Analyse Numérique

Notes de Cours

Licence L3 — 2025–2026

Yaë Ulrich Gaba

« Le but du calcul n'est pas les nombres, mais la compréhension. »
— Richard Hamming

25 mars 2026

Table des matières

Liste des Algorithmes

Préface

L'analyse numérique est l'art de résoudre des problèmes mathématiques *de façon approchée mais contrôlée*. Contrairement à l'analyse théorique qui établit l'existence et l'unicité des solutions, l'analyse numérique se préoccupe de les *calculer effectivement* et de *quantifier l'erreur commise*.

Ce cours couvre les fondements : arithmétique flottante, résolution de systèmes linéaires, interpolation, intégration numérique, équations différentielles ordinaires et problèmes aux valeurs propres. Chaque méthode est accompagnée de son analyse d'erreur, de son pseudo-code et d'implémentations en Python et Julia.

Prérequis. Analyse réelle I & II, algèbre linéaire, notions de programmation.

Références.

- Quarteroni, Sacco, Saleri — *Numerical Mathematics*, Springer.
- Süli, Mayers — *An Introduction to Numerical Analysis*, Cambridge.
- Trefethen, Bau — *Numerical Linear Algebra*, SIAM.
- Higham — *Accuracy and Stability of Numerical Algorithms*, SIAM.
- Demailly — *Analyse Numérique et Équations Différentielles*, EDP Sciences.

Chapitre 1

Arithmétique Flottante, Erreurs et Stabilité

« Le calcul numérique est l'art de donner la bonne réponse à un problème légèrement différent. »

1.1 Exemple motivant

Calculons $f(x) = \frac{1-\cos x}{x^2}$ pour $x = 10^{-8}$ en double précision. Le résultat théorique est $\approx 0{,}5$, mais Python donne $0{,}0$! La **cancellation catastrophique** entre 1 et $\cos x$ élimine tous les chiffres significatifs. Comprendre l'arithmétique flottante est indispensable.

1.2 Représentation des nombres en machine

Définition 1.1 (Nombre flottant)**.** En base β avec t chiffres significatifs, un nombre flottant s'écrit :

$$x = \pm m \times \beta^e, \quad m = d_0.d_1 d_2 \cdots d_{t-1}, \quad d_0 \neq 0,$$

où $e_{\min} \leq e \leq e_{\max}$ et $0 \leq d_i < \beta$.

Définition 1.2 (Standard IEEE 754)**.**

Format	**Bits**	**Mantisse**	$\varepsilon_{\text{mach}}$
Simple précision	32	23+1 bits	$\approx 1{,}19 \times 10^{-7}$
Double précision	64	52+1 bits	$\approx 2{,}22 \times 10^{-16}$

Définition 1.3 (Epsilon machine)**.** L'**epsilon machine** $\varepsilon_{\text{mach}}$ est le plus petit $\varepsilon > 0$ tel que $\text{fl}(1+\varepsilon) > 1$:

$$\varepsilon_{\text{mach}} = \frac{1}{2}\beta^{1-t}.$$

Pour tout $x \in \mathbb{R}$ représentable, $\text{fl}(x) = x(1+\delta)$ avec $|\delta| \leq \varepsilon_{\text{mach}}$.

1.3 Erreurs d'arrondi et propagation

Définition 1.4 (Erreur absolue et relative)**.**

$$\text{Erreur absolue : } E_a = |x - \tilde{x}|, \tag{1.1}$$

$$\text{Erreur relative : } E_r = \frac{|x - \tilde{x}|}{|x|} \quad (x \neq 0). \tag{1.2}$$

Théorème 1.5 (Propagation des erreurs)**.** *Si $\tilde{x} = x(1+\varepsilon_x)$ et $\tilde{y} = y(1+\varepsilon_y)$:*

- $\tilde{x} \cdot \tilde{y} \approx xy(1 + \varepsilon_x + \varepsilon_y)$ *: erreurs relatives s'ajoutent.*
- $\tilde{x} - \tilde{y} \approx (x-y) + x\varepsilon_x - y\varepsilon_y$ *: si $x \approx y$, l'erreur relative explose.*

Attention

La **cancellation catastrophique** se produit lors de la soustraction de nombres presque égaux. Exemple : $1 - \cos x$ pour x petit. Remède : utiliser $2\sin^2(x/2)$.

1.4 Conditionnement d'un problème

Définition 1.6 (Conditionnement)**.** Le **conditionnement** d'un problème f au point x est :

$$\kappa(f, x) = \left| \frac{x f'(x)}{f(x)} \right|.$$

Un problème est **bien conditionné** si $\kappa \sim 1$, **mal conditionné** si $\kappa \gg 1$.

Définition 1.7 (Conditionnement matriciel)**.** Pour $Ax = b$:

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|.$$

L'erreur relative sur x est amplifiée par $\text{cond}(A)$:

$$\frac{\|\delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|\delta b\|}{\|b\|}.$$

1.5 Stabilité numérique

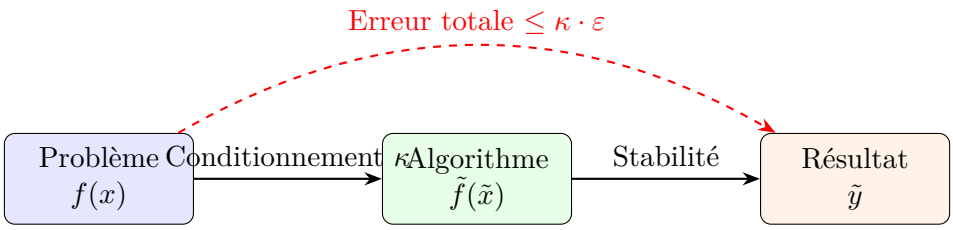
Définition 1.8 (Algorithme stable)**.** Un algorithme est **stable** s'il donne la solution exacte d'un problème **légèrement perturbé**. Il est **rétrograde stable** (backward stable) si la perturbation est de l'ordre de $\varepsilon_{\text{mach}}$.

Méthode

La précision du résultat dépend de deux facteurs :

1. Le **conditionnement** du problème (inhérent, on ne peut le changer).
2. La **stabilité** de l'algorithme (choix du programmeur).

Règle : erreur $\lesssim$ cond $\times\, \varepsilon_{\text{mach}}$.



1.6 Implémentation

Python

```python
import numpy as np

# Epsilon machine
eps = np.finfo(float).eps
print(f"eps_mach = {eps}")  # 2.220446049250313e-16

# Cancellation catastrophique
x = 1e-8
f_bad = (1 - np.cos(x)) / x**2
f_good = 2 * np.sin(x/2)**2 / x**2
print(f"Mauvais : {f_bad}")   # 0.0
print(f"Bon     : {f_good}")  # 0.49999999...

# Conditionnement d'une matrice
A = np.array([[1, 1], [1, 1.0001]])
print(f"cond(A) = {np.linalg.cond(A):.1f}")  # ~40000
```

Julia

```julia
# Epsilon machine
println("eps_mach = ", eps(Float64))

# Cancellation
x = 1e-8
f_bad = (1 - cos(x)) / x^2
f_good = 2sin(x/2)^2 / x^2
println("Mauvais : $f_bad")
println("Bon     : $f_good")

# Conditionnement
using LinearAlgebra
A = [1.0 1.0; 1.0 1.0001]
println("cond(A) = ", cond(A))
```

1.7 Exercices

Exercice 1.1 ($\star$ – Epsilon machine)**.** Écrire un programme qui calcule $\varepsilon_{\text{mach}}$ par dichotomie. Comparer avec `np.finfo(float).eps`.

Exercice 1.2 ($\star$ – Cancellation)**.** Proposer un calcul stable de $\sqrt{x+1}-\sqrt{x}$ pour x grand. Vérifier numériquement.

Exercice 1.3 ($\star\star$ – Conditionnement)**.** Calculer le conditionnement du problème $f(x) = \ln x$ au point $x = 1$. Interpréter.

Exercice 1.4 ($\star\star$ – Matrice de Hilbert)**.** Soit H_n la matrice de Hilbert ($h_{ij} = 1/(i+j-1)$). Calculer $\text{cond}(H_n)$ pour $n = 2, \ldots, 15$. Que constate-t-on ?

Exercice 1.5 ($\star\star\star$ – Projet)**.** Étudier la stabilité numérique de l'évaluation du polynôme $p(x) = \sum a_i x^i$ par l'algorithme de Horner vs. l'évaluation naïve. Comparer les erreurs relatives pour des polynômes de degré élevé.

Formules clés

$$\text{fl}(x) = x(1+\delta),\ |\delta| \le \varepsilon_{\text{mach}} \qquad \varepsilon_{\text{mach}} = \tfrac{1}{2}\beta^{1-t}$$
$$\kappa(f,x) = |xf'(x)/f(x)| \qquad \text{cond}(A) = \|A\| \cdot \|A^{-1}\|$$

Chapitre 2

Recherche de Racines

2.1 Exemple motivant

Trouver x tel que $e^x = 3x$ revient à chercher la racine de $f(x) = e^x - 3x$. Aucune formule exacte n'existe : il faut une méthode numérique.

2.2 Méthode de bissection

Théorème 2.1 (Théorème des valeurs intermédiaires)**.** *Si $f \in C([a,b])$ et $f(a)f(b) < 0$, alors $\exists\, c \in (a,b)$ tel que $f(c) = 0$.*

Algorithme 1 : Bissection

Entrée : f, a, b, tolérance ε
Sortie : Approximation de la racine c
while $b - a > \varepsilon$ **do**
    $c \leftarrow (a+b)/2$;
    if $f(a) \cdot f(c) < 0$ **then**
        $b \leftarrow c$
    else
        $a \leftarrow c$
return c

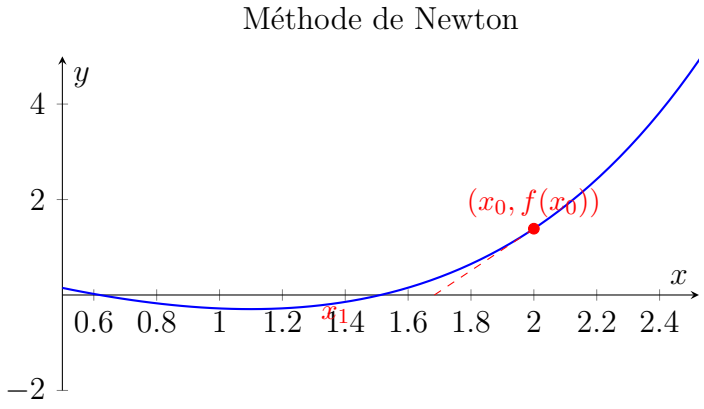
Théorème 2.2 (Convergence de la bissection)**.** *Après n itérations : $|c_n - x^*| \leq \frac{b-a}{2^{n+1}}$. La convergence est **linéaire** avec taux $1/2$.*

Démonstration. À chaque itération, l'intervalle est divisé par 2. Après n itérations, $|I_n| = (b-a)/2^n$ et $|c_n - x^*| \leq |I_n|/2$. □

2.3 Méthode de Newton

Définition 2.3. L'itération de Newton pour $f(x) = 0$ est :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$



Algorithme 2 : Méthode de Newton

Entrée : f, f', x_0, tolérance ε, $N_{\max}$
Sortie : Approximation x_n
pour $n = 0, 1, \ldots, N_{\max}$ **faire**
  $x_{n+1} \leftarrow x_n - f(x_n)/f'(x_n)$;
  si $|x_{n+1} - x_n| < \varepsilon$ **alors**
    return x_{n+1}

Théorème 2.4 (Convergence quadratique de Newton)**.** *Si $f \in C^2$, $f(x^*) = 0$, $f'(x^*) \neq 0$ et x_0 suffisamment proche de x^*, alors :*

$$|e_{n+1}| \leq \frac{M}{2m}|e_n|^2,$$

où $e_n = x_n - x^*$, $M = \sup|f''|$, $m = \inf|f'|$ au voisinage de x^*.

Démonstration. Par Taylor : $0 = f(x^*) = f(x_n) + f'(x_n)(x^* - x_n) + \frac{f''(\xi_n)}{2}(x^* - x_n)^2$. Divisant par $f'(x_n)$:

$$0 = \frac{f(x_n)}{f'(x_n)} + (x^* - x_n) + \frac{f''(\xi_n)}{2f'(x_n)}e_n^2.$$

Or $x_{n+1} = x_n - f(x_n)/f'(x_n)$, donc $e_{n+1} = x_{n+1} - x^* = -\frac{f''(\xi_n)}{2f'(x_n)}e_n^2$. □

2.4 Méthode de la sécante

Définition 2.5. On remplace $f'(x_n)$ par une différence finie :

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Théorème 2.6. *L'ordre de convergence de la sécante est le **nombre d'or** $\varphi = (1 + \sqrt{5})/2 \approx 1{,}618$.*

2.5 Méthode du point fixe

Définition 2.7. Trouver $x = g(x)$. Itération : $x_{n+1} = g(x_n)$.

Théorème 2.8 (Théorème du point fixe de Banach)**.** *Si $g : [a, b] \to [a, b]$ est une contraction ($|g'(x)| \leq L < 1$ sur $[a, b]$), alors g a un unique point fixe x^* et $x_n \to x^*$ avec :*

$$|x_n - x^*| \leq \frac{L^n}{1 - L}|x_1 - x_0|.$$

2.6 Comparaison des méthodes

Méthode	Ordre	Appels f	Robustesse
Bissection	1 (linéaire)	1/it.	Très robuste
Newton	2 (quadratique)	$f + f'$/it.	Sensible à x_0
Sécante	$\varphi \approx 1{,}618$	1/it.	Modérée

2.7 Exemple numérique

Racine de $f(x) = e^x - 3x$ avec $x_0 = 2$ (Newton) :

n	x_n	$f(x_n)$	$\lvert x_n - x_{n-1}\rvert$
0	2.000000	1.389056	—
1	1.373418	0.225698	0.627
2	1.524247	-0.046	0.151
3	1.512127	-0.000374	0.012
4	1.512135	$< 10^{-9}$	8×10^{-6}

2.8 Implémentation

Python

```python
import numpy as np

def bisection(f, a, b, tol=1e-12):
    while b - a > tol:
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return (a + b) / 2

def newton(f, df, x0, tol=1e-12, maxiter=100):
    x = x0
    for i in range(maxiter):
        dx = f(x) / df(x)
        x -= dx
        if abs(dx) < tol:
            return x, i+1
    return x, maxiter

def secant(f, x0, x1, tol=1e-12, maxiter=100):
    for i in range(maxiter):
        fx0, fx1 = f(x0), f(x1)
        x2 = x1 - fx1 * (x1 - x0) / (fx1 - fx0)
        if abs(x2 - x1) < tol:
            return x2, i+1
```

```
        x0, x1 = x1, x2
    return x1, maxiter

# Test
f = lambda x: np.exp(x) - 3*x
df = lambda x: np.exp(x) - 3

print(f"Bissection : {bisection(f, 0, 1):.12f}")
x_newton, n = newton(f, df, 2.0)
print(f"Newton     : {x_newton:.12f} ({n} iterations)")
x_sec, n = secant(f, 0.0, 1.0)
print(f"Secante    : {x_sec:.12f} ({n} iterations)")
```

Julia

```
function newton(f, df, x0; tol=1e-12, maxiter=100)
    x = x0
    for i in 1:maxiter
        dx = f(x) / df(x)
        x -= dx
        abs(dx) < tol && return x, i
    end
    return x, maxiter
end

f(x) = exp(x) - 3x
df(x) = exp(x) - 3
x, n = newton(f, df, 2.0)
println("Newton: x = $x ($n iterations)")
```

2.9 Exercices

Exercice 2.1 ($\star$). Trouver $\sqrt{2}$ par Newton appliqué à $f(x) = x^2 - 2$. Combien d'itérations pour 15 décimales ?

Exercice 2.2 ($\star\star$). Montrer que Newton appliqué à $f(x) = x^2$ converge linéairement (racine double). Généraliser : que se passe-t-il quand $f'(x^*) = 0$?

Exercice 2.3 ($\star\star$). Démontrer l'ordre φ de la sécante en posant $e_{n+1} \approx C\, e_n^p$ et en résolvant $p^2 - p - 1 = 0$.

Exercice 2.4 ($\star\star\star$ – Projet). Implémenter la méthode de Brent (combinaison bissection + sécante). Comparer avec les méthodes simples sur 10 fonctions test.

Formules clés

$$\text{Bissection : } |e_n| \leq \frac{b-a}{2^{n+1}}$$

$$\text{Newton : } x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad |e_{n+1}| \leq C|e_n|^2$$

$$\text{Sécante : } x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \quad \text{ordre } \varphi$$

Chapitre 3

Méthodes Directes pour Systèmes Linéaires

3.1 Exemple motivant

Un circuit électrique à n nœuds produit un système $Ax = b$ de n équations. Pour $n = 1000$, il faut une méthode systématique et efficace.

3.2 Élimination de Gauss et factorisation LU

Définition 3.1 (Factorisation LU)**.** Trouver L triangulaire inférieure (avec $\ell_{ii} = 1$) et U triangulaire supérieure telles que $A = LU$. Alors $Ax = b$ se résout par $Ly = b$ (descente) puis $Ux = y$ (remontée).

Algorithme 3 : Factorisation LU (sans pivot)

Entrée : Matrice $A \in \mathbb{R}^{n \times n}$
Sortie : L, U tels que $A = LU$
pour $k = 1, \ldots, n-1$ **faire**
  pour $i = k+1, \ldots, n$ **faire**
    $\ell_{ik} \leftarrow a_{ik}/a_{kk}$;
    pour $j = k+1, \ldots, n$ **faire**
      $a_{ij} \leftarrow a_{ij} - \ell_{ik} \cdot a_{kj}$;
$U \leftarrow$ partie triangulaire supérieure de A ;

Théorème 3.2 (Existence de LU)**.** *Si toutes les sous-matrices principales de A sont inversibles, alors la factorisation LU existe et est unique.*

Théorème 3.3 (Coût)**.** *La factorisation LU coûte $\frac{2n^3}{3} + \mathcal{O}(n^2)$ opérations flottantes.*

Démonstration. L'étape k effectue $(n-k)^2$ multiplications et soustractions. Total : $\sum_{k=1}^{n-1} 2(n-k)^2 = 2\sum_{j=1}^{n-1} j^2 = \frac{2n^3}{3} + \mathcal{O}(n^2)$. □

3.3 Pivot partiel

Attention

Sans pivotement, l'algorithme peut être instable (division par un petit pivot). Le **pivot partiel** permute les lignes pour maximiser $|a_{kk}|$.

On obtient $PA = LU$ où P est une matrice de permutation.

Théorème 3.4 (Stabilité). *Avec pivot partiel, l'élimination de Gauss est* ***rétrograde stable*** *:*

$$(A + \Delta A)\hat{x} = b, \quad \|\Delta A\| \leq g(n)\, \varepsilon_{\text{mach}}\, \|A\|,$$

où $g(n)$ est le facteur de croissance (borné par 2^{n-1} en théorie, rarement dépassé en pratique).

3.4 Factorisation de Cholesky

Théorème 3.5 (Factorisation de Cholesky). *Si A est* ***symétrique définie positive*** *(SDP), alors il existe une unique matrice triangulaire inférieure L à coefficients diagonaux positifs telle que $A = LL^\top$.*

Démonstration. Par récurrence sur n. Écrivons $A = \begin{pmatrix} a_{11} & \mathbf{v}^\top \\ \mathbf{v} & A' \end{pmatrix}$. Comme A SDP, $a_{11} > 0$. Posons $\ell_{11} = \sqrt{a_{11}}$, $\mathbf{l} = \mathbf{v}/\ell_{11}$. Alors $A' - \mathbf{l}\mathbf{l}^\top$ est SDP de taille $(n-1)\times(n-1)$ et on applique l'hypothèse de récurrence. □

Algorithme 4 : Factorisation de Cholesky

Entrée : A SDP, $n \times n$
Sortie : L telle que $A = LL^\top$
pour $j = 1, \ldots, n$ **faire**
  $\ell_{jj} \leftarrow \sqrt{a_{jj} - \sum_{k=1}^{j-1} \ell_{jk}^2}$;
  pour $i = j+1, \ldots, n$ **faire**
    $\ell_{ij} \leftarrow \frac{1}{\ell_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik}\ell_{jk} \right)$;

Théorème 3.6 (Coût de Cholesky). $\frac{n^3}{3} + \mathcal{O}(n^2)$ *: deux fois moins que LU.*

3.5 Exemple numérique

$A = \begin{pmatrix} 4 & 2 \\ 2 & 5 \end{pmatrix}$, $b = \begin{pmatrix} 8 \\ 9 \end{pmatrix}$. Cholesky : $L = \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix}$.

Descente $Ly = b$: $y_1 = 4$, $y_2 = (9 - 1\cdot 4)/2 = 2{,}5$. Remontée $L^\top x = y$: $x_2 = 2{,}5/2 = 1{,}25$, $x_1 = (4 - 1\cdot 1{,}25)/2 = 1{,}375$.

3.6 Implémentation

Python

```python
import numpy as np
from scipy.linalg import lu, cholesky, solve_triangular

# LU
A = np.array([[2., 1, 1], [4, 3, 3], [8, 7, 9]])
b = np.array([4., 10, 24])
P, L, U = lu(A)
y = solve_triangular(L, P @ b, lower=True)
x = solve_triangular(U, y)
print(f"LU: x = {x}")

# Cholesky
A_spd = np.array([[4., 2], [2, 5]])
b_spd = np.array([8., 9])
L_chol = cholesky(A_spd, lower=True)
y = solve_triangular(L_chol, b_spd, lower=True)
x = solve_triangular(L_chol.T, y)
print(f"Cholesky: x = {x}")
print(f"Cout LU (n=100): ~{2*100**3//3:.0f} flops")
```

Julia

```julia
using LinearAlgebra
A = [2.0 1 1; 4 3 3; 8 7 9]
b = [4.0, 10, 24]
F = lu(A)
x = F \ b
println("LU: x = $x")

A_spd = [4.0 2; 2 5]
b_spd = [8.0, 9]
C = cholesky(A_spd)
x = C \ b_spd
println("Cholesky: x = $x")
```

3.7 Exercices

Exercice 3.1 ($\star$). Factoriser $A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 8 & 14 \\ 3 & 14 & 34 \end{pmatrix}$ en LU à la main. Vérifier.

Exercice 3.2 ($\star\star$). Montrer que si A est SDP, tous ses pivots dans l'élimination de Gauss sont positifs (sans pivotement nécessaire).

Exercice 3.3 ($\star\star$). Comparer les temps de calcul de LU vs. Cholesky pour $n = 100, 500, 1000, 2000$ sur des matrices SDP aléatoires. Vérifier le rapport ~ 2.

Exercice 3.4 ($\star\star\star$ – Projet)**.** Implémenter la factorisation LU bande pour des matrices tridiagonales (algorithme de Thomas). Appliquer à la discrétisation de $-u'' = f$.

Formules clés

$$A = LU \quad (\text{coût } \tfrac{2n^3}{3})$$
$$A = LL^\top \quad (\text{Cholesky, coût } \tfrac{n^3}{3})$$
$$PA = LU \quad (\text{pivot partiel})$$
$$\mathrm{cond}(A) = \|A\| \cdot \|A^{-1}\|, \qquad \frac{\|\delta x\|}{\|x\|} \leq \mathrm{cond}(A) \frac{\|\delta b\|}{\|b\|}$$

Chapitre 4

Méthodes Itératives pour Systèmes Linéaires

4.1 Exemple motivant

En éléments finis, la matrice A est creuse ($\mathcal{O}(n)$ coefficients non nuls sur n^2). LU coûte $\mathcal{O}(n^3)$, inacceptable pour $n = 10^6$. Les méthodes itératives exploitent la structure creuse : chaque itération coûte $\mathcal{O}(n)$.

4.2 Principe général

Définition 4.1 (Méthode de décomposition)**.** On écrit $A = M - N$ avec M inversible facile. L'itération est :

$$Mx^{(k+1)} = Nx^{(k)} + b, \quad \text{soit} \quad x^{(k+1)} = Gx^{(k)} + c,$$

où $G = M^{-1}N$ est la **matrice d'itération** et $c = M^{-1}b$.

Théorème 4.2 (Convergence)**.** *La méthode converge pour tout $x^{(0)}$ si et seulement si $\rho(G) < 1$ (rayon spectral).*

Démonstration. $e^{(k)} = x^{(k)} - x^* = G^k e^{(0)}$. Or $\|G^k\| \to 0 \iff \rho(G) < 1$. □

4.3 Méthode de Jacobi

Définition 4.3. $A = D - E - F$ (diagonale, triangulaire inf., triangulaire sup.). $M = D$:

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

Matrice d'itération : $G_J = D^{-1}(E + F)$.

4.4 Méthode de Gauss–Seidel

Définition 4.4. $M = D - E$:

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

Matrice d'itération : $G_{GS} = (D - E)^{-1}F$.

Théorème 4.5 (Convergence pour matrices SDP)**.** *Si A est SDP, Gauss–Seidel converge.*

Théorème 4.6 (Convergence pour matrices à diagonale dominante)**.** *Si A est à diagonale strictement dominante ($|a_{ii}| > \sum_{j\neq i} |a_{ij}|$), alors Jacobi et Gauss–Seidel convergent.*

4.5 Relaxation (SOR)

Définition 4.7. On introduit un paramètre $\omega \in (0, 2)$:

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

$\omega = 1$: Gauss–Seidel. $\omega > 1$: sur-relaxation. $\omega < 1$: sous-relaxation.

Théorème 4.8. *SOR converge ssi $0 < \omega < 2$. L'ω optimal dépend de $\rho(G_J)$.*

4.6 Méthode du gradient conjugué

Définition 4.9. Pour A SDP, résoudre $Ax = b$ équivaut à minimiser $\phi(x) = \frac{1}{2}x^\top Ax - b^\top x$.

Algorithme 5 : Gradient conjugué

Entrée : A SDP, b, x_0, tolérance ε
Sortie : $x \approx A^{-1}b$
$r_0 \leftarrow b - Ax_0$, $p_0 \leftarrow r_0$;
pour $k = 0, 1, 2, \ldots$ **faire**
  $\alpha_k \leftarrow \frac{r_k^\top r_k}{p_k^\top A p_k}$;
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$;
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$;
  si $\|r_{k+1}\| < \varepsilon$ **alors**
    return x_{k+1}
  $\beta_k \leftarrow \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$;
  $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$;

Théorème 4.10 (Convergence du gradient conjugué)**.** *En arithmétique exacte, CG converge en au plus n itérations. L'erreur vérifie :*

$$\|e_k\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|e_0\|_A, \quad \kappa = \mathrm{cond}_2(A).$$

4.7 Implémentation

Python

```python
import numpy as np

def jacobi(A, b, x0, tol=1e-10, maxiter=1000):
    D = np.diag(np.diag(A))
    R = A - D
    x = x0.copy()
    for k in range(maxiter):
        x_new = np.linalg.solve(D, b - R @ x)
        if np.linalg.norm(x_new - x) < tol:
            return x_new, k+1
        x = x_new
    return x, maxiter

def conjugate_gradient(A, b, x0, tol=1e-10, maxiter=1000):
    x = x0.copy()
    r = b - A @ x
    p = r.copy()
    rs_old = r @ r
    for k in range(maxiter):
        Ap = A @ p
        alpha = rs_old / (p @ Ap)
        x += alpha * p
        r -= alpha * Ap
        rs_new = r @ r
        if np.sqrt(rs_new) < tol:
            return x, k+1
        p = r + (rs_new / rs_old) * p
        rs_old = rs_new
    return x, maxiter

# Test : matrice tridiagonale
n = 100
A = np.diag(2*np.ones(n)) - np.diag(np.ones(n-1), 1) -
↪  np.diag(np.ones(n-1), -1)
b = np.ones(n)
x0 = np.zeros(n)

x_j, nj = jacobi(A, b, x0)
x_cg, ncg = conjugate_gradient(A, b, x0)
print(f"Jacobi: {nj} iterations")
print(f"CG:     {ncg} iterations")
```

Julia

```julia
function cg(A, b, x0; tol=1e-10, maxiter=1000)
    x = copy(x0)
    r = b - A * x
    p = copy(r)
```

```
    rs = dot(r, r)
    for k in 1:maxiter
        Ap = A * p
        alpha = rs / dot(p, Ap)
        x .+= alpha .* p
        r .-= alpha .* Ap
        rs_new = dot(r, r)
        sqrt(rs_new) < tol && return x, k
        p .= r .+ (rs_new / rs) .* p
        rs = rs_new
    end
    return x, maxiter
end
```

4.8 Exercices

Exercice 4.1 ($\star$)**.** Appliquer 3 itérations de Jacobi et Gauss–Seidel au système $\begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, $x^{(0)} = (0,0)^\top$.

Exercice 4.2 ($\star\star$)**.** Montrer que pour une matrice tridiagonale T_n de taille n, $\rho(G_J) = \cos(\pi/(n+1))$. En déduire le nombre d'itérations de Jacobi pour une précision ε.

Exercice 4.3 ($\star\star\star$ – Projet)**.** Comparer Jacobi, Gauss–Seidel, SOR et CG pour la discrétisation 2D du Laplacien $-\Delta u = f$ sur $[0,1]^2$. Tracer le nombre d'itérations en fonction de n.

Formules clés

$$\text{Jacobi} : x^{(k+1)} = D^{-1}(b - (A-D)x^{(k)})$$
$$\text{Gauss–Seidel} : G = (D-E)^{-1}F$$
$$\text{CG} : \|e_k\|_A \le 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|e_0\|_A$$

Chapitre 5

Interpolation Polynomiale

5.1 Exemple motivant

On dispose de mesures de température à 5 instants. On veut estimer la température à un instant intermédiaire. L'interpolation polynomiale construit un polynôme passant exactement par les points mesurés.

5.2 Problème d'interpolation

Théorème 5.1 (Existence et unicité)**.** *Étant donnés $n+1$ points distincts $(x_0, y_0), \ldots, (x_n, y_n)$, il existe un* ***unique*** *polynôme $p_n \in \mathcal{P}_n$ tel que $p_n(x_i) = y_i$ pour tout i.*

Démonstration. La matrice de Vandermonde $V_{ij} = x_i^j$ est inversible car $\det(V) = \prod_{i>j}(x_i - x_j) \neq 0$. □

5.3 Forme de Lagrange

Définition 5.2.

$$p_n(x) = \sum_{i=0}^{n} y_i \, L_i(x), \quad L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

Les L_i sont les **polynômes de base de Lagrange** : $L_i(x_j) = \delta_{ij}$.

5.4 Forme de Newton

Définition 5.3 (Différences divisées)**.**

$$f[x_i] = f(x_i), \quad f[x_i, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}.$$

Définition 5.4 (Forme de Newton)**.**

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + f[x_0, \ldots, x_n] \prod_{j=0}^{n-1} (x - x_j).$$

Évaluation par l'algorithme de Horner généralisé : $\mathcal{O}(n)$ opérations.

5.5 Erreur d'interpolation

Théorème 5.5 (Erreur d'interpolation). *Si $f \in C^{n+1}([a,b])$, alors pour tout $x \in [a,b]$:*

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!} \prod_{i=0}^{n} (x - x_i),$$

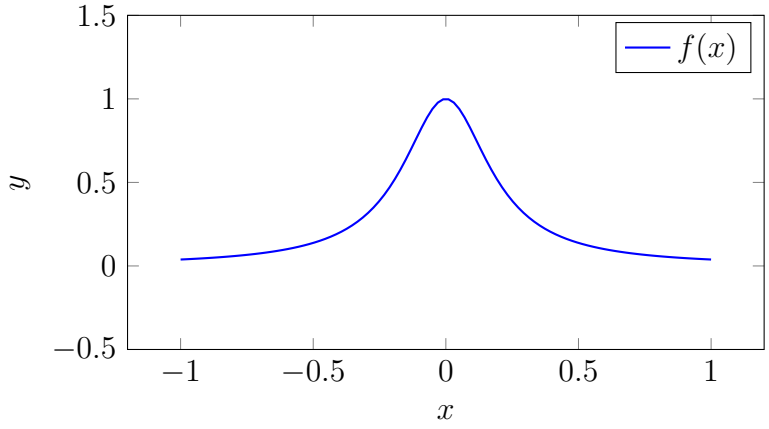
où $\xi_x \in (a,b)$ dépend de x.

Démonstration. Posons $w(t) = \prod(t - x_i)$ et $\varphi(t) = f(t) - p_n(t) - \lambda w(t)$ avec λ choisi pour que $\varphi(x) = 0$ (pour un x fixé $\neq x_i$). Alors φ s'annule en $n+2$ points. Par Rolle appliqué $n+1$ fois, $\varphi^{(n+1)}$ s'annule en un point ξ. Or $\varphi^{(n+1)} = f^{(n+1)} - \lambda(n+1)!$, d'où $\lambda = f^{(n+1)}(\xi)/(n+1)!$. □

Attention

Le **phénomène de Runge** : pour $f(x) = 1/(1+25x^2)$ sur $[-1,1]$ avec des nœuds équidistants, l'erreur d'interpolation *diverge* quand $n \to \infty$ près des bords.

Phénomène de Runge



5.6 Nœuds de Tchebychev

Définition 5.6. Les nœuds de Tchebychev sur $[-1,1]$:

$$x_k = \cos\left(\frac{2k+1}{2(n+1)}\pi\right), \quad k = 0, \ldots, n.$$

Ils minimisent $\max_{x\in[-1,1]} |\prod(x - x_i)|$.

Théorème 5.7. *Avec les nœuds de Tchebychev :* $\max|w(x)| \leq 2^{-n}$, *ce qui évite le phénomène de Runge pour les fonctions analytiques.*

5.7 Splines cubiques

Définition 5.8 (Spline cubique d'interpolation). $s \in C^2([a,b])$, $s|_{[x_i,x_{i+1}]} \in \mathcal{P}_3$, $s(x_i) = y_i$. Conditions aux bords :

- **Naturelle** : $s''(x_0) = s''(x_n) = 0$.
- **Clampée** : $s'(x_0) = f'(x_0)$, $s'(x_n) = f'(x_n)$.

Théorème 5.9 (Erreur spline cubique). *Si $f \in C^4$ et $h = \max(x_{i+1} - x_i)$:*

$$\|f - s\|_\infty \leq \frac{5}{384} h^4 \|f^{(4)}\|_\infty.$$

5.8 Implémentation

Python

```python
import numpy as np
from scipy.interpolate import lagrange, CubicSpline
import matplotlib.pyplot as plt

# Interpolation de Lagrange
x_nodes = np.array([-1, -0.5, 0, 0.5, 1])
f = lambda x: 1 / (1 + 25*x**2)
y_nodes = f(x_nodes)
p = lagrange(x_nodes, y_nodes)

x_fine = np.linspace(-1, 1, 300)
plt.figure(figsize=(10, 5))
plt.plot(x_fine, f(x_fine), 'b-', label='f(x)', lw=2)
plt.plot(x_fine, p(x_fine), 'r--', label=f'Lagrange deg
↪ {len(x_nodes)-1}')
plt.plot(x_nodes, y_nodes, 'ko', markersize=8)
plt.legend(); plt.title("Interpolation de Lagrange")
plt.savefig("ch05_lagrange.pdf"); plt.show()

# Noeuds de Tchebychev
n = 15
cheb = np.cos((2*np.arange(n+1)+1)/(2*(n+1))*np.pi)
y_cheb = f(cheb)
p_cheb = lagrange(cheb, y_cheb)
print(f"Erreur equidist (n=15):
↪ {np.max(np.abs(f(x_fine)-lagrange(np.linspace(-1,1,16),
↪ f(np.linspace(-1,1,16)))(x_fine))):.4f}")
print(f"Erreur Tchebychev (n=15):
↪ {np.max(np.abs(f(x_fine)-p_cheb(x_fine))):.6f}")

# Spline cubique
cs = CubicSpline(x_nodes, y_nodes)
print(f"Spline en 0.25: {cs(0.25):.6f}, exact: {f(0.25):.6f}")
```

Julia

```julia
using Interpolations
```

```
f(x) = 1 / (1 + 25x^2)
nodes = [-1.0, -0.5, 0, 0.5, 1.0]
vals = f.(nodes)

# Differences divisees (Newton)
function divided_diff(x, y)
    n = length(x)
    F = copy(y)
    for j in 2:n, i in n:-1:j
        F[i] = (F[i] - F[i-1]) / (x[i] - x[i-j+1])
    end
    return F
end
println("Coefs Newton: ", divided_diff(nodes, vals))
```

5.9 Exercices

Exercice 5.1 ($\star$)**.** Construire le polynôme de Lagrange passant par $(0,1)$, $(1,3)$, $(2,7)$ et vérifier.

Exercice 5.2 ($\star\star$)**.** Démontrer le théorème 5.5 en détail.

Exercice 5.3 ($\star\star$)**.** Comparer l'erreur d'interpolation avec nœuds équidistants vs. Tchebychev pour $f(x) = 1/(1+25x^2)$, $n = 5, 10, 15, 20$.

Exercice 5.4 ($\star\star\star$ – Projet)**.** Implémenter l'interpolation de Hermite (données de f et f'). Appliquer à la construction de courbes de Bézier.

Formules clés

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod (x - x_i)$$

$$x_k^{\text{Tcheb}} = \cos\left(\frac{2k+1}{2(n+1)}\pi\right)$$

$$\|f - s\|_\infty \leq \frac{5}{384} h^4 \|f^{(4)}\|_\infty$$

Chapitre 6

Approximation — Moindres Carrés et Tchebychev

6.1 Exemple motivant

On a 100 mesures expérimentales bruitées. Faire passer un polynôme de degré 99 par tous les points serait absurde (sur-ajustement). On cherche un polynôme de faible degré qui *approche* les données au mieux : c'est l'approximation par moindres carrés.

6.2 Approximation au sens des moindres carrés

Définition 6.1 (Problème des moindres carrés)**.** Étant donnés $(x_i, y_i)_{i=1}^m$ et des fonctions $\varphi_0, \ldots, \varphi_n$ $(n < m)$, trouver $c_0, \ldots, c_n$ minimisant :

$$S(c) = \sum_{i=1}^{m} \left(y_i - \sum_{j=0}^{n} c_j \varphi_j(x_i) \right)^2 = \|y - Ac\|_2^2,$$

où $A_{ij} = \varphi_j(x_i)$.

Théorème 6.2 (Équations normales)**.** *Le minimum est atteint pour c^* solution de :*

$$A^\top A\, c^* = A^\top y.$$

La matrice $A^\top A$ est SDP si A est de rang plein.

Démonstration. $\nabla_c \|y - Ac\|^2 = -2A^\top(y - Ac) = 0$ donne $A^\top Ac = A^\top y$. □

6.3 Polynômes orthogonaux

Définition 6.3. Pour le produit scalaire $\langle f, g \rangle = \int_a^b f(x)g(x)w(x)\,dx$ (ou discret), les polynômes orthogonaux évitent le mauvais conditionnement de $A^\top A$.

Théorème 6.4 (Projection orthogonale)**.** *Si $\{\varphi_j\}$ est une base orthonormale : $c_j = \langle y, \varphi_j \rangle$.*

6.4 Meilleure approximation uniforme

Définition 6.5. Trouver $p^* \in \mathcal{P}_n$ minimisant $\|f - p\|_\infty = \max_{x \in [a,b]} |f(x) - p(x)|$.

Théorème 6.6 (Théorème d'équioscillation de Tchebychev)**.** *p^* est la meilleure approximation uniforme de degré n ssi l'erreur $f - p^*$ équioscille en au moins $n + 2$ points.*

6.5 Implémentation

Python

```python
import numpy as np
import matplotlib.pyplot as plt

# Moindres carres polynomiaux
np.random.seed(42)
x = np.linspace(0, 5, 50)
y = 2 + 0.5*x + 0.3*x**2 + np.random.normal(0, 1, 50)

for deg in [1, 2, 5]:
    c = np.polyfit(x, y, deg)
    p = np.poly1d(c)
    residual = np.sqrt(np.mean((y - p(x))**2))
    print(f"Degre {deg}: RMSE = {residual:.4f}")

# Avec equations normales
A = np.column_stack([np.ones_like(x), x, x**2])
c_normal = np.linalg.solve(A.T @ A, A.T @ y)
print(f"Coefs (eq. normales): {c_normal}")

plt.scatter(x, y, s=10, label='Donnees')
x_fine = np.linspace(0, 5, 200)
plt.plot(x_fine, np.polyval(np.polyfit(x, y, 2), x_fine), 'r-', label='MC
↪  deg 2')
plt.legend(); plt.savefig("ch06_mc.pdf"); plt.show()
```

6.6 Exercices

Exercice 6.1 ($\star$)**.** Ajuster une droite $y = a + bx$ aux données $(1, 2{,}1)$, $(2, 3{,}8)$, $(3, 6{,}2)$, $(4, 7{,}9)$, $(5, 10{,}3)$ par moindres carrés. Calculer le RMSE.

Exercice 6.2 ($\star\star$)**.** Montrer que le conditionnement de $A^\top A$ est le carré du conditionnement de A. Pourquoi est-il préférable d'utiliser la factorisation QR ?

Exercice 6.3 ($\star\star\star$ – Projet)**.** Implémenter l'approximation de Tchebychev par l'algorithme de Remez. Comparer avec les moindres carrés sur des fonctions test.

Formules clés

$$\min_c \|y - Ac\|_2^2 \implies A^\top A c = A^\top y$$

$$\text{cond}(A^\top A) = \text{cond}(A)^2$$

Équioscillation : erreur change de signe en $n+2$ points

Chapitre 7

Dérivation et Intégration Numérique

7.1 Exemple motivant

Calculer $\int_0^1 e^{-x^2}\,dx$: pas de primitive en forme close. On utilise des formules de quadrature. De même, $f'(x)$ peut être approchée par des différences finies lorsque f n'est connue que numériquement.

7.2 Dérivation numérique

Définition 7.1 (Différences finies)**.**

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \qquad \text{(progressive, } \mathcal{O}(h)\text{)} \qquad (7.1)$$

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} \qquad \text{(centrée, } \mathcal{O}(h^2)\text{)} \qquad (7.2)$$

$$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \qquad (\mathcal{O}(h^2)) \qquad (7.3)$$

Théorème 7.2 (Erreur de la différence centrée)**.** *Si* $f \in C^3$ *:* $\frac{f(x+h)-f(x-h)}{2h} = f'(x) + \frac{h^2}{6} f'''(\xi)$.

Attention

Diminuer h améliore l'erreur de troncature mais augmente l'erreur d'arrondi. Le h optimal est $h^* \sim \varepsilon_{\text{mach}}^{1/3} \approx 6 \times 10^{-6}$ en double précision.

7.3 Formules de Newton–Cotes

Principe : remplacer f par son polynôme d'interpolation sur $[a,b]$ et intégrer.

7.3.1 Règle du trapèze

Définition 7.3.

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}[f(a)+f(b)].$$

Erreur : $-\frac{(b-a)^3}{12} f''(\xi)$.

7.3.2 Règle de Simpson

Définition 7.4.

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

Erreur : $-\frac{(b-a)^5}{2880} f^{(4)}(\xi)$.

Théorème 7.5 (Formule de Simpson — degré d'exactitude)**.** *Simpson est exacte pour les polynômes de degré ≤ 3 (un degré de mieux que prévu).*

7.4 Formules composites

Définition 7.6 (Trapèze composite)**.** n sous-intervalles, $h = (b-a)/n$:

$$T_n(f) = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right].$$

Erreur : $-\frac{(b-a)h^2}{12} f''(\xi) = \mathcal{O}(h^2)$.

Définition 7.7 (Simpson composite)**.** n pairs de sous-intervalles, $h = (b-a)/n$:

$$S_n(f) = \frac{h}{3}\left[f(a) + 4\sum_{\text{imp}} f(x_i) + 2\sum_{\text{pair}} f(x_i) + f(b)\right].$$

Erreur : $\mathcal{O}(h^4)$.

7.5 Extrapolation de Richardson

Théorème 7.8. *Si $T(h) = I + c_1 h^2 + c_2 h^4 + \cdots$, alors $\frac{4T(h/2) - T(h)}{3} = I + \mathcal{O}(h^4)$. C'est la formule de Simpson !*

7.6 Exemple numérique

$I = \int_0^1 e^{-x^2}\,dx \approx 0{,}746824$.

n	Trapèze T_n	Simpson S_n	Erreur Simpson
2	0.731531	0.747180	$3{,}6 \times 10^{-4}$
4	0.742984	0.746831	$7{,}0 \times 10^{-6}$
8	0.745866	0.746824	$1{,}1 \times 10^{-7}$
16	0.746584	0.746824	$1{,}7 \times 10^{-9}$

7.7 Implémentation

Python

```python
import numpy as np

def trapezoidal(f, a, b, n):
    x = np.linspace(a, b, n+1)
    h = (b - a) / n
    return h * (f(a)/2 + np.sum(f(x[1:-1])) + f(b)/2)

def simpson(f, a, b, n):
    assert n % 2 == 0
    x = np.linspace(a, b, n+1)
    h = (b - a) / n
    return h/3 * (f(a) + 4*np.sum(f(x[1::2])) + 2*np.sum(f(x[2:-1:2])) +
    ↪  f(b))

f = lambda x: np.exp(-x**2)
exact = 0.7468241328124271  # scipy.integrate.quad

for n in [2, 4, 8, 16, 32]:
    T = trapezoidal(f, 0, 1, n)
    S = simpson(f, 0, 1, n)
    print(f"n={n:3d}  Trap={T:.10f}  Simp={S:.10f}  "
          f"Err_T={abs(T-exact):.2e}  Err_S={abs(S-exact):.2e}")
```

Julia

```julia
function trapezoidal(f, a, b, n)
    h = (b - a) / n
    x = range(a, b, length=n+1)
    return h * (f(a)/2 + sum(f.(x[2:end-1])) + f(b)/2)
end

function simpson_rule(f, a, b, n)
    h = (b - a) / n
    x = range(a, b, length=n+1)
    return h/3 * (f(a) + 4sum(f.(x[2:2:end-1])) + 2sum(f.(x[3:2:end-2]))
    ↪  + f(b))
end

f(x) = exp(-x^2)
for n in [2, 4, 8, 16, 32]
    println("n=$n  Trap=$(trapezoidal(f,0,1,n))
    ↪  Simp=$(simpson_rule(f,0,1,n))")
end
```

7.8 Exercices

Exercice 7.1 ($\star$)**.** Calculer $\int_0^{\pi} \sin x \, dx$ par trapèze et Simpson composites ($n = 4$). Comparer à la valeur exacte.

Exercice 7.2 ($\star\star$)**.** Démontrer que Simpson est exacte pour les polynômes de degré 3.

Exercice 7.3 ($\star\star$)**.** Vérifier numériquement l'ordre $\mathcal{O}(h^2)$ du trapèze et $\mathcal{O}(h^4)$ de Simpson par un tableau d'erreurs.

Exercice 7.4 ($\star\star\star$ – Projet)**.** Implémenter l'intégration adaptative (Simpson adaptatif) et comparer avec `scipy.integrate.quad`.

Formules clés

$$T_n = \frac{h}{2}[f(a) + 2\sum f(x_i) + f(b)], \; E = \mathcal{O}(h^2)$$

$$S_n = \frac{h}{3}[f(a) + 4\sum_{\text{imp}} + 2\sum_{\text{pair}} + f(b)], \; E = \mathcal{O}(h^4)$$

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \; E = \mathcal{O}(h^2)$$

Chapitre 8

Quadrature de Gauss

8.1 Exemple motivant

Les formules de Newton–Cotes (nœuds fixés) atteignent un degré d'exactitude n ou $n+1$. En choisissant **les nœuds librement**, on peut atteindre le degré $2n+1$ avec $n+1$ points — c'est la quadrature de Gauss.

8.2 Principe

Définition 8.1 (Quadrature de Gauss)**.** Trouver $n+1$ nœuds $x_0, \dots, x_n$ et poids $w_0, \dots, w_n$ tels que :

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i)$$

soit exacte pour tout $f \in \mathcal{P}_{2n+1}$.

Théorème 8.2 (Théorème fondamental)**.** *Les nœuds de la quadrature de Gauss–Legendre à $n+1$ points sont les racines du polynôme de Legendre P_{n+1}. Le degré d'exactitude est $2n+1$.*

Esquisse. Soit $f \in \mathcal{P}_{2n+1}$. Division euclidienne : $f = q \cdot P_{n+1} + r$ avec $\deg q \le n$, $\deg r \le n$. Alors $\int_{-1}^{1} f = \int_{-1}^{1} qP_{n+1} + \int_{-1}^{1} r$. Par orthogonalité de P_{n+1} à $\mathcal{P}_n$: $\int qP_{n+1} = 0$. Et $\sum w_i f(x_i) = \sum w_i r(x_i)$ car $P_{n+1}(x_i) = 0$. Les poids sont choisis pour que la formule soit exacte sur $\mathcal{P}_n$, donc $\sum w_i r(x_i) = \int r$. □

8.3 Nœuds et poids

$n+1$	Nœuds x_i	Poids w_i
1	0	2
2	$\pm 1/\sqrt{3}$	1, 1
3	0, $\pm\sqrt{3/5}$	8/9, 5/9, 5/9

8.4 Gauss–Legendre, Gauss–Laguerre, Gauss–Hermite

Définition 8.3. Pour $\int_a^b f(x)w(x)\,dx$ avec poids w :

- **Gauss–Legendre** : $w(x) = 1$ sur $[-1, 1]$.
- **Gauss–Laguerre** : $w(x) = e^{-x}$ sur $[0, \infty)$.
- **Gauss–Hermite** : $w(x) = e^{-x^2}$ sur $(-\infty, \infty)$.

8.5 Erreur

Théorème 8.4. *L'erreur de Gauss–Legendre à $n+1$ points est :*

$$E_n(f) = \frac{2^{2n+3}[(n+1)!]^4}{(2n+3)[(2n+2)!]^3} f^{(2n+2)}(\xi).$$

8.6 Changement d'intervalle

Pour $\int_a^b f(x)\,dx$, le changement $x = \frac{b-a}{2}t + \frac{a+b}{2}$ ramène à $[-1, 1]$:

$$\int_a^b f(x)\,dx = \frac{b-a}{2} \int_{-1}^{1} f\left(\frac{b-a}{2}t + \frac{a+b}{2}\right) dt.$$

8.7 Implémentation

Python

```python
import numpy as np
from numpy.polynomial.legendre import leggauss

def gauss_legendre(f, a, b, n):
    """Quadrature de Gauss-Legendre a n points sur [a,b]."""
    nodes, weights = leggauss(n)
    # Changement d'intervalle [-1,1] -> [a,b]
    x = 0.5*(b-a)*nodes + 0.5*(a+b)
    w = 0.5*(b-a)*weights
    return np.sum(w * f(x))

f = lambda x: np.exp(-x**2)
exact = 0.7468241328124271

for n in [2, 3, 5, 10]:
    approx = gauss_legendre(f, 0, 1, n)
    print(f"n={n:2d}: I={approx:.15f}, erreur={abs(approx-exact):.2e}")
```

Julia

```julia
using FastGaussQuadrature

function gauss_legendre_ab(f, a, b, n)
    nodes, weights = gausslegendre(n)
    x = 0.5(b-a) .* nodes .+ 0.5(a+b)
```

```
    w = 0.5(b-a) .* weights
    return sum(w .* f.(x))
end

f(x) = exp(-x^2)
for n in [2, 3, 5, 10]
    I = gauss_legendre_ab(f, 0, 1, n)
    println("n=$n: I=$I")
end
```

8.8 Exercices

Exercice 8.1 ($\star$)**.** Calculer $\int_0^1 x^3\,dx$ par Gauss–Legendre à 2 points. Vérifier l'exactitude.

Exercice 8.2 ($\star\star$)**.** Montrer que les poids de Gauss sont toujours positifs.

Exercice 8.3 ($\star\star\star$ – Projet)**.** Comparer la convergence de Simpson composite (n points) et Gauss–Legendre (n points) pour $\int_0^1 e^{-x^2}dx$. Tracer l'erreur en échelle log-log.

Formules clés

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i), \quad \text{exacte pour } \mathcal{P}_{2n+1}$$

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}t + \frac{a+b}{2}\right)dt$$

Chapitre 9

Résolution Numérique des EDO

9.1 Exemple motivant

Le mouvement d'un pendule simple est régi par $\theta'' = -\frac{g}{L}\sin\theta$. Pour de grandes oscillations, pas de solution analytique. On discrétise en temps et on itère.

9.2 Problème de Cauchy

Définition 9.1.

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0, \quad t \in [t_0, T].$$

Discrétisation : $t_n = t_0 + nh$, $y_n \approx y(t_n)$.

9.3 Méthode d'Euler explicite

Définition 9.2.

$$y_{n+1} = y_n + h\, f(t_n, y_n).$$

Algorithme 6 : Euler explicite

Entrée : f, t_0, T, y_0, h
Sortie : $(t_n, y_n)_{n=0}^{N}$
$N \leftarrow \lfloor (T - t_0)/h \rfloor$;
pour $n = 0, \ldots, N-1$ **faire**
  $y_{n+1} \leftarrow y_n + h\, f(t_n, y_n)$;
  $t_{n+1} \leftarrow t_n + h$;

Théorème 9.3 (Erreur locale et globale d'Euler)**.**

- *Erreur locale de troncature :* $\tau_n = \frac{y(t_{n+1}) - y(t_n)}{h} - f(t_n, y(t_n)) = \mathcal{O}(h)$.
- *Erreur globale :* $\max_n |y(t_n) - y_n| = \mathcal{O}(h)$ *(méthode d'ordre 1).*

Esquisse. Par Taylor : $y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\xi)$. Donc $\tau_n = \frac{h}{2}y''(\xi) = \mathcal{O}(h)$. L'erreur globale s'accumule sur $N = T/h$ pas : $e_N \leq \frac{e^{LT}-1}{L} \cdot \max|\tau_n| = \mathcal{O}(h)$. □

9.4 Méthode d'Euler implicite

Définition 9.4.

$$y_{n+1} = y_n + h\, f(t_{n+1}, y_{n+1}).$$

Nécessite la résolution d'une équation (Newton) à chaque pas. Avantage : **A-stabilité**.

9.5 Méthodes de Runge–Kutta

Définition 9.5 (RK4 classique)**.**

$$k_1 = f(t_n, y_n), \tag{9.1}$$
$$k_2 = f(t_n + h/2,\ y_n + hk_1/2), \tag{9.2}$$
$$k_3 = f(t_n + h/2,\ y_n + hk_2/2), \tag{9.3}$$
$$k_4 = f(t_n + h,\ y_n + hk_3), \tag{9.4}$$
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{9.5}$$

Théorème 9.6 (Ordre de RK4)**.** *RK4 est d'ordre 4 : erreur globale $\mathcal{O}(h^4)$.*

9.6 Tableau de Butcher

Définition 9.7. Un schéma de Runge–Kutta à s étages est défini par :

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^\top \end{array}$$

Pour RK4 :

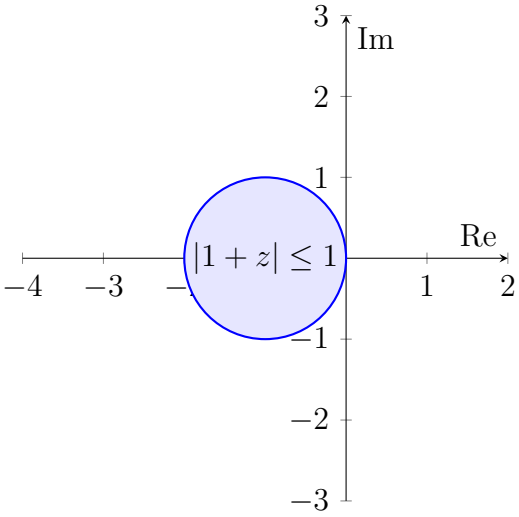
$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

9.7 Stabilité

Définition 9.8 (Équation test)**.** $y' = \lambda y$, $\lambda \in \mathbb{C}$, $\mathrm{Re}(\lambda) < 0$. Le facteur d'amplification est $R(z)$ avec $z = h\lambda$: $y_{n+1} = R(z) y_n$.

- Euler explicite : $R(z) = 1 + z$. Stable si $|1 + z| < 1$.
- Euler implicite : $R(z) = 1/(1 - z)$. Stable pour tout $\mathrm{Re}(z) < 0$ (A-stable).
- RK4 : $R(z) = 1 + z + z^2/2 + z^3/6 + z^4/24$.

Région de stabilité d'Euler explicite



9.8 Exemple numérique

$y' = -y$, $y(0) = 1$, solution exacte $y = e^{-t}$.

h	Euler	Err. Euler	RK4	Err. RK4
0.1	0.348678	$1{,}9 \times 10^{-2}$	0.367879	$3{,}8 \times 10^{-8}$
0.05	0.358486	$9{,}4 \times 10^{-3}$	0.367879	$2{,}4 \times 10^{-9}$
0.01	0.366032	$1{,}8 \times 10^{-3}$	0.367879	$< 10^{-15}$

9.9 Implémentation

Python

```python
import numpy as np
import matplotlib.pyplot as plt

def euler(f, t0, T, y0, h):
    t = np.arange(t0, T+h, h)
    y = np.zeros((len(t), len(np.atleast_1d(y0))))
    y[0] = y0
    for n in range(len(t)-1):
        y[n+1] = y[n] + h * np.array(f(t[n], y[n]))
    return t, y

def rk4(f, t0, T, y0, h):
    t = np.arange(t0, T+h, h)
    y = np.zeros((len(t), len(np.atleast_1d(y0))))
    y[0] = y0
    for n in range(len(t)-1):
        k1 = np.array(f(t[n], y[n]))
        k2 = np.array(f(t[n]+h/2, y[n]+h*k1/2))
```

```
        k3 = np.array(f(t[n]+h/2, y[n]+h*k2/2))
        k4 = np.array(f(t[n]+h, y[n]+h*k3))
        y[n+1] = y[n] + h/6 * (k1 + 2*k2 + 2*k3 + k4)
    return t, y

# Test : y' = -y
f = lambda t, y: -y
t_e, y_e = euler(f, 0, 1, [1.0], 0.1)
t_r, y_r = rk4(f, 0, 1, [1.0], 0.1)
exact = np.exp(-t_e)

plt.figure(figsize=(8, 5))
plt.plot(t_e, exact, 'k-', lw=2, label='Exact')
plt.plot(t_e, y_e, 'ro--', label='Euler')
plt.plot(t_r, y_r, 'bs-', label='RK4')
plt.legend(); plt.xlabel('t'); plt.ylabel('y')
plt.title("Euler vs RK4"); plt.savefig("ch09_edo.pdf"); plt.show()
```

Julia

```
function rk4(f, t0, T, y0, h)
    t = t0:h:T
    y = [copy(y0)]
    for n in 1:length(t)-1
        yn = y[end]
        k1 = f(t[n], yn)
        k2 = f(t[n]+h/2, yn .+ h/2 .* k1)
        k3 = f(t[n]+h/2, yn .+ h/2 .* k2)
        k4 = f(t[n]+h, yn .+ h .* k3)
        push!(y, yn .+ h/6 .* (k1 .+ 2k2 .+ 2k3 .+ k4))
    end
    return collect(t), y
end

f(t, y) = -y
t, y = rk4(f, 0.0, 1.0, [1.0], 0.1)
println("y(1) = $(y[end][1]), exact = $(exp(-1))")
```

9.10 Exercices

Exercice 9.1 ($\star$). Appliquer Euler explicite à $y' = y$, $y(0) = 1$, $h = 0{,}1$, pour $t \in [0, 1]$. Comparer à e^t.

Exercice 9.2 ($\star\star$). Montrer que RK4 est d'ordre 4 en développant $y(t+h)$ par Taylor jusqu'au terme h^4.

Exercice 9.3 ($\star\star$). Tracer la région de stabilité de RK4 numériquement.

Exercice 9.4 ($\star\star\star$ – Projet). Résoudre le système de Lorenz ($\sigma = 10, \rho = 28, \beta = 8/3$) par RK4. Visualiser l'attracteur en 3D. Étudier la sensibilité aux conditions initiales.

Formules clés

$$\text{Euler} : y_{n+1} = y_n + hf(t_n, y_n), \quad \mathcal{O}(h)$$
$$\text{RK4} : y_{n+1} = y_n + \tfrac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad \mathcal{O}(h^4)$$
$$\text{Stabilité} : |R(h\lambda)| < 1$$

Chapitre 10

Valeurs Propres et Vecteurs Propres

10.1 Exemple motivant

L'analyse des vibrations d'une structure (pont, bâtiment) revient à trouver les fréquences propres, i.e. les valeurs propres d'une matrice de rigidité. Pour une matrice 1000×1000, calculer le polynôme caractéristique est impraticable : on utilise des méthodes itératives.

10.2 Rappels

Définition 10.1. $\lambda \in \mathbb{C}$ est **valeur propre** de $A \in \mathbb{R}^{n\times n}$ s'il existe $x \neq 0$ tel que $Ax = \lambda x$. Le vecteur x est un **vecteur propre** associé. L'ensemble des valeurs propres est le **spectre** $\sigma(A)$.

Théorème 10.2 (Propriétés spectrales)**.**

- $\sigma(A) \subset \{z : |z| \leq \|A\|\}$ *(rayon spectral* $\rho(A) \leq \|A\|$*).*
- *Si A est symétrique réelle, toutes ses valeurs propres sont réelles.*
- *Si A est symétrique définie positive, toutes ses valeurs propres sont strictement positives.*
- *Théorème de Gerschgorin :* $\sigma(A) \subset \bigcup_{i=1}^{n} D(a_{ii}, R_i)$ *où* $R_i = \sum_{j\neq i} |a_{ij}|$.

10.3 Méthode de la puissance itérée

Définition 10.3. Pour trouver la valeur propre dominante λ_1 (celle de plus grand module) et le vecteur propre associé.

Algorithme 7 : Puissance itérée

Entrée : $A \in \mathbb{R}^{n\times n}$, $x^{(0)}$ vecteur initial, tolérance ε, $k_{\max}$
Sortie : λ_1, x_1
pour $k = 1, 2, \ldots, k_{\max}$ **faire**
  $w \leftarrow Ax^{(k-1)}$;
  $x^{(k)} \leftarrow w/\|w\|$;
  $\lambda^{(k)} \leftarrow (x^{(k)})^\top A x^{(k)}$; `// quotient de Rayleigh`
  si $|\lambda^{(k)} - \lambda^{(k-1)}| < \varepsilon$ **alors**
    retourner $\lambda^{(k)}$, $x^{(k)}$;

Théorème 10.4 (Convergence de la puissance itérée)**.** *Soit $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. Si $x^{(0)}$ a une composante non nulle selon le vecteur propre associé à λ_1, alors :*

$$|\lambda^{(k)} - \lambda_1| = \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right).$$

La convergence est linéaire de rapport $|\lambda_2/\lambda_1|$.

Esquisse. Décomposons $x^{(0)} = \alpha_1 v_1 + \cdots + \alpha_n v_n$ dans la base de vecteurs propres. Alors $A^k x^{(0)} = \alpha_1 \lambda_1^k \left[v_1 + \sum_{i\geq 2} \frac{\alpha_i}{\alpha_1}\left(\frac{\lambda_i}{\lambda_1}\right)^k v_i\right]$. Quand $k \to \infty$, le terme entre crochets tend vers v_1. □

10.4 Méthode de la puissance inverse

Définition 10.5. Pour trouver la valeur propre la plus proche d'un **décalage** μ : on applique la puissance itérée à $(A-\mu I)^{-1}$, dont la valeur propre dominante est $1/(\lambda_{\min}-\mu)$.

Algorithme 8 : Puissance inverse avec décalage

Entrée : A, μ, $x^{(0)}$, ε, $k_{\max}$
Sortie : λ, x
Factoriser $A - \mu I = LU$;
pour $k = 1, 2, \ldots, k_{\max}$ **faire**
  Résoudre $(A - \mu I)w = x^{(k-1)}$;
  $x^{(k)} \leftarrow w/\|w\|$;
  $\lambda^{(k)} \leftarrow (x^{(k)})^\top A x^{(k)}$;
  si $|\lambda^{(k)} - \lambda^{(k-1)}| < \varepsilon$ **alors**
    retourner $\lambda^{(k)}$, $x^{(k)}$;

Théorème 10.6. *La puissance inverse avec décalage μ converge vers la valeur propre λ_j la plus proche de μ, avec un taux $|\lambda_j - \mu|/|\lambda_k - \mu|$ où λ_k est la deuxième plus proche.*

10.5 Méthode QR

Définition 10.7 (Itération QR)**.** À partir de $A_0 = A$:

1. Factoriser $A_k = Q_k R_k$ (factorisation QR).
2. Poser $A_{k+1} = R_k Q_k$.

A_k converge vers une matrice triangulaire supérieure (ou quasi-triangulaire) dont la diagonale contient les valeurs propres.

Algorithme 9 : Itération QR basique

Entrée : $A \in \mathbb{R}^{n \times n}$, $k_{\max}$
Sortie : Valeurs propres approchées
$A_0 \leftarrow A$;
pour $k = 0, 1, \ldots, k_{\max} - 1$ **faire**
  $Q_k R_k \leftarrow A_k$; `// factorisation QR`
  $A_{k+1} \leftarrow R_k Q_k$;
retourner $\mathrm{diag}(A_{k_{\max}})$;

Théorème 10.8 (Convergence de l'algorithme QR)**.** *Si A est réelle avec des valeurs propres de modules distincts $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0$, alors A_k converge vers une matrice triangulaire supérieure. Les éléments sous-diagonaux satisfont :*

$$|(A_k)_{ij}| = \mathcal{O}\left(\left|\frac{\lambda_i}{\lambda_j}\right|^k\right), \quad i > j.$$

10.6 Améliorations pratiques

10.6.1 Réduction de Hessenberg

Définition 10.9. Avant l'itération QR, on réduit A en forme de Hessenberg supérieure ($h_{ij} = 0$ pour $i > j + 1$) par des transformations de Householder. Coût : $\frac{10}{3}n^3$ opérations. Chaque itération QR sur une matrice de Hessenberg coûte alors $\mathcal{O}(n^2)$ au lieu de $\mathcal{O}(n^3)$.

10.6.2 Décalage de Wilkinson

Définition 10.10. À chaque itération, on factorise $A_k - \mu_k I = Q_k R_k$ puis $A_{k+1} = R_k Q_k + \mu_k I$. Le décalage μ_k est la valeur propre de la sous-matrice 2×2 en bas à droite la plus proche de $a_{nn}^{(k)}$. Cela assure une convergence **cubique**.

10.7 Décomposition en valeurs singulières (SVD)

Définition 10.11. Pour toute matrice $A \in \mathbb{R}^{m \times n}$:

$$A = U \Sigma V^\top,$$

où $U \in \mathbb{R}^{m \times m}$ et $V \in \mathbb{R}^{n \times n}$ sont orthogonales, $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_p)$ avec $\sigma_1 \geq \cdots \geq \sigma_p \geq 0$, $p = \min(m, n)$.

Théorème 10.12 (Propriétés de la SVD)**.**

- $\sigma_i = \sqrt{\lambda_i(A^\top A)}$.
- $\|A\|_2 = \sigma_1$, $\|A\|_F = \sqrt{\sum \sigma_i^2}$.

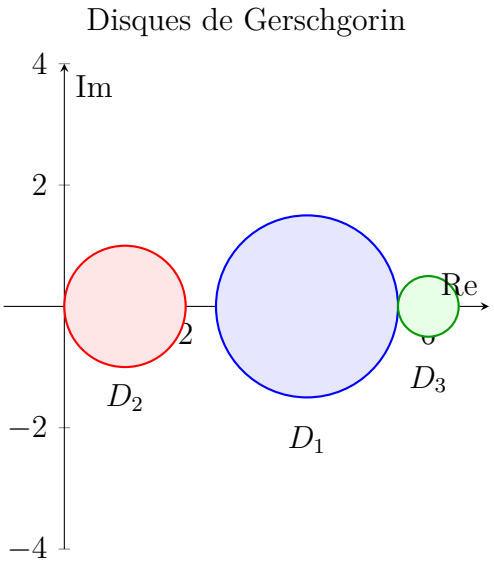
- $\mathrm{cond}_2(A) = \sigma_1/\sigma_n$.
- $\mathrm{rang}(A) =$ *nombre de* $\sigma_i > 0$.
- *Meilleure approximation de rang* r *:* $A_r = \sum_{i=1}^{r} \sigma_i u_i v_i^\top$ *(théorème d'Eckart–Young).*

10.8 Disques de Gerschgorin

Théorème 10.13 (Gerschgorin)**.** *Toute valeur propre de* A *se trouve dans au moins un des disques :*

$$D_i = \{z \in \mathbb{C} : |z - a_{ii}| \le R_i\}, \quad R_i = \sum_{j \neq i} |a_{ij}|.$$

De plus, si l'union de k *disques est disjointe des* $n-k$ *autres, elle contient exactement* k *valeurs propres.*



10.9 Exemple numérique

$A = \begin{pmatrix} 4 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{pmatrix}$, valeurs propres exactes : $\lambda \approx 5{,}049,\ 2{,}643,\ 1{,}307$.

Itération k	$\lambda_1^{(k)}$	$\lambda_2^{(k)}$	$\lambda_3^{(k)}$
0 (Puissance)	4.000	–	–
5	5.023	–	–
10	5.048	–	–
20	5.049	–	–
QR $k = 5$	5.049	2.644	1.307
QR $k = 10$	5.049	2.643	1.307

10.10 Implémentation

Python

```python
import numpy as np

def power_iteration(A, x0=None, tol=1e-10, maxiter=1000):
    n = A.shape[0]
    x = x0 if x0 is not None else np.random.randn(n)
    x = x / np.linalg.norm(x)
    lam_old = 0
    for k in range(maxiter):
        w = A @ x
        x = w / np.linalg.norm(w)
        lam = x @ A @ x  # Rayleigh quotient
        if abs(lam - lam_old) < tol:
            break
        lam_old = lam
    return lam, x

def inverse_iteration(A, mu=0, x0=None, tol=1e-10, maxiter=1000):
    n = A.shape[0]
    x = x0 if x0 is not None else np.random.randn(n)
    x = x / np.linalg.norm(x)
    B = A - mu * np.eye(n)
    lu = np.linalg.lu_factor(B) if np.linalg.det(B) != 0 else None
    lam_old = 0
    for k in range(maxiter):
        if lu is not None:
            w = np.linalg.lu_solve(lu, x)
        else:
            w = np.linalg.solve(B, x)
        x = w / np.linalg.norm(w)
        lam = x @ A @ x
        if abs(lam - lam_old) < tol:
            break
        lam_old = lam
    return lam, x

def qr_algorithm(A, maxiter=100):
    Ak = A.astype(float).copy()
    n = Ak.shape[0]
    for k in range(maxiter):
        Q, R = np.linalg.qr(Ak)
        Ak = R @ Q
        # Check convergence (sub-diagonal elements)
        off = np.sum(np.abs(np.tril(Ak, -1)))
        if off < 1e-12:
            break
    return np.diag(Ak)
```

```python
# Test
A = np.array([[4, 1, 0],
              [1, 3, 1],
              [0, 1, 2]], dtype=float)

lam1, v1 = power_iteration(A)
print(f"Puissance iteree: lambda_1 = {lam1:.6f}")
print(f"numpy.linalg.eig: {sorted(np.linalg.eig(A)[0], reverse=True)}")

eigenvalues_qr = qr_algorithm(A)
print(f"QR algorithm: {sorted(eigenvalues_qr, reverse=True)}")

# SVD
B = np.array([[1, 2], [3, 4], [5, 6]], dtype=float)
U, s, Vt = np.linalg.svd(B)
print(f"Valeurs singulieres: {s}")
print(f"cond(B) = {s[0]/s[-1]:.4f}")
```

Julia

```julia
using LinearAlgebra

function power_iteration(A; x0=nothing, tol=1e-10, maxiter=1000)
    n = size(A, 1)
    x = x0 === nothing ? randn(n) : copy(x0)
    x ./= norm(x)
    lam_old = 0.0
    lam = 0.0
    for k in 1:maxiter
        w = A * x
        x = w / norm(w)
        lam = dot(x, A * x)
        abs(lam - lam_old) < tol && return lam, x
        lam_old = lam
    end
    return lam, x
end

function qr_algorithm(A; maxiter=100)
    Ak = float.(copy(A))
    for k in 1:maxiter
        Q, R = qr(Ak)
        Ak = R * Matrix(Q)
        sum(abs.(tril(Ak, -1))) < 1e-12 && break
    end
    return diag(Ak)
end

A = [4.0 1 0; 1 3 1; 0 1 2]
lam, v = power_iteration(A)
```

```
println("Puissance iteree: lambda_1 = $lam")
println("eigvals: $(eigvals(A))")
println("QR: $(sort(qr_algorithm(A), rev=true))")

# SVD
B = [1.0 2; 3 4; 5 6]
F = svd(B)
println("Valeurs singulieres: $(F.S)")
println("cond(B) = $(F.S[1]/F.S[end])")
```

10.11 Exercices

Exercice 10.1 ($\star$)**.** Appliquer la méthode de la puissance à $A = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$ avec $x^{(0)} = (1, 1)^\top$. Calculer les 5 premières itérations.

Exercice 10.2 ($\star\star$)**.** Montrer que l'itération QR préserve la similarité : $A_{k+1} = Q_k^\top A_k Q_k$.

Exercice 10.3 ($\star\star$)**.** Tracer les disques de Gerschgorin pour $A = \begin{pmatrix} 5 & 1 & 0 \\ 1 & 3 & 0{,}5 \\ 0 & 0{,}5 & 1 \end{pmatrix}$ et localiser les valeurs propres.

Exercice 10.4 ($\star\star\star$ – Projet)**.** Implémenter l'algorithme QR avec réduction de Hessenberg et décalage de Wilkinson. Comparer les performances avec l'algorithme QR basique sur des matrices aléatoires de taille $n = 50, 100, 200$.

Formules clés

$$Ax = \lambda x, \quad \rho(A) = \max |\lambda_i|$$

$$\text{Puissance} : x^{(k)} = \frac{Ax^{(k-1)}}{\|Ax^{(k-1)}\|}, \quad \mathcal{O}(|\lambda_2/\lambda_1|^k)$$

$$\text{QR} : A_k = Q_k R_k, \; A_{k+1} = R_k Q_k$$

$$\text{SVD} : A = U\Sigma V^\top, \quad \text{cond}_2(A) = \sigma_1/\sigma_n$$

Formulaire

.1 Normes vectorielles et matricielles

$$\|x\|_1 = \sum |x_i|, \quad \|x\|_2 = \sqrt{\sum x_i^2}, \quad \|x\|_\infty = \max |x_i| \tag{1}$$

$$\|A\|_1 = \max_j \sum_i |a_{ij}|, \quad \|A\|_\infty = \max_i \sum_j |a_{ij}|, \quad \|A\|_2 = \sigma_{\max}(A) \tag{2}$$

.2 Erreurs

$$\text{Erreur absolue} : |x - \tilde{x}| \tag{3}$$

$$\text{Erreur relative} : \frac{|x - \tilde{x}|}{|x|} \tag{4}$$

$$\text{Conditionnement} : \text{cond}(A) = \|A\| \cdot \|A^{-1}\| \tag{5}$$

.3 Quadratures

$$\text{Trapèzes} : \frac{h}{2}[f(a) + 2\sum f(x_i) + f(b)], \quad E = \mathcal{O}(h^2) \tag{6}$$

$$\text{Simpson} : \frac{h}{3}[f(a) + 4\sum f_{\text{imp}} + 2\sum f_{\text{pair}} + f(b)], \quad E = \mathcal{O}(h^4) \tag{7}$$

.4 Méthodes pour EDO

$$\text{Euler explicite} : y_{n+1} = y_n + hf(t_n, y_n), \quad \mathcal{O}(h) \tag{8}$$

$$\text{RK4} : y_{n+1} = y_n + \tfrac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad \mathcal{O}(h^4) \tag{9}$$

Bibliographie

[1] Quarteroni, A., Sacco, R. & Saleri, F. (2007). *Numerical Mathematics*. 2e éd., Springer.

[2] Süli, E. & Mayers, D. (2003). *An Introduction to Numerical Analysis*. Cambridge University Press.

[3] Trefethen, L.N. & Bau, D. (1997). *Numerical Linear Algebra*. SIAM.

[4] Higham, N.J. (2002). *Accuracy and Stability of Numerical Algorithms*. 2e éd., SIAM.

[5] Demailly, J.-P. (2006). *Analyse Numérique et Équations Différentielles*. EDP Sciences.