Méthodes numériques

Pierre-Antoine Comby
(d'après notes du cours de S. Capdevielle)

22 décembre 2017

Table des matières

Chapitre 1

Résolution de Systèmes linéaires

1 Position du problème

1.1 Pourquoi étudier la résolution de système linéaire

Exemple : Travée de pont On cherche à connaître la déformée $v(x)$ de la poutre. *figure* La modélisation physique du problème donne les équations :

$$\begin{cases} EI\frac{\mathrm{d}^2 v}{\mathrm{d}x^2} = M(x) \quad (*) \\ v(0) = v(L) = 0 \end{cases}$$

résolution par discrétisation spatiale :

$$v(x_i + \Delta x) = v(x_i) + \Delta x \frac{\mathrm{d}^2 v}{\mathrm{d}x^2}(x_i) + \frac{\Delta x^2}{2}\frac{\mathrm{d}^2 v}{\mathrm{d}x^2}(x_i) + o(\Delta x^2)$$
$$v(x_i + \Delta x) = v(x_i) - \Delta x \frac{\mathrm{d}^2 v}{\mathrm{d}x^2}(x_i) + \frac{\Delta x^2}{2}\frac{\mathrm{d}^2 v}{\mathrm{d}x^2}(x_i) + o(\Delta x^2)$$

Alors :

$$\frac{\mathrm{d}^2 v}{\mathrm{d}x^2}(x_i) = \frac{v(x_i + \Delta x) + v(x_i + \Delta x) - 2v(x_i)}{\Delta x^2}$$

On obtient donc $(*)'$:

$$EI\frac{v_{i+1} + v_{i-1} - 2v_i}{\Delta x^2} = M_i$$

Système linéaire à résoudre

$$\begin{cases} v_{i+1} + v_{i-1} - 2v_i = M_i \frac{\Delta x^2}{EI} \\ v(0) = v(L) = 0 \end{cases}$$

Soit matriciellement :

$$\begin{bmatrix} -2 & 1 & 0 & \dots & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 & -2 & 1 \\ 0 & \dots & \dots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_i \\ \vdots \\ \vdots \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} M_1 \frac{\Delta x^2}{EI} - v_0 \\ \vdots \\ M_i \frac{\Delta x^2}{EI} \\ \vdots \\ \vdots \\ M_n \frac{\Delta x^2}{EI} - v_n \end{bmatrix}$$

De façon générale tout système linéaire après discrétisation conduit à un système d'équations linéaires à résoudre :

$$AX = B$$

On considère dans ce cours des systèmes dont la solution existe toujours , elle est de plus unique.

1.2 Comment résoudre un système de linéaire

On cherche une méthode systématique programmable

Méthode de Cramer En notant A_j la j-ième colonne de A :

Méthode

- Calculer $det(A)$
- $\forall i \in [\![1, n]\!]$:
  $x_i = \dfrac{\det(A \dots A_{i-1}|b|A_{i+1} \dots A_n)}{\det(A)}$

On effectue un grand nombre d'opération !

Temps de calcul associé :
Pour un supercalculateur $10^{17} FLOP/s$

Taille système	nb d'opération	temps de calcul
n	n !	
10	4.10^6	10^7s
100	10^{158}	10^{134}ans

Une meilleur méthode – plus rapide – est nécessaire. Deux alternatives majeurs existent :

- Méthodes directes : repose sur des algorithmes d'inversion complète de la matrice ou la solution est obtenue par un nombre fini d'opération élémentaires.
- Méthodes itératives : On tend par approximation successive vers une solution approchée.

2 Méthodes directes

2.1 Résolution de système triangulaire

2.1.1 système matriciel triangulaire inférieur

On souhaitent résoudre $LY = B$ avec :

$$L = \begin{bmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ L_{n1} & \dots & \dots & L_{nn} \end{bmatrix}$$

écriture du système

$$\begin{cases} L_{11}y_1 = b_1 \\ L_{21}y_1 + L_{22}y_2 = b_2 \\ \vdots \\ L_{n1}y_1 + \dots + L_{nn}y_n = b_n \end{cases}$$

Solution

$$\forall i \in [\![1, n]\!], y_i = \frac{1}{L_{ii}} \left(b_i + \sum_{k=1}^{i-1} L_{ik} y_k \right)$$

La résolution du système se fait en : $\underbrace{\sum 1}_{\div} + \underbrace{2 \sum i}_{+,\times} = n^2$ opérations

2.1.2 système matriciel triangulaire supérieur

La résolution est identique, mais on "remonte" le système :

$$\forall i \in [\![1, n]\!], y_{n+1-i} = \frac{1}{L_{ii}} \left(b_i + \sum_{k=1}^{i-1} L_{i,n-k} y_{n-k} \right)$$

Il est rare que le problème étudié conduisent naturellement à un système triangulaire , les calculs sont cependant peu coûteux . L'objectif est donc de ramener le problème à une résolution d'un ou plusieurs système triangulaire

2.2 Factorisation de matrice

2.2.1 Factorisation LU

On cherche $A = LU$ avec $\begin{cases} L \in T_n^- \\ T \in T_n^+ \end{cases}$, Ainsi résoudre $AX = b$ est équivalent à $\begin{cases} LY = b \\ UX = Y \end{cases}$

Définition

On appelle *mineur fondamental d'ordre k* d'une matrice , le déterminant de la sous-matrice constituée des k premières lignes et colonnes de A.

Proposition

Pour $A \in GL_n(\mathbb{K}), \exists! L, U \in T_n^+ \times T_n^-$ où L est à diagonale unité,
si et seulement si tous les mineurs fondamentaux de A sont non nuls.

Méthode de construction : Pivot de gauss

on considère la matrice $A = \begin{pmatrix} a_{11} & a_{12} & \dots \\ a_{12} & \ddots & \\ \vdots & & \ddots \end{pmatrix}$ La i-ème ligne à l'étape k devient donc :

$$L_i^{(k)} = L_i^{(k-1)} - \frac{a_{i,k-1}^{(k-1)}}{a_{kk}^{(k-1)}} L_i^{(k-1)}$$

On a donc $\underbrace{A^{(n)}}_{Tsup} = \underbrace{E^{(n)} \dots E^{(1)}}_{Tinf, \text{diag unitaire}} A^{(1)}$

Remarque: On effectue de l'ordre de $\underbrace{\sum_{k=1}^{n}}_{etape} \underbrace{\sum_{i=k+1}^{n}}_{ligne} \underbrace{\left(1 + \sum_{j=k}^{n} 2\right)}_{\div, col} = O(n^3)$.

La méthode de résolution est donc très efficace (le plus long étant la factorisation). Dans le cas d'un second membre variable on peux conserver la décomposition.

Temps de calcul associé :
Pour un laptop $10^9 FLOP/s$

Taille système	nb d'opération	temps de calcul (laptop)
n	n^3	
100	10^9	$<$ 1s
10 000	10^{12}	1 min

2.2.2 Décomposition de Cholesky

Proposition

Soit $A \in S_n^{++}$ (symétrique , définie positive) .
Il existe une unique matrice $L \in T_n^+$ tel que :

$$A = LL^T$$

Démonstration : existence par récurrence ; unicité prouvé ■

Construction de l'algorithme de calcul de L

$$\forall i \in [\![1, n]\!], j > i : \begin{cases} L_{ii} = \sqrt{A_{ii} - \sum_{k=1}} \\ L_{ji} = \frac{1}{L_i i} \left(A_{ii} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right) \end{cases}$$

2.2.3 Cas des matrices bandes

Dans le cas (fréquent) ou $A = \begin{bmatrix} \ddots & \ddots & \\ \ddots & \ddots & \ddots \\ & \ddots & \ddots \end{bmatrix}$ On a pas besoin de stocker les cotés de la matrice et on effectue moins d'opération. Il faut tout de même avoir la bande entièrement définie (même si elle contient des zéros).

2.3 Conditionnement de la matrice

2.3.1 Propagation d'une perturbation du 2nd membre

Les méthodes directes pour résoudre $AX = B$ sont exactes. Cependant la résolution numérique d'un système linéaire peux être différente (erreur d'arrondi, modification du 2nd membre/matrice)

Exemple:

2.3.2 Quantification de l'influence

On note $X + \delta X$s solution approché du système suite à $b + \delta b$

$$AX = b \Rightarrow A(X + \delta X) = b + \delta b$$

$$\begin{aligned} A\delta X &= \delta B \\ \|\delta X\| &\leq \|A^{-1}\|\|b\| \\ \|b\| &\leq \|A\|\|X\| \\ \|\delta X\|\|b\| &\leq \|A\|\|A^{-1}\|\|\delta b\|\|X\| \\ \Longrightarrow \quad \frac{\|\delta X\|}{\|X\|} &\leq \|A\|\|A^{-1}\|\frac{\|\delta b\|}{\|b\|} \end{aligned}$$

Pour des erreur numériques :

$$\begin{aligned} (A + \delta A)((X + \delta X) &= b \\ A\delta X + \delta AX + \delta X\delta A &= 0 \\ \delta X &= -A^{-1}\delta A(X + \delta X) \\ \|\delta X\| &\leq \|A\|\|A^{-1}\|\frac{\|\delta A\|}{\|A\|}\|X + \delta X\| \\ \frac{\|\delta X\|}{X + \delta X} &\leq \|A\|\|A^{-1}\|\frac{\|\delta A\|}{\|A\|} \end{aligned}$$

2.3.3 Conditionnement

On défini $Cond(A) = \|A\|\|A^{-1}\|$ qui donne une indication sur la sensibilité de la matrice aux erreurs. Il dépend cependant de la norme utilisé.

Les méthode directes sont rapide mais nécessite de stocker des (grandes) matrices en mémoires.

normes de matrice

$$\|A\|_p = \max_{X \neq 0} \frac{\|AX\|_p}{\|X\|_p}$$

on peux montrer que :

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_i j|$$

$$\|A\|_1 = \sqrt{\rho(A^T A)}$$

$$\|A\|_\infty = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_i j|$$

3 Méthode itératives

3.1 Principe

On cherche maintenant à résoudre le système AX=b par approximation successives de la solution X^*. On construit une suite de vecteur $(X^{(k)})$ telle que $(X^{(k)}) \xrightarrow[k\to+\infty]{} X^*$.

Méthode

On écrit :

$$\begin{aligned} A &= M - N \text{Avec } M \in GL_n(\mathbb{R}) \\ AX &= (M-N)X + b \\ MX &= b + NX \\ X^{(k+1)} &= M^{-1}(b + NX^{(k)} \end{aligned}$$

Convergence

On défini l'erreur par rapport à la solution exacte :

$$\begin{aligned} \rho^{(k)} &= X^* - X^{(k)} \\ \rho^{(k+1)} &= X^* - X^{(k+1)} \\ \rho^{(k+1)} &= X^* - M^{-1}(b + NX^{(k)} \\ \rho^{(k+1)} &= M^{-1}N(X^* - X^k) - \underbrace{M^{-1}NX^* - M^-1X^* + X^*}_{=0 \text{ car sol exacte}} \\ \rho^{(k+1)} &= (M^{-1}N)^k \rho^{(0)} \end{aligned}$$

On a donc convergence ssi $(M^{-1}N)^k \xrightarrow[k\to+\infty]{} 0$.

Proposition *(Condition de convergence)*

La méthode itérative décrite par $X^{(k+1)} = M^{-1}(b + NX^{(k)}$ converge pour tout $X^{(0)}$ ssi

$$\rho(M^{-1}N) = \sup_{|\lambda|} Sp(MN) < 1$$

3.2 Méthode de Jacobi

Construction de l'algorithme

On pose $M = D = diag(A)$ on a donc :

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij} x_j^{(k)} \right)$$

Convergence

Proposition

Si A est à diagonale strictement dominante i.e :

$$\forall i, \begin{cases} |a_{ii}| > \sum_{j=1}^{n} |a_{ij}| \\ ou \\ |a_{jj}| > \sum_{\substack{i=1 \\ j\neq i}}^{n} |a_{ji}| \end{cases}$$

, alors la méthode de Jacobi converge pour tout $X^{(0)}$.

Démonstration : Laissé au lecteur , utilisé le théorème de Brown et la norme infini de matrice. ■

3.3 Méthode de Gauss-Seidel

Principe et construction

On améliore la méthode de Jacobi en réutilisant les termes déjà calculés :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

La méthode de Gauss-Seidel revient à appliquer la méthode de Jacobi avec $M = D + A_L$ Avec $A = \begin{pmatrix} \ddots & & A_U \\ & D & \\ A_L & & \ddots \end{pmatrix}$

par rapport à Jacobi

- converge plus vite
- utilise moins de mémoire

Convergence

Proposition

- Si A est à diagonale dominante (strictement) par ligne alors la méthode G-S converge
- Si $A \in S_n^{++}(\mathbb{R})$ Alors la convergence est assurée

3.4 Méthode de relaxation

on améliore la vitesse de convergence de G-S grâce à l'introduction de $\omega \neq 0$ et on prend :

$$A = \underbrace{\left(\frac{D}{\omega} - A_L\right)}_{M} \underbrace{\left(\frac{1-\omega}{\omega}\right)(D - A_U)}_{N}$$

La convergence est pilotée par $\rho(M^{-1}N)$ on choisi $\begin{cases} \omega > 1 \text{ (sur relaxation)} \\ \omega < 1 \text{ (sous relaxation)} \end{cases}$ de telle sorte à accélérer la convergence de la méthode (En pratique $0 < \omega < 2$)

Algorithme

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega) x_i^{(k)}$$

3.5 Critère d'arrêt

- Le critère idéal est défini à partir de la solution exacte $\|X^* - X^{(k)}\| < \epsilon$
- On peux fixer un critère sur l'écart relatif $\frac{\|X^{(k+1)} - X^{(k)}\|}{\|X^{(k)}\|} < \epsilon$
  - Critère peu coûteux
  - ne contrôle pas l'erreur absolue de la méthode (CV ver une fausse solution possible)
- On peux comparer $\|AX^{(k)} - b\|$ à 0, et définir de même un seuil, on a cette fois si une convergence vers la bonne solution.

Chapitre 2

Approximation et interpolation

Introduction

On souhaite calculer la valeur/dérivée d'une fonction f en certains points. Dans le contexte où cette fonction est difficile à manipuler ou alors connue qu'en certains points, il faut passer par une approximation de f.

1 Méthode d'interpolation

1.1 Méthode de collocation

La fonction d'interpolation F coïncide avec la fonction f en $N+1$ points $(x_0 \dots x_N)$ où f est connue.

1.1.1 Forme polynomiale

On approxime f par F , polynômiale :

$$F(x) = \sum_{k=0}^{N} = q_k x^k$$

Or $\forall j \in [\![0, N]\!], F(x_j) \sum_{k=0}^{N} = q_k x_j^k = f(x_j) = f_j$. On a un système linéaire d'équation :

$$\begin{pmatrix} 1 & x_0 & \dots & x_0^N \\ 1 & x_1 & & x_1^N \\ & \vdots & & \vdots \\ 1 & x_N & \dots & x_N^N \end{pmatrix} \begin{pmatrix} q_0 \\ \vdots \\ \\ \vdots \\ q_N \end{pmatrix} = \begin{pmatrix} f_0 \\ \vdots \\ \\ \vdots \\ f_N \end{pmatrix}$$

On résout le système linéaire (Vandermond) et on obtient l'interpolation.

1.1.2 Interpolation de Lagrange

On approxime f par la somme de polynômes de degré N :

$$L_k = \prod_{\substack{j=0 \\ j \neq k}}^{N} \frac{x - x_j}{x_k - x_j} \text{ et } F(x) = \sum_{k=0}^{N} f_k L_k(x)$$

En effet $L_i(x_j) = \delta_{ij}$

1.1.3 Limite de l'interpolation polynomiale

- Le coût de calcule est grand pour N grand .
- Difficulté de représentation de la fonction à approximer entre les points de collocation.
- Apparition d'instabilité (diverge de Runge)

$\Rightarrow$ Adapté pour N petit, ou avec une approximation par morceau.

1.2 Interpolation d'éléments finis

$N+1$ points sont connus , on cherche un assemblage de fonction aux variation simple (linéaire, définis par morceaux ...)

$$\begin{cases} \phi_k(x_j) = \delta_{kj} \\ \phi_k(x) \text{ linéaire par morceau} \end{cases}$$

Alors

$$F(x) = \sum_{k=0}^{N} f_k \phi_k(x)$$

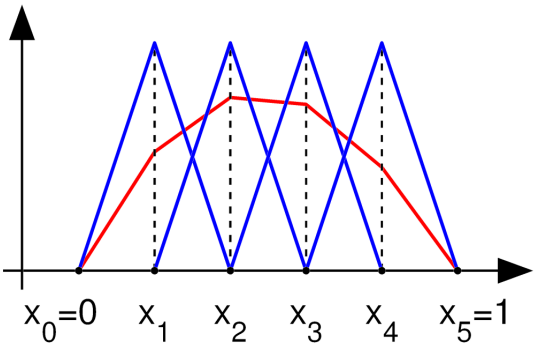

FIGURE 2.1 – Représentation des ϕ_k et de leur somme simple

Équation des ϕ_k

$$\begin{cases} \phi_k(x) = \frac{x-x_{k-1}}{x_k-x_{k-1}} \text{ sur } [x_{k-1}, x_k] \\ \phi_k(x) = -\frac{(x-x_k)}{x_{k+1}-x_k} + 1 \text{ sur } [x_k, x_{k+1}] \end{cases}$$

Remarque: Avec cette méthode l'augmentation du nombre de point est aisée.

1.3 Polynôme osculatoire

En plus de la coïncidence de $F(x_j) = f_j$ on cherche la coïncidence des m premières dérivées.

1.3.1 Cas des Polynômes d'Hermite

$$F(x) = \sum f_k U_k + \sum f'_k V_k$$

Avec

$$\begin{cases} U_k = (1 - 2L'_k(x))(x - x_k)L_k(x)^2 \\ V_k = (x - x_k)L_k(x)^2 \end{cases}$$

Qui vérifie : $\begin{cases} u_k(x_j) = \delta_{kj} \\ u'_k(x_j) = 0 \forall j \\ v_k(x_j) = 0 \forall j \\ v'_k(x_j) = \delta_{kj} \end{cases}$

2 Méthode d'approximations sans coïncidence

2.1 Approximation par une droite au sens des moindres carrés

$$f \longrightarrow P(x) = q_0 + q_1 x$$

On veux minimiser l'erreur au sens des moindres carrés :

$$S(q_0, q_1) = \sum_{i=0}^{N} (q_0 + q_1 x_i - f_i)^2$$

Ce qui revient un problème de minimisation :

$$\begin{cases} \frac{\partial S}{\partial q_0} = 2\sum_{i=0}^{N}(q_0 + q_1 x_i - f_i) = 0 \\ \frac{\partial S}{\partial q_1} = 2\sum_{i=0}^{N} x_i(q_0 + q_1 x_i - f_i) = 0 \end{cases}$$

D'où le système :

$$\begin{bmatrix} \sum_{i=0}^{N} 1 & \sum_{i=0}^{N} q_i \\ \sum_{i=0}^{N} x_i & \sum_{i=0}^{N} x_i^2 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{N} f_i \\ \sum_{i=0}^{N} f_i x_i \end{bmatrix}$$

2.2 Généralisation Polynomiale

On prend cette fois

$$P(x) = \sum_{i=0}^{n} q_i x^i$$

On minimise donc

$$S(q_0, \ldots, q_n) = \sum_{j=0}^{N} \left(\sum_{i=0}^{n} q_i x^i - f_i \right)^2$$

ie :

$$\frac{\partial S}{\partial q_k} = 2\sum_{j=0}^{N}\sum_{i=0}^{n}(q_i x_j^k - f_i)x_j^k = 0$$

Chapitre 3

Recherche de valeurs propres et vecteurs propres

1 Mise en contexte

1.1 Pourquoi s'intéresser au problème ?

1.2 Exemple flambement d'une poutre

1.3 Aperçu des méthodes et domaines d'emploi

- Déterminer les racines du polynôme caractéristiques
  - Manque d'efficacité pour de grandes valeurs de n
  - $n \simeq 5 - 10$ en pratique
- Méthode itérative
  - Puissance / Puissance inverse
  - Jacobi
  - Householder

2 Méthode de la puissance et puissance inverse

On prend A inversible réelle et diagonalisable, de valeurs propres distinctes : $|\lambda_1| < \dots |\lambda_n|$.les vecteurs propres forment une base $v_1^* \dots v_n^*$

2.1 Méthode de la puissance

Permet de trouver une approximation de (λ_n, v_n^*)

Algorithme

On initialise $V_n^{(0)}$ de norme 1 . À chaque itération :

$$\begin{cases} Z = AV_n(k) \\ V_n^{(k+1)} = \frac{Z}{\|Z\|_2} \\ |\lambda_n^{(k+1)} = \|Z\|_2 \end{cases}$$

Arrêt quand λ_n n'évolue (presque plus).
L'algorithme converge bel et bien :

Démonstration :

$$V_n^{(k)} = \frac{AV_n^{(k-1)}}{\|AV_n^{k-1}\|} = \dots = \frac{A^k V_n^{(0)}}{\|A^k V_n^{(0)}\|}$$

On note $V_n^{(0)} = \sum \alpha_i v_i^*$ dans la base des vecteurs propres. d'où :

$$\begin{aligned} A^k V_n^{(0)} &= \sum_{i=1}^{N} \alpha_i A^k v_i^* \\ &= \sum_{i=1}^{N} \alpha_i \lambda_i^k v_i^* \\ &= \alpha_n \lambda_n^k \left(v_n^* + \sum_{i=1}^{N} \frac{\alpha_i \lambda_i^k}{\alpha_n \lambda_n^k} v_i^* \right) \end{aligned}$$

Alors

$$V_n^{(k)} = \frac{A^k V_n^{(0)}}{\|A^k V_n^{(0)}\|} \xrightarrow[k \to +\infty]{} \frac{\alpha_n \lambda_n^k v_n^*}{|\alpha_n \lambda_n^k|}$$

Le vecteur $V_n^{(k)}$ s'aligne donc dans la direction de V_n. ■

Remarque: Si $\alpha_n = 0$ en théorie on récupère alors λ_{n-1} et v_{n-1} mais par approximation numérique on ressort de cette convergence.
signe de λ_n :

- si le résultat change de signe à chaque itération : $\lambda_n < 0$ (on fait le test sur le produit scalaire de deux itérations du vecteur propre)
- sinon $\lambda_n > 0$

2.2 Obtention des modes suivants

On parle de déflation orthogonale :

- On choisi un vecteur d'initialisation $V_n^{(0)} \perp v_n^*$
- On applique la méthode des puissance en ré-orthogonalisant à chaque itération :

$$V_{n-1}^{(k+1)'} = V_{n-1}^{k+1} - \frac{V_n^T V_{n-1}^{k+1} V_n}{V_n^T V_n}$$

2.3 Méthode de la puissance inverse

- Permet de déterminer les valeurs propres minimales
- Beaucoup d'application concrète.
- On applique la méthode vu en 2.2 à A^{-1} :

Méthode

On initialise $V_n^{(0)}$ de norme 1 . À chaque itération :

$$\begin{cases} \text{On résout } AZ = V_1^{(k)} \\ V_n^{(k+1)} = \frac{Z}{\|Z\|_2} \\ |\lambda_1^{(k+1)}| = \frac{1}{\|Z\|_2} \end{cases}$$

Arrêt quand λ_n n'évolue (presque plus).

2.4 Méthode de la puissance inverse avec décalage spectral

Contexte : on cherche la valeur propre la plus proche d'un nombre μ donné.
On applique la méthode de la puissance inverse à

$$B = A + \mu I_n$$

2.5 Condition sur la matrice A

- puissance
  - A dz
  - valeurs propres distinctes 2 à 2 .

  $\Rightarrow$ Les vecteurs propres forment une base de $\mathbb{R}^n$
- Déflation orthogonale
  - Si A est symétrique, les vecteurs propres sont orthogonaux on applique la méthode tel quelle.
  - SI A n'est pas symétrique, il faut passer par la base duale :
    si on note U_i les vecteurs propres de A, la base duale V_i est définie par $V_i^T U_j = \delta_{ij}$. On construit alors une "matrice de déflation" ayant les mêmes vecteurs propres et valeurs propre que A :

$$B = A - \lambda_n U_n^T V_n$$

3 Méthode de Jacobi

- Méthode itérative pour approcher simultanément *toutes* les valeurs propres de A.
- Il est nécessaire que A soit symétrique.

La méthode consiste à diagonaliser A de manière itérative par une séquence de transformation orthogonales (matrice de rotation). On construit une suite matrice $(A^{(k)})$ telle que $\lim_{k\to\infty} A^{(k)} = diag(\lambda_1, \ldots, \lambda_n)$. A chaque itération : $A^{(k+1)} = R_{pq}^T A^{(k)} R_{pq}$

Construction de R_{pq}

Idée en 2D : On part de la matrice de rotation élémentaire :

$$\begin{bmatrix} cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

D'où la transposée de la matrice de rotation élémentaire R_{pq} (matrice de GIVENS)

$$R_{pq} = \begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & \cos\theta & & & & \sin\theta & & \\ & & & 1 & & 0 & & & \\ & & & & \ddots & & & & \\ & & & 0 & & 1 & & & \\ & & -\sin\theta & & & & \sin\theta & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{bmatrix}$$

La transformation $R_{pq}^T A R_{pq}$ affecte seulement les lignes et les colonnes de A . On va se sortir de cette transformation en réglant θ pour annuler le terme A_{pq} de A :

$$B = R_{pq}^T A R_{pq}$$

$$B_{pq} = (\cos\theta[ligne\ p\ de\ A] - \sin\theta[ligne\ q\ de\ A]) \begin{pmatrix} R_q \end{pmatrix}$$

Comme A est symétrique :

$$(*)\quad B_{pq} = \cos\theta\sin\theta A_{pp} - \cos\theta\sin\theta A_{qq} + \cos^2\theta - \sin^2\theta A_{pq}$$

- si $A_{pq} = 0$ on prend $\theta = 0$

- sinon On cherche θ tq $B_pq = 0$ On divise $(*)$ par $\cos\theta\sin\theta$

$$B_{pq} = A_{pp} - A_{qq} + \left(\frac{\cos\theta}{\sin\theta} - \frac{\sin\theta}{\cos\theta}\right) A_{pq} = 0$$
$$A_{pp} - A_{qq} + (\frac{1}{\tan\theta} - \tan\theta)A_{pq} = 0$$
$$\tan^2\theta + \frac{A_{qq} - A_{pp}}{A_{pq}}\tan\theta - 1 = 0$$

On résout donc (pour t) :

$$t^2 + 2\eta t - 1 = 0$$

Alors on a :

$$\begin{cases} \cos\theta = \frac{1}{\sqrt{1+\tan^2\theta}} \\ \sin\theta = \cos\theta\tan\theta \end{cases}$$

- Choix du terme A_{pq} à annuler : On peux montrer qu'à chaque étape k le choix optimal des indices p et q est donnée par : $A_{pq}^{(k)} = \sup_{i\neq j}|A_{ij}^{(k)}|$.Mais la recherche de ce maximum est coûteuse ($O(n^2)$)

Algorihtme de Jacobi

```
1 Pour chaque ligne i  de A  (1 à n-1) :
2 On pose p=i ; q=i+1
```

```
1 tant que $\sum|A_{ij}|^2)^1/2 > \epsilon$  :
2 pour p-1 à n-1:
3 pour q=p+1 à n
4 -- calcul de $\eta, t$
5 -- calcul de \cos\theta et \sin\theta
6 -- calcul de $R_{pq}^TAR_{pq}
```

Annexe A

Code python

1 Méthodes directes

1.1 Décomposition LU

Décomposition LU

```
1
2  def decomp_LU(A): # les mineurs fondamentaux de A sont tous non nuls
3      # Initialisation
4      n=len(A)
5      U=np.zeros((n,n))
6      L=np.eye(n)
7      U[0,0]=A[0,0]
8      # Remplissage de U et L selon l'algorithme fourni
9      for j in range(1,n):
10         U[0,j]=A[0,j]
11         L[j,0]=A[j,0]/A[1,1]
12     for i in range(1,n-1):
13         U[i,i]=A[i,i]- sum([L[i,k]*U[k,i] for k in range(i)])
14         for j in range(i,n):
15             U[i,j]=A[i,j]-sum([L[i,k]*U[k,j] for k in range(i)])
16             L[j,i]=(A[j,i]-sum([L[i,k]*U[k,j] for k in range(i)]))/U[i,i]
17     U[-1,-1]=A[-1,-1]- sum([L[-1,k]*U[k,-1] for k in range(n)])
18     return L,U
```

1.2 Décomposition de Cholesky

Cholesky

```
1  def decomp_cholesky(A): # est symétrique positive
2      # initialisation
3      n=len(A)
4      L=np.zeros((n,n))
5      \# Remplissage de L
6      for i in range(n):
7          L[i,i]=np.sqrt(A[i,i]-sum([L[i,k]**2 for k in range(i)]))
8          for j in range(i,n):
9              L[j,i]=(A[j,i]-sum([L[i,k]*L[j,k] for k in range(i)]))/L[i,i]
10     return L
```

1.2.1 Application à la résolution de système linéaire

Résolution de système triangulaire inférieur

```
def resol_Tinf(B,d):
    n=len(d)
    y=np.zeros(n)
    for i in range(n):
        y[i]=(d[i]-sum([B[i,k]*d[k] for k in range(i)]))/B[i,i]
    return y
```

Résolution de système triangulaire supérieur

```
def resol_Tsup(C,e):
    n=len(e)
    y=np.zeros(n)
    for i in range(n-1,-1,-1):
        y[i]=(e[i]-sum([C[i,k]*y[k] for k in range(i,n)]))/C[i,i]
    return y
```

Résolution par la décomposition LU

```
def resol_LU(A,b):
    L,U=decomp_LU(A)
    return resol_Tsup(U,resol_Tinf(L,b))
```

Résolution par la décomposition de Cholesky

```
def resol_Cholesky(A,b):
    L=decomp_cholesky(A)
    return resol_Tsup(L.transpose(),resol_Tinf(L,b))
```

2 Méthodes indirectes

```
def norme(X):
    n=len(X)
    return sum([X[i]**2 for i in range(n)])**(1/2)
def jacobi_absolu(A,b):
    n=len(A)
    X=2*np.ones(n)
    critere_arret_residu_absolu=0.001
    residu=np.dot(A,X)-b
    while norme(residu) > critere_arret_residu_absolu:
        for i in range(n):
            X[i]=(b[i]-sum([A[i,j]*X[j] for j in range(n) if i!=j]))/A[i,i]
        residu=np.dot(A,X)-b
    print(X)
    return X
```

2.1 méthode de Jacobi avec critère relatif

```
def jacobi_relatif(A,b):
    n=len(A)
    X=np.ones(n)
    X2=np.ones(n)*2
    critere_arret_residu_relatif=0.001
    while norme(X-X2)/norme(X) > critere_arret_residu_relatif:
        X=cp.deepcopy(X2)
        for i in range(n):
            X2[i]=(b[i]-sum([A[i,j]*X2[j] for j in range(n) if i!=j]))/A[i,i]
        residu=norme(X-X2)/norme(X)
    return X
```

2.2 méthode de Gauss-Seidel

```
def gs(A,b):
    n=len(A)
    X2=2*np.ones(n)
    X=np.ones(n)
    critere_arret_residu_gs=0.1
    while norme(np.dot(A,X2)-b) > critere_arret_residu_gs:
        X=cp.deepcopy(X2)
        for i in range(n):
            X2[i]=(b[i] - sum([A[i,j]*X2[j] for j in range(n) if j<i])
                   - sum([A[i,j]*X[j] for j in range(n) if j>i]))/A[i,i]
    return X2
```

2.3 Méthode de relaxation

```
def relaxation(A,b,omega):
    assert omega!=0
    n=len(A)
    X2=2*np.ones(n)
    X=np.ones(n)
    critere_arret_residu_relaxation=0.1
    residu=np.dot(A,X2)-b
    while norme(residu) > critere_arret_residu_relaxation:
        X=cp.deepcopy(X2)
        for i in range(n):
            X2[i]=(omega*(b[i] - sum([A[i,j]*X2[j] for j in range(n) if j<i])
                   - sum([A[i,j]*X[j] for j in range(n) if j>i]))/A[i,i])
                   + (1-omega)*X[i]
    return X2
```

3 Recherche de valeurs propres et vecteurs propres

recherche des éléments propre minimaux

```
1
2  def puissance_inv(A,eps=1e-5):
3      n=len(A)
4      V=np.ones(n)/np.sqrt(n)
5      Z=np.linalg.solve(A,V)
6      V=Z/norme2(Z) #Vecteur propre
7      vpr=1/norme2(Z) #valeur propre
8      ecart=2*eps
9      while abs(ecart) > eps:
10         vpr_prec= vpr
11         V_prec=deepcopy(V)
12         Z=np.linalg.solve(A,V)
13         V=Z/norme2(Z)
14         vpr=1/norme2(Z)
15         ecart=abs(vpr)-abs(vpr_prec)
16     ps=sum([V[k]*V_prec[k] for k in range(len(V))])
17     if ps <0 :
18         return (-1)*vpr, V
19     return vpr, V
```

recherche par déflation orthogonale

```
1  def defla_orth(A,l_proj,eps=1e-5):
2      def orthog(V,l_proj):
3          ortho=0
4          for vec in l_proj:
5              ortho -= np.dot(np.dot(np.transpose(vec),V),vec)
6                       /np.dot(np.transpose(vec),vec)
7          return V-ortho
8      n=len(A)
9      V=np.ones(n)/np.sqrt(n)
10     V=orthog(V,l_proj)
11     Z=np.linalg.solve(A,V)
12     V=Z/norme2(Z)
13     vpr=1/norme2(Z)
14     ecart=2*eps
15     while abs(ecart) > eps:
16         vpr_prec= vpr
17         V_prec=deepcopy(V)
18         Z=np.linalg.solve(A,V) #plus rapide que LU
19         V=Z/norme2(Z)
20         V=orthog(V,l_proj)
21         vpr=1/norme2(Z)
22         ecart=abs(vpr)-abs(vpr_prec)
23     ps=sum([V[k]*V_prec[k] for k in range(len(V))])
24     if ps <0 :
25         return (-1)*vpr,V
26     return vpr, V
```

recherche de tous les élements propres d'une matrice en partant des plus petits

```
def puissance_inv_tot(A,N,eps=1e-5):
    """ Détermine les N premières valeures propres de A """
    n=len(A)
    assert N <= n, "nombre de valeur demandée supérieur à la taille de la matrice"
    valpr=[]
    vecpr=[]
    val,vec=puissance_inv(A)
    valpr.append(val);vecpr.append(vec)
    for i in range(N-1):
        val,vec=defla_orth(A,vecpr)
        valpr.append(val)
        vecpr.append(vec)
return valpr,vecpr
```