.
- Il est facile de résoudre un système linéaire avec une matrice triangulaire :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Valeurs et vecteurs propres

Soit A une matrice dans $K^{n\times n}$ ($K = \mathbb{R}$ ou $\mathbb{C}$).

- On dit que $\lambda \in K$ est ***valeur propre*** de A s'il existe $x \in K^n$ non nul tel que $Ax = \lambda x$.
- Le vecteur x s'appelle ***vecteur propre*** de A.
- L'ensemble des valeurs propres de A s'appelle le ***spectre*** de A et se note $\sigma(A)$.
- On appelle ***polynôme caractéristique*** de A le polynôme $\det(A - XI_n)$.
- Les valeurs propres de A sont les racines du polynôme caractéristique.
- On peut montrer que :

$$\det A = \prod_{i=1}^{n} \lambda_i, \quad \operatorname{tr}(A) = \sum_{i=1}^{n} \lambda_i$$

- On appelle ***rayon spectral*** de A le plus grand des modules des valeurs propres : $\rho(A) = \max_{\lambda\in\sigma(A)} |\lambda|$

Librairies et softwares

- **Trilinos** (C++), **PETSc** (C) : très grosses librairies orientées sur la résolution d'EDP. Structures de matrices et vecteurs en parallèle (MPI), denses et creuses.
- **Armadillo** (C++) : librairie d'algèbre linéaire, structures de matrices et vecteurs, denses et creuses.
- **Eigen** (C++, header only) : librairie d'algèbre linéaire, structures de matrices et vecteurs, denses et creuses.
- **Elemental** (C++) : librairie d'algèbre linéaire, structures de matrices et vecteurs, parallèle (MPI), denses et creuses.
- **Python** (numpy/scipy), **Julia**, **Matlab**, **Scilab**, **R**, **Sage** : ils ont tous leurs structures de matrices et vecteurs.
- sur GPU : **ViennaCL** (C++, supporte OpenCL et CUDA), **cuSPARSE** (Nvidia).

Sommaire

Un exemple

On considère le système linéaire suivant :

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \quad \text{de solution} \quad \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Un exemple

On considère le système linéaire suivant :

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \quad \text{de solution} \quad \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Que devient la solution si on perturbe un peu le second membre ?

Un exemple

On considère le système linéaire suivant :

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \quad \text{de solution} \quad \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Que devient la solution si on perturbe un peu le second membre ?

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} 32 + 0.1 \\ 23 - 0.1 \\ 33 + 0.1 \\ 31 - 0.1 \end{pmatrix}, \quad \text{de solution} \quad \begin{pmatrix} 9.2 \\ -12.6 \\ 4.5 \\ -1.1 \end{pmatrix}$$

Un exemple

On considère le système linéaire suivant :

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \quad \text{de solution} \quad \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Et si on perturbe la matrice ?

Un exemple

On considère le système linéaire suivant :

$$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}, \quad \text{de solution} \quad \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Et si on perturbe la matrice ?

$$\left[\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0.1 & 0.2 \\ 0.08 & 0.04 & 0 & 0 \\ 0 & -0.02 & -0.11 & 0 \\ -0.01 & -0.01 & 0 & 0.02 \end{pmatrix} \right] \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}$$

de solution

$$\begin{pmatrix} -81 \\ 137 \\ -34 \\ 22 \end{pmatrix}$$

Normes vectorielles

Soit $v \in K^n$ où $K = \mathbb{R}$ ou $\mathbb{C}$. Les normes les plus utilisées sont les suivantes :

$$\|v\|_1 = \sum_{i=1}^{n} |v_i|$$

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} |v_i|^2}$$

$$\|v\|_\infty = \max_i |v_i|$$

Plus généralement, sur K^n, pour $p \geq 1$, l'application

$$\|v\|_p = \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p},$$

est une norme.

Normes matricielles

Étant donnée une norme vectorielle $\|.\|$ sur $\mathbb{C}^n$, l'application $\|.\|$ définie par :

$$\|A\| = \sup_{v \in \mathbb{C}^n, v \neq 0} \frac{\|Av\|}{\|v\|} = \sup_{v \in \mathbb{C}^n, \|v\| \leq 1} \|Av\| = \sup_{v \in \mathbb{C}^n, \|v\| = 1} \|Av\|$$

est une norme matricielle appelée ***norme matricielle subordonnée*** (à la norme vectorielle associée).
Il résulte de la définition que $\|Av\| \leq \|A\|\|v\|$ pour tout v dans $\mathbb{C}^n$.

Les normes matricielles subordonnées classiques

Soit $A = (a_{ij})$ une matrice carrée d'ordre n.

$$\|A\|_1 = \sup \frac{\|Av\|_1}{\|v\|_1} = \max_j \sum_i |a_{ij}|$$

$$\|A\|_2 = \sup \frac{\|Av\|_2}{\|v\|_2} = \sqrt{\rho(A^*A)} = \sqrt{\rho(AA^*)} = \|A^*\|_2$$

$$\|A\|_\infty = \sup \frac{\|Av\|_\infty}{\|v\|_\infty} = \max_i \sum_j |a_{ij}|$$

De plus, si A est normale, alors $\|A\|_2 = \rho(A)$. Par conséquent, si A est hermitienne, ou symétrique, on a aussi $\|A\|_2 = \rho(A)$.

Norme de Frobenius

La norme $\|.\|_E$ définie par :

$$\|A\|_E = \sqrt{\sum_{i,j} |a_{ij}|^2} = \sqrt{\mathrm{tr}(A^* A)}$$

est une norme matricielle non subordonnée et qui vérifie

$$\|A\|_2 \leq \|A\|_E \leq \sqrt{n}\|A\|_2$$

C'est la norme euclidienne. Elle a l'avantage, tout comme les normes $\|.\|_1$ et $\|.\|_\infty$, d'être simple à calculer.

Retour aux perturbations

Soit A une matrice inversible, on veut comparer les solutions exactes des deux systèmes linéaire suivant :

$$Au = b$$
$$A(u + \delta u) = b + \delta b$$

On a $\delta u = A^{-1}\delta b$ et $b = Au$. D'où,

$$\|\delta u\| \leq \|A^{-1}\|\|\delta b\|, \quad \|b\| \leq \|A\|\|u\|.$$

Ainsi, on en déduit que :

$$\frac{\|\delta u\|}{\|u\|} \leq \|A\|\|A^{-1}\|\frac{\|\delta b\|}{\|b\|}$$

Retour aux perturbations

On peut faire de même avec les deux systèmes :

$$Au = b$$
$$(A + \delta A)(u + \delta u) = b$$

et on trouve l'inégalité suivante :

$$\frac{\|\delta u\|}{\|u + \delta u\|} \leq \|A\|\|A^{-1}\| \frac{\|\delta A\|}{\|A\|}$$

Le conditionnement

Soit $\|.\|$ une norme matricielle subordonnée et A une matrice inversible. Le nombre

$$\mathrm{cond}(A) = \|A\|\|A^{-1}\|$$

s'appelle le ***conditionnement*** de la matrice A, relativement à la norme considérée.

Propriétés du conditionnement

(i) Pour toute matrice A, on a

$$\begin{aligned} \mathrm{cond}(A) &\geq 1, \\ \mathrm{cond}(A) &= \mathrm{cond}(A^{-1}), \\ \mathrm{cond}(\alpha A) &= \mathrm{cond}(A) \end{aligned}$$

(ii) Si A est normale, on a

$$\mathrm{cond}_2(A) = \frac{\max_i |\lambda_i(A)|}{\min_i |\lambda_i(A)|}$$

(iii) Si A est unitaire ou orthogonale, $\mathrm{cond}_2(A) = 1$

Retour sur l'exemple

La matrice de notre exemple,

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}$$

a pour valeur propre :

$$\lambda_1 = 0.01015, \lambda_2 = 0.8431, \lambda_3 = 3.858, \lambda_4 = 30.2887$$

de sorte que $\mathrm{cond}_2(A) = \frac{\lambda_4}{\lambda_1} = 2984$
Par ailleurs, on a :

$$\frac{\|\delta u\|_2}{\|u\|_2} = 8.1985 \leq 9.9424 = \mathrm{cond}_2(A)\frac{\|\delta b\|_2}{\|b\|_2}$$

Préconditionneurs

On considère le système linéaire $Ax = b$, avec A inversible. Soit P une matrice inversible, la solution du système suivant :

$$P^{-1}Ax = P^{-1}b,$$

est la même que celle du premier système. La matrice P s'appelle un ***préconditionneur*** à gauche. L'idée est de choisir P de telle sorte que $\mathrm{cond}(P^{-1}A) << \mathrm{cond}(A)$.
Remarques :

- Le second système est en général résolu avec une méthode itérative, qui ne nécessite pas le calcul de P^{-1}.
- $P = I$ est le préconditionneur le plus simple, mais n'a aucun effet.
- $P = A$ est optimal, mais revient à résoudre le système non préconditionné.

Propriétés d'un bon préconditionneur

Un bon préconditionneur doit :

- être facile à inverser (la résolution d'un système linéaire $Pz = r$ est peu couteuse).
- regrouper les valeurs propres de $P^{-1}A$ dans une petite partie du plan complexe.
- pour une méthode itérative, de nécessiter un nombre d'itération pour atteindre une précision donnée indépendant de la taille de la matrice A.

Trouver un bon préconditionneur est un problème en soit ! Il n'existe pas de méthodes pour en trouver.

Sommaire

Système linéaire avec matrice triangulaire

On considère le système $Ux = b$ avec U matrice triangulaire supérieure :

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Alors, on peut ***remonter*** la matrice et la solution est :

$$x_i = \frac{b_i - \sum_{j>i} u_{ij} x_j}{u_{ii}}, \quad i = n, \dots, 1$$

De façon similaire, on a la solution dans le cas d'une matrice triangulaire inférieure.
La complexité d'une telle résolution est de l'ordre de n^2 opérations.

Méthode d'élimination de Gauss

On considère le système linéaire $Ax = b$, avec A inversible. L'objectif est de se ramener à un système de la forme $Ux = \tilde{b}$, avec U triangulaire supérieure.
On pose $A^1 = A$, $b^1 = b$ et on suppose que $a_{11} \neq 0$. On transforme A^1 en A^2 de la façon suivante :

$$a_{ij}^2 = a_{ij}^1 - \frac{a_{i1}^1}{a_{11}^1} a_{1j}^1, \quad i, j = 2, \ldots, n$$

$$b_i^2 = b_i^1 - \frac{a_{i1}^1}{a_{11}^1} b_1^1, \quad j = 2, \ldots, n$$

On obtient un système de la forme :

$$\begin{pmatrix} a_{11}^1 & a_{12}^1 & \ldots & a_{1n}^1 \\ 0 & a_{22}^2 & \ldots & a_{2n}^2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^2 & \ldots & a_{nn}^2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^1 \\ b_2^2 \\ \vdots \\ b_n^2 \end{pmatrix}$$

Méthode d'élimination de Gauss

En supposant $a_{kk}^k \neq 0$ pour $k = 2, \ldots, n-1$, on recommence ce procédé pour aboutir, lorsque $k = n$ au système linéaire suivant :

$$\begin{pmatrix} a_{11}^1 & a_{12}^1 & \ldots & a_{1n}^1 \\ 0 & a_{22}^2 & \ldots & a_{2n}^2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{nn}^n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^1 \\ b_2^2 \\ \vdots \\ b_n^n \end{pmatrix}$$

Il est alors facile de résoudre ce problème avec une remontée.
Le procédé d'élimination de Gauss a une complexité de l'ordre de $2n^3/3$.

Factorisation LU

La factorisation LU consiste à écrire A sous la forme d'un produit d'une matrice triangulaire inférieure L par une matrice triangulaire supérieure U.

- Les opérations de la méthode de Gauss s'écrivent comme le produit de matrices triangulaires inférieures bien choisies par A.
- On a donc, avec la méthode de Gauss, une matrice M, triangulaire inférieure, telle que $MA = U$. Soit encore $A = M^{-1}U$ avec M^{-1} triangulaire inférieure et U triangulaire supérieure.
- La résolution de $LUx = b$ s'effectue de la façon suivante :
  1. On résout $Ly = b$ par une descente,
  2. On résout $Ux = y$ par une remontée.
- Le coup de la factorisation est le même que celui de la méthode de Gauss.
- Une fois calculée, on peut réutiliser la factorisation avec un second membre différent.

Existence et unicité d'une factorisation LU

Soit A matrice d'ordre n. La décomposition $A = LU$ avec

(i) L triangulaire inférieure dont les coefficients diagonaux sont 1,

(ii) U triangulaire supérieure,

existe et est ***unique*** si et seulement si les sous-matrices principales A_i de A d'ordre $i = 1, \ldots, n-1$ sont inversibles.

Les conditions suivantes sont plus restrictives mais plus simples à vérifier :
La factorisation LU existe et est unique si :

- A est à diagonale dominante par ligne.
- ou A est à diagonale dominante par colonne, et dans ce cas on a $|l_{ij}| \leq 1$.

Changement de pivot

Pour améliorer la stabilité de la factorisation LU, on peut changer de pivot si :

- le coefficient diagonal a_{kk}^k est nul,
- le coefficient diagonal a_{kk}^k est petit, ce qui entraîne des erreurs numériques.

On peut rechercher le nouveau pivot :

- dans la colonne que l'on annule, c'est la méthode du ***pivot partiel***. On choisit l'élément le plus grand de la colonne et on permutte les lignes.
  On obtient alors la décomposition $PA = LU$ où P est une matrice de permutation. Pour résoudre $Ax = b$, on résout $PAx = LUx = Pb$
- dans la sous-matrice qu'il reste à traiter, c'est la méthode du ***pivot total***. On a alors $PAQ = LU$ où Q est aussi une matrice de permutation.

Librairies et softwares pour la méthode LU

- **LAPACK** (Fortran) : LU avec pivot partiel.
- **MUMPS** (Fortran) : Solver LU parallèle (OpenMP + MPI). Interface Fortran et C.
- **SuperLU** (C) : Solver LU parallèle (OpenMP, MPI, CUDA). Interface Fortran et C. LU avec pivot partiel.
- **PaStiX**, **SuiteSparse** : d'autres solver LU.
- **Trilinos**, **PETSc** : contiennent des interfaces avec des solvers LU externes (aller voir la liste sur leurs sites web).
- **Armadillo**, **Eigen**, **Elemental** : contiennent une ou des méthodes LU (pivot partiel, pivot total. Allez voir les documentations !).
- **Python** (scipy.linalg.lu), **Julia** (factorize, lufact), **Matlab** (lu), **Scilab** (lu), **R** (LU, avec pivot partiel), **Sage** (LU).
- sur GPU, **ViennaCL** (sans pivot), **cuSOLVER** (pivot partiel).

Calcul d'une factorisation LU

On considère le problème suivant :

$$\begin{aligned} -\Delta u &= 1 \quad \text{sur } \Omega = [0,1]^2, \\ u &= 0 \quad \text{sur } \partial\Omega \end{aligned}$$

On regarde les temps de résolution de ce problème par différence finie avec une méthode LU. On obtient ainsi une matrice creuse.
On utilise Julia 1.1 pour résoudre le système linéaire obtenu. On considère des grilles cartésiennes de taille $N \times N$ où N est une puissance de 2. La matrice est donc de taille $N^2 \times N^2$.

N	Factorisation LU (s)	Résolution LU (s)	Mémoire utilisée (Mo)
32	0.0023	0.000114	1.42
64	0.0089	0.000544	5.75
128	0.0415	0.0032	22.9
256	0.2297	0.0287	103
512	1.298	0.123	465
1024	6.767	0.560	2000

Cas d'une matrice symétrique : factorisation LDL^T

Soit A une matrice inversible dont tous les mineurs principaux sont non nuls. Il existe une unique factorisation de A sous la forme $A = LDM^T$ avec D diagonale, L triangulaire inférieure, M^T triangulaire supérieure, telles que L et M n'aient que des 1 sur leur diagonale.

Quelques remarques :

- Trouver la factorisation a le même coût que celui de la décomposition LU.
- La résolution de $Ax = b$ s'effectue en trois étapes :
  1. résoudre $Ly = b$,
  2. résoudre le système diagonal $Dz = y$,
  3. résoudre $M^T x = z$.
- Si de plus A est symétrique, alors $M = L$ et on a $A = LDL^T$.
- Si A est symétrique, le coût de la factorisation est deux fois moindre qu'une factorisation LU.

Librairies et softwares

- **LAPACK** : oui.
- **MUMPS** (Fortran) : Effectue une factorisation LDL^T si on indique que la matrice est symétrique.
- **PaStiX**, **SuiteSparse** : oui.
- **Trilinos** : à voir, pas clair.
- **PETSc** : oui, à travers MUMPS par exemple.
- **Eigen** : oui, voir `http://eigen.tuxfamily.org/dox/group__TopicLinearAlgebraDecompositions.html`
- **Elemental** : oui, voir `http://libelemental.org/documentation/0.85/lapack_like/factor.html`
- **Julia** (factorize, ldltfact), **Matlab** (ldl).

Benchmark

- Temps de factorisation :

N	Factorisation LU (s)	Factorisation LDL^T (s)
32	0.0023	0.000857
64	0.0089	0.00413
128	0.0415	0.019
256	0.2297	0.095
512	1.298	0.548
1024	6.767	2.691

- Temps de résolution

N	Résolution LU (s)	Résolution LDL^T (s)
32	0.000114	0.000024
64	0.000544	0.000124
128	0.0032	0.00108
256	0.0287	0.0074
512	0.123	0.0332
1024	0.560	0.145

Benchmark

Comparaison des allocations mémoires.

N	Mémoire LU (Mo)	Mémoire LDL^T (Mo)
32	1.42	0.628
64	5.75	2.76
128	22.9	14.31
256	103	61.16
512	465	262
1024	2000	1090

Cas symétrique définie positive : factorisation de Cholesky

Soit A une matrice symétrique définie positive (symétrique, telle que $(Ax, x) > 0$). Alors il existe une unique matrice H triangulaire supérieure dont les termes diagonaux sont strictement positifs et telle que :

$$A = H^T H$$

Cette factorisation s'appelle la factorisation de ***Cholesky***

Quelques remarques :

- Les termes de H peuvent être calculés simplement (voir le livre de Quarteroni et al par exemple).
- Le coût est deux fois moindre que pour une décomposition LU.
- Il est possible de stocker la matrice H et la matrice A dans la même matrice (elles sont toutes les deux symétriques).

Librairies et softwares

- **LAPACK** : oui.
- **MUMPS** (Fortran) : oui.
- **PaStiX**, **SuiteSparse** : oui.
- **Trilinos**, **PETSc** : oui via les librairies externes.
- **Armadillo**, **Eigen**, **Elemental** : oui pour les trois.
- **Python** (numpy/scipy.linalg.cholesky), **Julia** (factorize, chol, cholfact), **Matlab** (chol), **Scilab** (chol), **R** (chol), **Sage** (via numpy).
- sur GPU, **cuSOLVER**.

Benchmark

N	Factorisation (s)			Résolution (s)		
	LU	LDL^T	$Chol$	LU	LDL^T	$Chol$
32	0.0023	0.000857	0.000857	0.000114	0.000024	0.000026
64	0.0089	0.00413	0.00413	0.000544	0.000124	0.000122
128	0.0415	0.019	0.020	0.0032	0.00108	0.00156
256	0.2297	0.095	0.096	0.0287	0.0074	0.0089
512	1.298	0.548	0.554	0.123	0.0332	0.0397
1024	6.767	2.691	2.714	0.560	0.145	0.170

Benchmark

N	Mémoire (Mo)		
	LU	LDL^T	$Chol$
32	1.42	0.628	0.628
64	5.75	2.76	2.76
128	22.9	14.31	14.31
256	103	61.16	61.16
512	465	262	262
1024	2000	1090	1090

Factorisation QR

Étant donnée une matrice A d'ordre n, il existe une matrice unitaire Q et une matrice triangulaire supérieure R telles que $A = QR$. De plus, on peut s'arranger pour que tous les éléments diagonaux de R soient positifs ou nuls.
Si la matrice A est inversible, la factorisation QR est alors unique.

Remarques :

- La méthode est numériquement stable, contrairement à LU sans pivot.
- La décomposition nécessite environ deux fois plus d'opérations qu'une méthode LU sans pivot.
- Sa construction, non décrite ici, est basée sur les matrices de Householder (voir le livre de Ciarlet).
- Il n'y a pas d'hypothèses sur A.
- Si A est réelle, alors Q est rélle (donc orthogonale) et R aussi.
- Pour résoudre le système $Ax = b$, il suffit de résoudre $Rx = Q^* b$.

Librairies et sofwares

- **LAPACK** : oui.
- **SuiteSparse** : oui.
- **Trilinos** : oui.
- **PETSc** : oui via l'interface avec Matlab...
- **Armadillo**, **Eigen**, **Elemental** : oui !
- **Python** (numpy/scipy.linalg.qr), **Julia** (factorize, qr, qrfact), **Matlab** (qr), **Scilab** (qr), **R** (QR), **Sage** (QR).
- sur GPU, **cuSOLVER**.

Benchmark

N	Factorisation (s)			
	LU	LDL^T	$Chol$	QR
32	0.0023	0.000857	0.000857	0.0033
64	0.0089	0.00413	0.00413	0.0189
128	0.0415	0.019	0.020	0.156
256	0.2297	0.095	0.096	0.668
512	1.298	0.548	0.554	3.352
1024	6.767	2.691	2.714	X

N	Résolution (s)			
	LU	LDL^T	$Chol$	QR
32	0.000114	0.000024	0.000026	0.000311
64	0.000544	0.000124	0.000122	0.00202
128	0.0032	0.00108	0.00156	0.0126
256	0.0287	0.0074	0.0089	0.0682
512	0.123	0.0332	0.0397	0.331
1024	0.560	0.145	0.170	X

Benchmark

N	Mémoire (Mo)			
	LU	LDL^T	$Chol$	QR
32	1.42	0.628	0.628	4.35
64	5.75	2.76	2.76	22.83
128	22.9	14.31	14.31	115
256	103	61.16	61.16	554
512	465	262	262	2580
1024	2000	1090	1090	10000

Sommaire

Méthodes itérative par recherche de point fixe

Soit A une matrice inversible, on veut résoudre le système $Ax = b$.

On suppose que l'on a trouvé une matrice B et un vecteur c tels que la matrice $I - B$ soit inversible et tels que la solution unique du système $(I - B)u = c$ est la même que celle du système $Ax = b$.

La forme $(I - B)u = c$, que l'on réécrit $u = Bu + c$, suggère de trouver un point fixe à la fonction $f(u) = Bu + c$.
On se donne donc un vecteur initial arbitraire u_0 et on définit la suite de vecteur $(u_k)_{k\geq 0}$ par

$$u_{k+1} = Bu_k + c$$

On dit que la méthode est ***convergente*** si $\lim_{k\to\infty} u_k = u$ pour tout u_0.

Critère de convergence

Pour une matrice B telle que $(I - B)$ soit inversible, les propositions suivantes sont équivalentes :

(i) La méthode itérative est convergente,

(ii) $\rho(B) < 1$,

(iii) $\|B\| \leq 1$ pour au moins une norme matricielle $\|.\|$.

Exemple de matrices B

Soit A inversible, on veut résoudre le système $Ax = b$.
On suppose que l'on peut écrire A sous la forme $A = M - N$ avec M inversible. On a donc :

$$Ax = b \iff Mx = Nx + b \iff x = \underbrace{M^{-1}N}_{B} x + \underbrace{M^{-1}b}_{c}$$

On a, au passage,

$$I - B = I - M^{-1}N = I - (I - M^{-1}A) = M^{-1}A,$$

et donc $I - B$ est bien inversible.
La méthode itérative est donc :

$$u_{k+1} = M^{-1}Nu_k + M^{-1}b$$

ce qui revient, en pratique, à résoudre les systèmes linéaires :

$$Mu_{k+1} = Nu_k + b$$

Quelques remarques

- On voit bien que la matrice M doit être facile à inverser.
- On peut réutiliser une factorisation de M à chaque itération (factorisation LU par exemple).
- Le cas limite $M = A$ (et donc $N = 0$) donne une méthode qui converge en une itération. Mais cela revient à inverser A avec une méthode directe, donc pas très utile.
- On résout donc en réalité le système $(I - B)x = c = M^{-1}b$. On a vu que $I - B = M^{-1}A$, on résout donc le système

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

La matrice M est donc un préconditionneur de A ! On veut les mêmes propriété pour M que celles pour un préconditionneur.

La méthode de Jacobi

On écrit $A = D - E - F$, avec :

- D la diagonale de A,
- $-E$ la partie triangulaire inférieure de A
- $-F$ la partie triangulaire supérieure de A

On prend alors $M = D$ et $N = E + F$ et on doit résoudre à chaque itération :

$$Du_{k+1} = (E + F)u_k + b$$

Comme M doit être inversible, il faut que les coefficient diagonaux de A soient tous non nuls !

La méthode de Gauss-Seidel

On prend ici :

$$M = D - E$$
$$N = F$$

de sorte que les itérations s'écrivent :

$$(D - E)u_{k+1} = Fu_k + b$$

La matrice $D - E$ est triangulaire inférieure, et donc facile à inverser.

La méthode de relaxation (SOR)

C'est une variante de Gauss-Seidel. On considère un paramètre $\omega \neq 0$ et on prend

$$M = \frac{D}{\omega} - E$$
$$N = \frac{1-\omega}{\omega} D + F$$

de sorte que les itérations s'écrivent :

$$\left(\frac{D}{\omega} - E\right) u_{k+1} = \left(\frac{1-\omega}{\omega} D + F\right) u_k + b$$

On retrouve Gauss-Seidel pour $\omega = 1$.

Résultats de convergence

- On a vu que si $\rho(M^{-1}N) < 1$, alors la méthode converge.
- Si A et $2D - A$ sont symétriques définies positives, alors la méthode de Jacobi converge.
- Si A est symétrique définie positive, alors la méthode de Gauss-Seidel est convergente de manière monotone (l'erreur décroit à chaque itération) pour la norme $\|.\|_A$ (si A est symétrique définie positive, on définie $\|x\|_A = (Ax, x)$).
- Si A est une matrice à diagonale dominante stricte, alors les méthodes de Jacobi et Gauss-Seidel convergent.
- Si $\omega \leq 0$ ou $\omega \geq 2$, la méthode SOR diverge.
- Si A est symétrique définie positive, la méthode SOR converge si et seulement si $0 < \omega < 2$. Sa convergence est monotone pour $\|.\|_A$.

Les méthodes de Richardson

On écrit les itérations sous la forme :

$$Mx_{k+1} = Nx_k + b \iff x_{k+1} = x_k + M^{-1} r_k,$$

où $r_k = b - Ax_k$.
On peut, comme pour la méthode SOR, relaxer en introduisant un paramètre α :

$$x_{k+1} = x_k + \alpha M^{-1} r_k,$$

c'est la méthode de Richardson ***stationnaire***.
Si le paramètre dépend de k :

$$x_{k+1} = x_k + \alpha_k M^{-1} r_k,$$

c'est la méthode de Richardson ***instationnaire***.

Algorithme

On se donne un x_0 arbitraire et on pose $r_0 = b - Ax_0$. L'étape k s'écrit :

1. résoudre le système linéaire $Mz_k = r_k$
2. calculer le paramètre α_k
3. mettre à jour la solution $x_{k+1} = x_k + \alpha_k z_k$
4. mettre à jour le résidu $r_{k+1} = r_k - \alpha_k A z_k$

Comment choisir α ?

Voici un résultat, non utilisable en pratique, garantissant la convergence de la méthode de Richardson :

Si les valeurs propres de $M^{-1}A$ sont strictement positives et telles que $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$, alors la méthode de Richardson stationnaire est convergente si et seulement si $0 < \alpha < 2/\lambda_1$.
De plus,

$$\alpha_{\text{opt}} = \frac{2}{\lambda_1 + \lambda_n}$$

est le coefficient optimal.

Gradient à pas constant et optimal

On considère ici que la matrice A est symétrique définie positive.
Résoudre le système linéaire $Ax = b$ revient dans ce cas à minimiser la fonctionnelle

$$\phi(y) = \frac{1}{2} y^T A y - y^T b$$

Partant d'un point x_0 arbitraire, on ne connait pas la direction $x - x_0$
On peut donc, de façon itérative, choisir une direction de descente p_k et une distance α_k de laquelle se déplacer à partir d'une position x_k.

- Le nouveau point sera donc de la forme $x_{k+1} = x_k + \alpha_k p_k$.
- La direction peut être la direction de plus grande pente de ϕ au point x_k, qui est $-\nabla\phi(x_k) = b - Ax_k = r_k$.
  - Prendre α_k constant constitue la méthode de ***gradient à pas simple***
  - On peut minimiser la fonctionnelle le long des x_{k+1} possible. On obtient la valeur

    $$\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k}$$

    C'est la méthode de ***gradient à pas optimal***.

Algorithmes

Les deux algorithmes sont donc,

Etant donné x_0, poser $r_0 = b - Ax_0$ et, pour $k = 0, \ldots,$

$$\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k},$$
$$x_{k+1} = x_k + \alpha_k r_k,$$
$$r_{k+1} = r_k - \alpha_k A r_k$$

Dans le cas du gradient à pas simple, il n'y a pas besoin de calculer α_k puisqu'il est constant.
On peut voir ces méthodes comme des méthodes de Richardson *non préconditionnée* (i.e. $M = I$).

Résultat de convergence

Soit A une matrice symétrique définie positive, alors la méthode de gradient à pas optimal est convergente pour n'importe quelle donnée initiale x_0 et on a

$$\|x_{k+1} - x\|_A \leq \frac{\mathrm{cond}_2(A) - 1}{\mathrm{cond}_2(A) + 1} \|x_k - x\|_A$$

par conséquent, si le conditionnement de A est grand, la méthode est lente à converger.

Méthode de gradient préconditionnée

Les méthodes de gradient sont des méthodes de Richardson.
On peut refaire la même analyse en prenant une matrice M inversible, symétrique définie positive. On obtient des méthodes de ***gradient préconditionnée***. Le pas optimal dans ce cas est :

$$\alpha_k = \frac{z_k^T r_k}{z_k^T A z_k}$$

où z_k est celui de l'algorithme de Richardson ($Mz_k = r_k$).
Dans ces conditions, on a l'estimation :

$$\|x_{k+1} - x\|_A \leq \frac{\mathrm{cond}_2(M^{-1}A) - 1}{\mathrm{cond}_2(M^{-1}A) + 1} \|x_k - x\|_A$$

Méthode du gradient conjugué

Au lieu de prendre la direction de plus grande pente dans la méthode du gradient, on peut choisir une direction ***conjuguée*** aux précédentes.
Ce sont des directions A-orthogonales les unes aux autres ($p_i^T A p_j = 0, i \neq j$).
On obtient alors l'algorithme du gradient conjugué : soit x_0, on pose $r_0 = b - Ax_0$ et $p_0 = r_0$.

$$\alpha_k = \frac{p_k^T r_k}{p_k^T A p_k},$$
$$x_{k+1} = x_k + \alpha_k p_k,$$
$$r_{k+1} = r_k - \alpha_k A p_k,$$
$$\beta_k = \frac{p_k^T A r_{k+1}}{p_k^T A p_k},$$
$$p_{k+1} = r_{k+1} - \beta_k p_k.$$

Convergence du gradient conjugué

Soit A une matrice symétrique définie positive d'ordre n. Toute méthode qui utilise des directions conjuguées pour résoudre $Ax = b$ conduit à la solution exacte en au plus n itérations.

Soit A une matrice symétrique définie positive. La méthode du gradient conjuguée converge en au plus n itérations et on a :

$$\|x_k - x\|_A \leq \frac{2c^k}{1 + c^{2k}} \|x_0 - x\|_A, \quad c = \frac{\sqrt{\mathrm{cond}_2(A)} - 1}{\sqrt{\mathrm{cond}_2(A)} + 1}$$

Généralisation

Il existe des méthodes plus générales, dans le même esprit que le gradient conjuguée. On les appelle des méthodes de Krylov. Le gradient conjugué en fait partie.
En voici quelques unes :

- ***MINRES*** : Fonctionne sur des matrices symétriques. Elle est un peu plus coûteuse qu'un gradient conjugué mais finit aussi en n itérations maximum.
- ***GMRES*** : Fonctionne sur tout type de matrice. Coûteuse en mémoire et effectue plus de calculs qu'un gradient conjugué.
- ***BiCGStab*** : Fonctionne pour des matrices non nécessairement symétriques.

Librairies et softwares

- **Trilinos**, **PETSc** : possèdent tous les deux des solveurs utilisant des méthodes de Krylov préconditionnées.
- **Python** (dans scipy), **Matlab** (cg, gmres, ...), **Scilab** (conjgrad, gmres), **R** (CG, gmres), **Sage** (avec python).
- sur GPU, **ViennaCL** (CG, GMRES, BiCGStab).

Comparaison des solveurs itératifs

On reprend le même benchmark que pour les solveurs directs et on regarde les temps de résolution pour les méthodes de Krylov avec et sans préconditionneur.
L'erreur relative demandée est de 10^{-12}.

N	Temps de résolution (s)			
	LU	CG	$CG + AMG$	$CG + ILU$
32	0.0023	0.000556	0.00815	0.000846
64	0.0089	0.004508	0.0185	0.00508
128	0.0415	0.0382	0.0732	0.0395
256	0.2297	0.324	0.305	0.299
512	1.298	2.659	1.508	2.499
1024	6.767	35.457	6.549	24.826

Comparaison des solveurs itératifs

N	Temps de résolution (s)			
	LU	*MINRES*	*MINRES + AMG*	*MINRES + ILU*
32	0.0023	0.000835	0.00472	0.000829
64	0.0089	0.00787	0.0195	0.00675
128	0.0415	0.0717	0.0777	0.0552
256	0.2297	0.6604	0.3103	0.4206
512	1.298	6.340	1.552	3.706
1024	6.767	74.352	7.093	44.339

N	Temps de résolution (s)			
	LU	*GMRES*	*GMRES + AMG*	*GMRES + ILU*
32	0.0023	0.00223	0.004509	0.00104
64	0.0089	0.0389	0.0188	0.00926
128	0.0415	0.8428	0.0737	0.1108
256	0.2297	13.58	0.295	1.839
512	1.298	??	1.463	43.32
1024	6.767	??	6.930	??

Comparaison des solveurs itératifs

N	Nombre d'itérations		
	CG	$CG + AMG$	$CG + ILU$
32	71	9	37
64	144	10	69
128	288	11	132
256	577	12	244
512	1160	13	468
1024	2523	14	995

N	Nombre d'itérations		
	$MINRES$	$MINRES + AMG$	$MINRES + ILU$
32	71	9	38
64	143	11	69
128	285	12	131
256	619	12	243
512	1311	13	454
1024	2623	15	1115

Comparaison des solveurs itératifs

N	Nombre d'itérations		
	GMRES	*GMRES* + *AMG*	*GMRES* + *ILU*
32	100	8	37
64	435	10	76
128	2156	11	199
256	7748	11	746
512	$>$ 10000	12	2936
1024	$>$ 10000	14	$>$ 10000

Sommaire

Méthode de la puissance

Soit A une matrice complexe diagonalisable d'ordre n. On suppose que les valeurs propres sont ordonnées de la façon suivante :

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$$

Étant donné un vecteur initial arbitraire q_0 de norme euclidienne 1, on considère l'algorithme itératif suivant :

$$\begin{aligned} z_k &= Aq_{k-1} \\ q_k &= \frac{z_k}{\|z_k\|_2} \\ \nu_k &= q_k^* A q_k \end{aligned}$$

Sous certaines conditions, le vecteur q_k s'aligne sur le premier vecteur propre x_1 et ν_k converge vers la première valeur propre λ_1.

Convergence de la méthode de la puissance

Soit A une matrice complexe diagonalisable d'ordre n, dont les valeurs propres sont ordonnées comme précédemment. Soit x_i les vecteurs propres de A. Ils forment une base sur laquelle on décompose $q_0 = \sum_i \alpha_i x_i$. Si $\alpha_1 \neq 0$, il existe une constante $C > 0$ telle que,

$$\|\tilde{q}_k - x_1\|_2 \leq C \left|\frac{\lambda_2}{\lambda_1}\right|^k, k \leq 1,$$

où

$$\tilde{q}_k = c_k q_k, \quad c_k = \frac{\|A^k q_0\|_2}{\alpha_1 \lambda_1^k}$$

Méthode de la puissance inverse

En appliquant la méthode de la puissance à A^{-1}, on peut trouver la valeur propre de plus petit module de A (en supposant qu'elle est simple).
Plus généralement, on peut appliquer la méthode de la puissance à $(A - \mu I)^{-1}$ pour trouver la valeur propre de A la plus proche de μ (en supposant qu'elle est aussi de multiplicité 1).
L'algorithme est le suivant :

$$\begin{aligned} (A - \mu I) z_k &= q_{k-1} \\ q_k &= \frac{z_k}{\|z_k\|_2} \\ \nu_k &= q_k^* A q_k \end{aligned}$$

On résout donc un système linéaire à chaque itération.

Le principe des méthodes QR

L'idée est, tout commme pour les sytèmes linéaires, de se ramener à une matrice dont les valeurs propres sont connues. Par exemple, une matrice triangulaire !
L'algorithme est le suivant :

Soit A une matrice réelle d'ordre n. On se donne Q_0 une matrice orthogonale et on pose $T_0 = Q_0^T A Q_0$. Pour k=1,. . . ,

1. déterminer Q_k et R_k telle que $T_{k-1} = Q_k R_k$ (factorisation QR).
2. poser $T_k = R_k Q_k$.

Remarques :

- Pour la méthode QR basique, on utilise $Q_0 = I_n$.
- Il peut ne pas y avoir convergence vers une matrice triangulaire.

Convergence de la méthode QR

Soit A une matrice réelle d'ordre n telle que

$$|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$$

Alors

$$\lim_{k\to\infty} T_k = \begin{pmatrix} \lambda_1 & t_{12} & \ldots & t_{1n} \\ 0 & \lambda_2 & t_{23} & \ldots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{pmatrix}$$

Le taux de convergence est de la forme

$$|t_{i,i-1}^k| = \mathcal{O}\left(\left|\frac{\lambda_i}{\lambda_{i-1}}\right|^k\right).$$

Si on suppose de plus A symétrique, la suite T_k tend vers une matrice diagonale.

Optimisation de la méthode QR

- La méthode QR classique demande n^3 opérations à chaque itérations !
- Il est possible de partir d'une matrice T_0 bien choisie pour réduire ce coût de calcul. Ce sont des matrices de type Hessenberg supérieures (telle que $t_{ij} = 0$ dans la partie triangulaire inférieure).
- La factorisation QR peut alors être effectuée en moins d'opérations (de l'ordre de n^2 opérations).
- Lorsque les valeurs propres sont proches les unes des autres, la convergence de la méthode peut être très lente. Pour l'améliorer, on applique le même principe que pour la méthode de la puissance inverse en effectuant des translations. C'est la méthode QR avec translation.

Cas des matrices symétriques

Si la matrice A est symétrique, la méthode de Jacobi prend en compte cette information.
On pose $A_0 = A$ et on va construire une suite de matrice A_k, orthogonalement semblables à la matrice A, tendant vers une matrice diagonale dont les coefficients sont les valeurs propres de A.

1. Pour $k = 1, \ldots,$ on se donne deux indices p et q tels que $1 \leq p < q \leq n$.
2. On pose $G_{pq} = G(p, q, \theta)$ matrice de Givens de paramètre θ (ce sont des matrices de rotation d'angle θ dans le plan des indices (p, q)).
3. On choisit θ de sorte que $A_k = G_{pq}^T A_{k-1} G_{pq}$ vérifie $a_{pq}^k = 0$.

Le calcul du θ est automatique à chaque itération.

Librairies et softwares

- **Arpack** (Fortran) : Résolution de gros problèmes aux valeurs propres avec matrices creuses.
- **Trilinos** : Anasazi permet de résoudre des problèmes aux valeurs propres.
- **SLEPc** : basé sur PETSc, librairie dédiée à la résolution de problèmes aux valeurs propres.
- **Armadillo**, **Eigen**, **Elemental** : oui, contiennent tous des solveurs pour les problèmes aux valeurs propres.
- **Python** (numpy / scipy), **Julia** (oui, eig* et d'autres), **Matlab** (eig, eigs, ...), **Scilab** (eig, ...), **R** (eig), **Sage** (oui).