Algèbre Linéaire Numérique

François Boulier

21 février 2022

Introduction

Ce document constitue le support du cours d'Algèbre Linéaire Numérique à Polytech Lille, pour les étudiants en troisième année de la filière IS (Informatique et Statistique).

Les principaux points abordés sont la résolution de systèmes d'équations linéaires et le calcul de valeurs et de vecteurs propres. Le support de cours contient toutes les indications nécessaires pour permettre la mise en application des méthodes en Python3 (paquetage `numpy` et sous-paquetage `linalg` de `scipy`). Il contient aussi des implantations académiques en FORTRAN ainsi que des références aux bibliothèques BLAS et LAPACK qui permettent de mieux comprendre les subtilités des algorithmes. Et il contient aussi quelques indications permettant d'appeler du code FORTRAN depuis Python, en utilisant `f2py3`.

Ce cours est lié à la fois à la partie informatique et à la partie statistique de la formation IS. L'algorithme du *Page rank*, qui est à la base du moteur de recherche de Google est en fait un calcul de valeurs propres. La méthode du simplexe du cours d'optimisation, les méthodes de régression linéaire en statistiques, correspondent à des résolutions de systèmes d'équations et de moindres carrés. L'analyse en composante principale du cours de statistique exploratoire repose sur un calcul de valeur propre.

Ce cours apporte aux futurs ingénieurs une culture générale numérique qui doit leur permettre de discuter avec des numériciens : que veulent dire les mots ? quelles sont les questions et les concepts clefs de ce domaine ?

Ce cours s'inscrit dans une série (calcul numérique en IS4, calcul haute performance en IS5), qui permet tous les ans, à quelques ingénieurs IS, d'effectuer des carrières internationales étonnantes dans de grands laboratoires de recherche privés.

Pour finir, je voudrais remercier Serge Petiton qui a fondé ce cours et en a fixé la structure, Julien Jacques et Cristian Preda pour leurs relectures concernant l'analyse en composantes principales, François Lemaire et Alexandre Sedoglavic pour diverses suggestions d'amélioration, David-Alexandre Eklo (GIS3 2013) pour avoir repéré une erreur dans une preuve du chapitre 8 et Lucas Deleforge (GIS2A4 2014) pour une erreur dans un code du chapitre 6.

Table des matières

Chapitre 1

Arithmétique flottante

1.1 Nombres à virgule flottante

Sur un ordinateur, en calcul numérique, les réels sont représentés (approximés) par des nombres en virgule flottante. Dans tous les langages de programmation classiques on dispose de flottants en simple précision (codés avec 32 bits sur la machine qui me sert en ce moment) et des flottants en double précision (sur 64 bits). Ci-dessous, on décrit quelques propriétés clefs du codage et des opérations élémentaires, qui suivent la norme IEEE 754. Un flottant est représenté ainsi, en 32 et 64 bits (les nombres entre parenthèses représentent des nombres de bits) :

$S(1)$	$E(8)$	$M(23)$
$S(1)$	$E(11)$	$M(52)$

Le bit de signe S vaut 0 si le flottant est positif, 1 sinon. La valeur absolue d'un flottant s'obtient par la formule (le (2) en indice signifie que la base de numération est 2) :

$$1.M_{(2)} \times 2^{E-D}$$

où la constante D vaut 127 dans le cas des 32 bits et 1023 dans le cas des 64 bits. Remarquer qu'un bit implicite se rajoute aux bits explicites de la mantisse M. On dispose donc en réalité de 24 bits pour la mantisse, en simple précision, et de 53 bits en double précision. Le programme C donné Figure 1.1 permet d'observer la structure d'un flottant simple précision (`float` en C).

```
#include <stdio.h>

int main ()
{   float f;
    unsigned int signe, exposant, mask, bit, *p;

    scanf ("%f", &f);
    p = (unsigned int*)&f;
    signe = (*p) >> 31;
    exposant = ((*p) >> 23) & 0xff;
    printf ("signe = %d, exposant = %d, mantisse (binaire) = ",
                (int)signe, (int)exposant);
    for (mask = 1 << 22; mask; mask >>= 1)
    {   bit = (*p) & mask ? 1 : 0;
        printf ("%x", bit);
    }
    printf ("\n");
    return 0;
}
```

À l'exécution, on obtient les résultats suivants :

```
3
signe = 0, exposant = 128, mantisse (binaire) = 10000000000000000000000
.1875
signe = 0, exposant = 124, mantisse (binaire) = 10000000000000000000000
-3072
signe = 1, exposant = 138, mantisse (binaire) = 10000000000000000000000
```

FIGURE 1.1 – Programme C décomposant un flottant en signe, mantisse et exposant. On vérifie que $3 = 1.1_{(2)} \times 2^{128-127} = 11_{(2)}$. Les nombres $.1875 = 3/16$ et $-3072 = 3 \times 2^{10}$ s'obtiennent par simple ajustement de l'exposant puisqu'ils diffèrent tous d'un facteur de la forme 2^k.

La norme IEEE 754 prévoit des codages pour des nombres flottants spéciaux (on dit « anormaux »). Les plus importants sont plus ou moins l'infini et `NaN` (acronyme pour *Not a Number*). Les deux infinis apparaissent naturellement lorsqu'un flottant atteint une valeur qui dépasse, en valeur absolue, la plus grande valeur représentable avec le codage utilisé. Le flottant `NaN` apparaît lorsqu'on calcule la racine carrée d'un nombre négatif, par exemple. Les opérations arithmétiques sont également définies pour ces flottants anormaux. Le code Python (Python3) donné Figure 1.2, qui s'appuie sur la bibliothèque numpy illustre ces notions. Voici les codages de ces flottants, en simple précision, obtenus avec le programme de la Figure 1.1. Pour `NaN`, la norme impose seulement que la mantisse soit différente de zéro. Il est ainsi possible de coder dans la mantisse, la nature du calcul qui a produit ce résultat.

```
import numpy as np
print ('Plus grand flottant 64 bits =', np.finfo(np.float64).max)
print ('Plus petit flottant 64 bits =', np.finfo(np.float64).min)
print ('Flottant anormal +infini =', 1e200 * 1e200)
print ('Flottant anormal Not-A-Number =', np.sqrt(-1))
```

À l'exécution :

```
Plus grand flottant 64 bits = 1.7976931348623157e+308
Plus petit flottant 64 bits = -1.7976931348623157e+308
Flottant anormal +infini = inf
Flottant anormal Not-A-Number = nan
```

FIGURE 1.2 – Commandes Python affichant le plus grand et le plus petit nombre flottant double précision représentable en machine. La constante `1e200` représente 1×10^{200} en 64 bits.

```
nan
signe = 0, exposant = 255, mantisse (binaire) = 10000000000000000000000
inf
signe = 0, exposant = 255, mantisse (binaire) = 00000000000000000000000
-inf
signe = 1, exposant = 255, mantisse (binaire) = 00000000000000000000000
```

1.2 Notion abstraite d'arithmétique flottante

Un concept clef : l'epsilon machine. La simplicité de la définition masque des idées extrêmement importantes.

Définition 1 *On définit l'epsilon machine, noté* $\varepsilon_{\text{machine}}$*, comme la demi-distance entre le flottant* 1 *et le plus petit flottant supérieur ou égal à* 1*.*

1.2.1 À toute arithmétique flottante, un epsilon machine

En un sens, dans la pratique, $\varepsilon_{\text{machine}}$ n'est rien d'autre qu'une constante qui peut prendre deux valeurs différentes. Voir Figure 1.3.

1.2.2 À tout epsilon machine, une arithmétique flottante

En effet, il suffit de préciser un epsilon machine pour préciser une arithmétique flottante. L'idée, c'est qu'il est possible de spécifier les arithmétiques flottantes (arrondi sur les données, arrondi sur les opérations), en s'appuyant sur l'epsilon machine. On peut ainsi tenir des raisonnements sur les algorithmes numériques sans jamais supposer qu'on calcule en 32 bits, en 64 bits ou n'importe quelle autre précision ! Voir [2, Lecture 13].

```
print ('epsilon machine 32 bits =', np.finfo(np.float32).eps * 5e-1)
print ('epsilon machine 64 bits =', np.finfo(np.float64).eps * 5e-1)
```

À l'exécution, on obtient :

```
epsilon machine 32 bits = 5.960464477539063e-08
epsilon machine 64 bits = 1.1102230246251565e-16
```

FIGURE 1.3 – Calcul de $\varepsilon_{\text{machine}}$ en Python (la valeur `np.finfo(np.float32).eps` vaut le double de $\varepsilon_{\text{machine}}$).

Dans toute la suite du cours, nous supposerons donc qu'on calcule avec une arithmétique flottante vérifiant les propositions 1 et 2.

Proposition 1 *(arrondi sur les données)*
Quel que soit $x \in \mathbb{R}$, il existe un flottant $\tilde{x}$ tel que

$$\frac{|x - \tilde{x}|}{|x|} \leqslant \varepsilon_{\text{machine}} \cdot$$

Dit autrement, quel que soit $x \in \mathbb{R}$, il existe un flottant $\tilde{x}$ et un ε, avec $|\varepsilon| \leqslant \varepsilon_{\text{machine}}$, tels que

$$\tilde{x} \;=\; x\,(1+\varepsilon)\,. \tag{1.1}$$

Supposons que dans un calcul, nous devions utiliser la constante π et que nous la rangions dans une variable. Après affectation, le contenu de la variable ne sera pas π, mais un flottant $\tilde{\pi} = \pi\,(1+\varepsilon)$, pour un certain $\varepsilon \leqslant \varepsilon_{\text{machine}}$.

Dans la proposition suivante, on désigne par $*$ l'une des quatre opérations sur les réels (addition, soustraction, multiplication et division) et $\circledast$ l'opération correspondante sur les flottants. On rappelle que, même si x et y sont deux flottants, $x{*}y$ n'est pas forcément un flottant [1] et donc, même si x et y sont deux flottants, $x{*}y$ et $x \circledast y$ ne sont pas forcément égaux.

Proposition 2 *(arrondi sur le résultat des opérations)*
Pour tous flottants x et y, il existe ε, avec $|\varepsilon| \leqslant \varepsilon_{\text{machine}}$, tel que

$$x \circledast y \;=\; (x * y)\,(1+\varepsilon)\,. \tag{1.2}$$

1. En simple précision, les nombres 2^{100} et 2^{-100} sont des flottants, mais le nombre $2^{100} + 2^{-100}$ ne peut pas être représenté exactement par un flottant.

1.2.3 Grand O d'epsilon machine

En analyse numérique, on écrit parfois que le résultat d'un calcul est égal au résultat théorique, avec une erreur e vérifiant :

$$e \;=\; O(\varepsilon_{\text{machine}})\,. \tag{1.3}$$

La signification du grand O est : il existe une constante $c > 0$ telle que $e \leqslant c\,\varepsilon_{\text{machine}}$.

Si on voit l'epsilon machine comme une constante, l'énoncé ci-dessus semble totalement vide d'information !

La signification de cet énoncé est en fait la suivante : on imagine que le calcul qui a donné l'erreur est effectué sur des arithmétiques flottantes de plus en plus précises et donc pour lesquelles $\varepsilon_{\text{machine}}$ tend vers zéro. L'epsilon machine est donc vu comme une variable. Le véritable énoncé devient alors : il existe une constante $c > 0$ telle que, quel que soit l'epsilon machine (supposé tendre vers zéro), l'erreur $e \leqslant c\,\varepsilon_{\text{machine}}$. Dit autrement,

$$\exists\, c > 0,\ \forall\, \varepsilon_{\text{machine}},\ e \leqslant c\,\varepsilon_{\text{machine}}\,. \tag{1.4}$$

Cet énoncé-là est nettement plus riche en information.

Dans les chapitres suivants, nous étudierons des algorithmes d'algèbre linéaire, qui s'appliquent à des matrices de dimensions m et n. La constante c de la formule (1.4), qui ne dépend évidemment pas de l'epsilon machine, peut dépendre de m et de n. Cela peut se justifier ainsi : changer m et n, c'est changer les dimensions de l'espace de départ et de l'espace d'arrivée du problème (au sens du chapitre 5) à résoudre. C'est donc changer le problème. Voir [2, Lecture 14].

1.3 Erreur relative, erreur absolue

Supposons qu'en cherchant à calculer un réel x, on calcule en fait un flottant $\tilde{x}$. La différence

$$|x - \tilde{x}|$$

est appelée, l'*erreur absolue* faite par le calcul. On la distingue de l'*erreur relative* (qu'on a rencontrée dans la section précédente) :

$$\frac{|x - \tilde{x}|}{|x|}.$$

En analyse numérique, on considère plutôt l'erreur relative mais l'erreur absolue peut être extrêmement importante, lorsqu'on cherche à résoudre des applications concrètes. L'erreur relative donne le nombre de chiffres exacts de la mantisse de $\tilde{x}$:

$$\text{nombre de chiffres exacts} \simeq -\log_{10}\left(\frac{|x - \tilde{x}|}{|x|}\right). \tag{1.5}$$

Par exemple, si on prend $x = 0.001234$ et $\tilde{x} = 0.001233$, la formule ci-dessus donne approximativement 3.09. Si on remplace le logarithme en base 10 par le logarithme en base 2, on obtient le nombre de bits exacts de la mantisse.

1.4 Les erreurs d'arrondis

Les exemples qui suivent sont extraits, pour la plupart, de [1, Lecture 8].

Question 1. Écrire un algorithme aussi précis que possible qui additionne 10^8 fois le nombre 1 (le résultat doit donc valoir 10^8). Utiliser des flottants sur 32 bits (qui ont moins de 8 chiffres significatifs). Quelle est la difficulté ? Comment la résoudre ?

1.4.1 La soustraction

L'exemple donné Figure 1.4 montre que l'ordre des calculs peut être extrêmement important, surtout en présence de soustractions entre deux flottants de valeurs très proches.

```
print ('0.1234 =', np.float32(1e7) + np.float32(25.1234) - np.float32(10000025))
print ('0.1234 =', np.float32(1e7) - np.float32(10000025) + np.float32(25.1234))
```

À l'exécution :

```
0.1234 = 0.0
0.1234 = 0.123399734
```

FIGURE 1.4 – Deux formules équivalentes avec des réels mais pas avec des flottants 32 bits (des exemples équivalents existent en 64 bits).

1.4.2 Le choix de la bonne formule

Le deuxième exemple, Figure 1.5, montre qu'en calcul numérique, il est souvent utile de réécrire les formules pour éviter de méchantes pertes de précision. On cherche à calculer les racines x_1, x_2 de l'équation $a\,x^2 + b\,x + c = 0$. La première méthode consiste à utiliser la formule bien connue. Dans la deuxième méthode, on utilise le fait que $b = -x_1 - x_2$ et que $c = x_1\,x_2$ (parce que $a = 1$). On utilise surtout le fait que la plus grande des deux racines est calculée de façon très précise. Notons x la plus petite des deux racines et $\tilde{x}$ la valeur calculée par la méthode 1. Le nombre de chiffres exacts (1.5) est 1 (seul le chiffre 1 initial est exact).

1.4.3 L'effet papillon

Le troisième exemple, Figure 1.6, montre qu'une toute petite erreur d'arrondi à une étape peut causer une erreur importante plus tard. Le programme suivant calcule les 80 premiers termes de la suite définie par

$$x_0 = \frac{1}{3}\,, \quad x_1 = \frac{1}{12}\,, \quad x_{n+2} = \frac{9}{4}\,x_{n+1} - \frac{1}{2}\,x_n\,. \tag{1.6}$$

```
x1 = 1.23456e8
x2 = 1.3579177e-10
print ('vraies racines =', x1, x2)
a = 1e0
b = - x1 - x2
c = x1 * x2
delta = b**2 - 4*a*c
x1 = (-b + np.sqrt(delta))/(2*a)
x2 = (-b - np.sqrt(delta))/(2*a)
print ('racines calculées par la méthode 1 =', x1, x2)
x2 = c / x1
print ('racines calculées par la méthode 2 =', x1, x2)
```

À l'exécution :

```
vraies racines = 123456000.0 1.3579177e-10
racines calculées par la méthode 1 = 123456000.0 0.0
racines calculées par la méthode 2 = 123456000.0 1.3579177e-10
```

FIGURE 1.5 – Deux façons différentes de calculer les racines d'un polynôme de degré 2. Ces formules sont équivalentes avec des réels. Elles ne sont pas équivalentes avec des flottants, lorsque l'une des racines est beaucoup plus grande que l'autre, en valeur absolue.

La solution exacte, $x_n = (1/3)\,(1/4)^n$, tend vers zéro. Le graphique de la Figure 1.7 montre que dans un premier temps, x_n tend vers zéro mais qu'un phénomène étrange se produit un peu avant $k = 20$. L'explication est la suivante : si on oublie les conditions initiales x_0 et x_1, la solution de l'équation de récurrence est de la forme :

$$x_n = a\,2^n + b\,\left(\frac{1}{4}\right)^n .$$

Les conditions initiales ont été choisies de telle sorte que $a = 0$. Toutefois, en raison des erreurs d'arrondi, a est différent de zéro, ce qui implique que le terme en 2^n doit finir par dominer le terme en $(1/4)^n$. Plus précisément, la proposition 1 implique que $a \simeq \varepsilon_{\text{machine}}$, qui vaut approximativement 10^{-16} dans notre cas. En résolvant l'équation

$$\varepsilon_{\text{machine}}\,2^n - \frac{1}{3}\,\left(\frac{1}{4}\right)^n = 0\,,$$

on trouve que le terme en 2^n domine le terme en $(1/4)^n$ pour $n \geqslant 18$, ce que confirme le graphique.

Si on reproduit le même exemple avec $x_0 = 4$ et $x_1 = 1$, deux conditions initiales qui imposent $a = 0$ aussi, mais qui sont représentables exactement comme des flottants, on observe que la suite tend bien numériquement vers zéro.

```
x0 = 4
x1 = 1
for i in range (80) :
    print (i, x0)
    x0, x1 = x1, 2.25*x1 - 5e-1*x0
```

FIGURE 1.6 – Ce programme calcule les 80 premiers termes d'une suite définie par récurrence et condition initiale. Le comportement de ce programme est extrêmement sensible aux erreurs d'arrondis sur les *conditions initiales*.

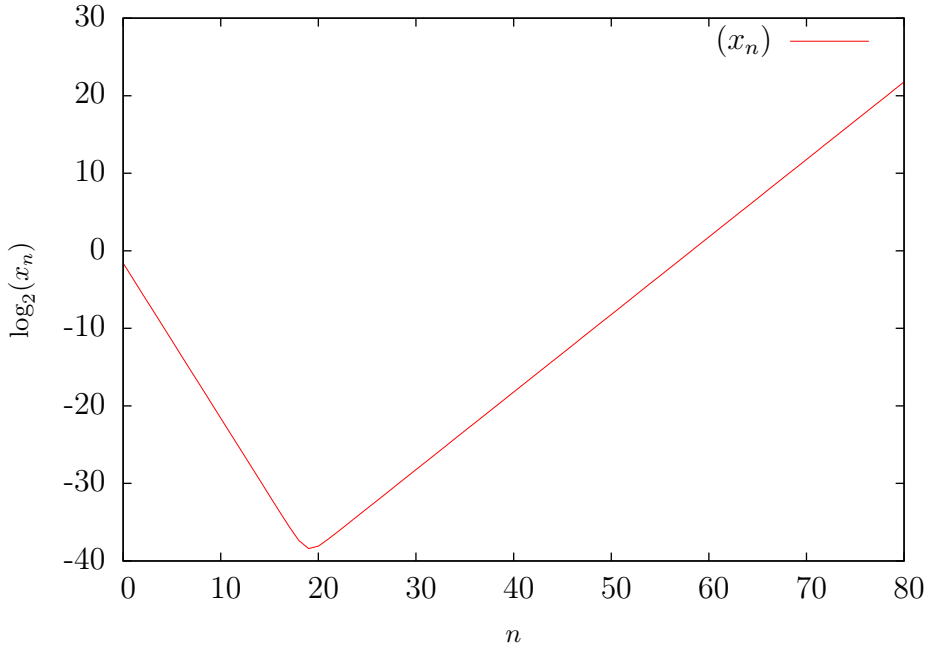


FIGURE 1.7 – La solution calculée de l'équation (1.6).

1.5 Conclusion

En calcul numérique, les nombres employés sont à virgule flottante. Parmi eux, on trouve des flottants normaux et des flottants anormaux. Un langage classique est FORTRAN. En analyse numérique, on abstrait la notion de nombre à virgule flottante pour aboutir à la notion d'arithmétique flottante, dont les spécifications dépendent d'un paramètre : l'epsilon machine. Un problème important est le contrôle des erreurs d'arrondis, qui peuvent se manifester de différentes façons [1, Lecture 8, Envoi] :

— Elles peuvent se propager au cours de sommes très longues (question 1). Ce problème peut parfois se résoudre en réécrivant le code avec soin.

— Elles peuvent apparaître lorsqu'une soustraction fait disparaître les chiffres les plus significatifs des nombres soustraits. Parfois, il est possible de corriger le problème en

retravaillant les formules (exemple du calcul des racines d'une équation de degré 2, dans le cas où elles sont d'ordres de grandeur très différents).
Parfois, la difficulté est due au problème mathématique qu'on cherche à résoudre (on pourrait construire un exemple en cherchant à calculer les racines d'une équation de degré 2, dans le cas où elles sont *très proches* l'une de l'autre). Dans ce cas, il n'y a pas de solution simple. Cette difficulté est mesurée par la notion de conditionnement, abordée dans le cours suivant.

— Elles peuvent être amplifiées par un algorithme (exemple de la suite définie par récurrence). Dans ce cas non plus, pas de solution simple.

Bibliographie

[1] Gilbert W. Stewart. *Afternotes on Numerical Analysis*. SIAM, 1996.

[2] Lloyd Nicholas Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

Chapitre 2

Matrices et vecteurs

Les matrices sont désignées par des majuscules, les vecteurs par des minuscules et les scalaires (en général, des réels), par des lettres grecques ou latines.

En général, la matrice considérée est appelée A. On note m son nombre de lignes et n son nombre de colonnes. Dans le cas où A est supposée carrée, on note $m \times m$ sa dimension. L'élément de A, ligne i, colonne j, est notée a_{ij}. La $i$ème colonne de A est un vecteur colonne de dimension n, noté a_i.

Tous les vecteurs sont des vecteurs colonne. Lorsque nous aurons besoin de considérer un vecteur ligne, on le désignera comme la transposée, x^T d'un vecteur colonne x.

2.1 Opérations sur les matrices

La multiplication par un scalaire est commutative : $\mu\, A = A\, \mu$.

Si A et B ont mêmes dimensions $m \times n$, leur somme $A + B$ est aussi de dimension $m \times n$ et on a $A + B = (a_{ij} + b_{ij})$.

Si A est de dimension $m \times n$ et B de dimension $n \times p$, alors le produit $A\, B$ est défini. La matrice $A\, B$ est de dimension $m \times p$ et on a :

$$A\, B = \left(\sum_{k=1}^{n} a_{ik}\, b_{kj} \right) .$$

Le produit de matrices n'est pas commutatif, c'est-à-dire qu'en général, $A\, B \neq B\, A$. Par contre, il est associatif : $(A\, B)\, C = A\, (B\, C)$.

Question 2. Mettre le système d'équations linéaires suivant sous la forme $A\, x = b$. Quelles relations y a-t-il entre le nombre d'équations, le nombre d'inconnues et les dimensions de la matrice A et des vecteurs x et b ?

$$2\, x_1 + x_3 + 6\, x_4 = 4\,, \quad x_1 + x_2 - x_3 = 5\,, \quad 5\, x_1 + x_2 + 6\, x_3 - x_4 = 0\,.$$

En Python, le paquetage `numpy` de Python fournit quelques outils de calcul matriciel. Le sous-paquetage `linalg` de `scipy` fournit des outils d'algèbre linéaire numérique.

```
>>> import numpy as np
```

La documentation de `numpy` recommande d'utiliser `array` à deux dimensions (et plus `matrix`) pour les matrices.

```
>>> A = np.array ([[1,2,3],[4,5,6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])

>>> B = np.transpose (A)
>>> B
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Le produit *par* un scalaire

```
>>> 3 * B
array([[ 3, 12],
       [ 6, 15],
       [ 9, 18]])
```

Le produit scalaire s'appelle « dot product » en Anglais. La fonction `dot` permet de réaliser des produits de matrices (qui n'est rien d'autre qu'une succession de produits scalaires). On vérifie ci-dessous que le produit de matrices n'est pas commutatif.

```
>>> np.dot (A, B)
array([[14, 32],
       [32, 77]])

>>> np.dot (B, A)
array([[17, 22, 27],
       [22, 29, 36],
       [27, 36, 45]])
```

Les vecteurs sont représentés par des `array` à une dimension. L'information « vecteur ligne » ou « vecteur colonne » n'est pas stockée.

```
>>> v = np.array([10,-10,20])
>>> v
array([ 10, -10,  20])
```

Suivant les contextes, le même `array` peut être interprété comme l'un ou comme l'autre. Ici, `v` est interprété comme un vecteur colonne. La fonction `dot` permet de calculer le produit matrice vecteur.

```
>>> np.dot (A,v)
array([ 50, 110])
```

Ici, `v` est interprété comme un vecteur ligne.

```
>>> np.dot (v,B)
array([ 50, 110])
```

La *transposée* d'une matrice A de dimension $m \times n$ est une matrice A^T de dimension $n \times m$, définie par $A^t = (a_{ji})$. Une matrice égale à sa transposée est une matrice *symétrique*. La transposition interagit avec les autres opérations matricielles de la façon suivante (la formule pour le produit est à rapprocher de la formule pour l'inverse) :

$$(\mu\, A)^T = \mu\, A^T\,, \qquad (A + B)^T = A^T + B^T\,, \qquad (A\, B)^T = B^T\, A^T\,.$$

Question 3. [Utile pour la décomposition $L\,U$]. Soit A une matrice 3×5. Chacune des opérations suivantes peut être obtenue en multipliant A, à gauche, par une certaine matrice. Quelles sont ces matrices ?

1. Diviser la première ligne par 2.
2. Ajouter 4 fois le troisième ligne à la deuxième.
3. Permuter les lignes 1 et 2.

Question 4. [Utile pour la méthode de Cholesky et le chapitre 7]. Montrer que le produit $A\, A^T$ d'une matrice par sa transposée est toujours défini (que suffit-il de vérifier ?). Montrer, en utilisant la formule pour la transposée d'un produit, que $A\, A^T$ est symétrique.

2.2 Partitionnement de matrices

Partitionner une matrice consiste à la découper en sous-matrices. Voici un exemple :

$$A = \left(\begin{array}{c|cc} a_{11} & a_{12} & a_{13} \\ \hline a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

avec

$$A_{11} = a_{11}\,, \quad A_{21} = \begin{pmatrix} a_{21} \\ a_{31} \end{pmatrix}, \quad A_{12} = \begin{pmatrix} a_{12} & a_{13} \end{pmatrix}, \quad A_{22} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}.$$

Si B est une matrice de même dimension que A et partitionnée exactement de la même façon, alors la somme $A + B$ peut se faire bloc par bloc :

$$A + B = \begin{pmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{pmatrix}$$

De même, le produit $A\,B$ peut aussi se formuler bloc par bloc lorsque la dimension de B et les partitionnements de A et de B le permettent.

Question 5. [Utile pour la méthode de Cholesky]. On considère la matrice A suivante. Donner le partitionnement de A^T en fonction de celui de A (exprimer A^T à partir du scalaire a_{11}, du vecteur x et de la matrice A_{22}). Après avoir vérifié que le partitionnement le permet, formuler le produit $A\,A^T$ par blocs.

$$A = \left(\begin{array}{c|cc} a_{11} & 0 & 0 \\ \hline a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array}\right) = \begin{pmatrix} a_{11} & 0 \\ x & A_{22} \end{pmatrix}$$

2.3 Rang, inversibilité

Un ensemble de vecteurs $x_1, \ldots, x_r$ est *linéairement indépendant*, si $\lambda_1\,x_1 + \cdots + \lambda_r\,x_r = 0$ implique $\lambda_1 = \cdots = \lambda_r = 0$.

Le *rang* d'une matrice A est la dimension de l'espace vectoriel engendré par les colonnes de A. En d'autres termes, c'est le nombre maximal de colonnes de A linéairement indépendantes. Le rang d'une matrice est égal au rang de sa transposée. Le rang d'une matrice A de dimension $m \times n$ est donc inférieur ou égal à m et à n.

Une matrice A de dimension $m \times n$ est dite de *rang maximal* si son rang est le plus grand rang possible pour une matrice $m \times n$. Par exemple, si $m \geqslant n$, la matrice A est de rang maximal si elle est de rang n ; si A est carrée, A est de rang maximal si et seulement si elle est inversible (voir cette notion ci-dessous). La notion de rang maximal peut être vue comme une sorte de généralisation, pour les matrices rectangulaires, de la notion d'inversibilité, qui n'a de sens que pour les matrices carrées. On s'en servira aux chapitres 6 et 7.

Une matrice A est *inversible* si elle est carrée, et s'il existe une matrice carrée B telle que $A\,B$ soit la matrice identité. La matrice B, appelée *inverse* de A, est notée A^{-1}. L'inverse d'une matrice A est un inverse à la fois à droite et à gauche : $A\,A^{-1} = A^{-1}\,A =$ l'identité. L'inverse d'un produit de matrices vérifie :

$$(A\,B)^{-1} = B^{-1}\,A^{-1}\,.$$

Le théorème suivant recense différentes caractérisations des matrices inversibles.

Théorème 1 *Soit A une matrice carrée de dimension $m \times m$. Les conditions suivantes sont équivalentes :*

1. *A est inversible,*
2. *A est de rang m,*
3. *les colonnes de A (de même que les lignes) sont linéairement indépendantes,*
4. *si $A\,x = 0$ alors $x = 0$ (le noyau de A vaut $\{0\}$),*

5. *quel que soit b, le système $A\,x = b$ a une unique solution,*
6. *les valeurs propres de A sont non nulles,*
7. *les valeurs singulières de A sont non nulles,*
8. *le déterminant de A est non nul.*

Le déterminant est une notion peu utile en pratique, en calcul numérique. En effet, le déterminant de $\mu\,A$ est égal à μ^m fois le déterminant de A. Ainsi, si $m = 30$ et $\mu = 10^{-1}$, alors le déterminant de $\mu\,A$ est égal au déterminant de A, fois 10^{-30}. Sur de petites matrices, il est possible de vérifier le théorème. On charge non seulement `numpy` mais aussi `scipy.linalg`.

```
>>> import numpy as np
>>> import scipy.linalg as nla
>>> A = np.array([[1,3],[2,-1]])
>>> A
array([[ 1,  3],
       [ 2, -1]])
```

La matrice A est inversible.

```
>>> nla.inv(A)
array([[ 0.14285714,  0.42857143],
       [ 0.28571429, -0.14285714]])
```

Numériquement, le calcul du rang d'une matrice est une opération qui demande du soin. Une des meilleures méthodes connues consiste à calculer ses valeurs singulières et vérifier qu'elles sont non nulles ou, plus précisément, suffisamment éloignées de zéro. C'est le cas ici.

```
>>> nla.svdvals(A)
array([3.1925824, 2.1925824])
```

Comme une matrice réelle peut avoir des valeurs propres complexes, celles-ci sont retournées sous la forme de complexes.

```
>>> nla.eigvals(A)
array([ 2.64575131+0.j, -2.64575131+0.j])
```

Mais comme les parties imaginaires sont nulles, on peut souhaiter en extraire les parties réelles.

```
>>> np.real(nla.eigvals(A))
array([ 2.64575131, -2.64575131])
```

Enfin sur une si petite matrice, il n'y a aucune difficulté à calculer le déterminant.

```
>>> nla.det(A)
-7.0
```

Question 6. Vérifier le théorème en prenant pour exemple une matrice carrée singulière (non inversible).

Question 7. [Utile au chapitre 7]. Soit A une matrice rectangulaire, plus haute que large ($m \geqslant n$). Vérifier que, si x est un vecteur non nul et $A\,x = 0$ alors les colonnes de A sont linéairement dépendantes (A est de rang maximal). Supposons maintenant que le rang de A soit n (c'est-à-dire que A est de rang maximal) et que le produit $A\,x = 0$. Est-il possible que x soit non nul ?

2.4 Produit scalaire, produit tensoriel

Soient x et y deux vecteurs de même dimension, m. Le *produit scalaire* de x et de y est le réel $x^T\,y$. La norme euclidienne (norme 2) d'un vecteur peut être définie grâce au produit scalaire :

$$\|x\|_2 = \sqrt{x^T\,x}\,.$$

Le cosinus de l'angle α entre deux vecteurs x et y aussi :

$$\cos\alpha = \frac{x^T\,y}{\|x\|_2\,\|y\|_2}.$$

Soient x et y deux vecteurs de dimensions m et n. Le *produit tensoriel* de x et de y est la matrice $x\,y^T$. Cette matrice est de dimension $m \times n$. Elle est de rang 1. On peut montrer que toute matrice de rang 1 est le produit tensoriel de deux vecteurs [1, chapter 9].

Question 8. On étudiera au chapitre 8 des algorithmes qui calculent des vecteurs v_i censés devenir de plus en plus colinéaires à un vecteur x. On aimerait calculer un nombre dépendant de v_i et de x, qui vaille 1 si v_i et x sont orthogonaux et 0 si v_i et x alignés. Quelle formule utiliser ?

Question 9. [Utile à beaucoup de démonstrations]. Montrer que, si x est un vecteur non nul, alors $x^T\,x$ est un réel strictement positif.

Question 10. [Utile au chapitre 7]. On considère un partitionnement d'un vecteur x comme ci-dessous. Exprimer le produit scalaire $x^T\,x$ en fonction de x_1 et de x_2.

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

Les matrices de rang 1 apparaissent souvent en calcul numérique. Si on représente une matrice $A = x\,y^T$ de rang 1 par les deux vecteurs x et y, on peut se demander comment effectuer, mettons, la multiplication de A par un vecteur z. Une réponse élégante est fournie par le raisonnement suivant, qui met en œuvre certaines des notions rappelées précédemment :

$$\begin{aligned} A\,z &= (x\,y^T)\,z \\ &= x\,(y^T\,z) \quad \text{car le produit est associatif} \\ &= (y^T\,z)\,x \quad \text{car } y^T\,z \text{ est un scalaire.} \end{aligned}$$

Pour calculer $A\,z$, il suffit donc de calculer un produit scalaire, puis la multiplication d'un vecteur par un scalaire. Voir [1, Lecture 9].

Question 11. Déterminer la complexité des deux façons différentes de calculer le produit $A\,z$. Voir éventuellement la section suivante pour quelques indications sur cette notion.

2.5 Normes

On sera amené à étudier les effets d'une perturbation sur une matrice ou un vecteur. Il est impossible de mesurer l'ordre de grandeur de ces perturbations en considérant la matrice ou le vecteur, coefficient par coefficient (il y en a trop). Les *normes* permettent de résumer tous ces nombres en un seul.

Il y a des normes pour les vecteurs et des normes pour les matrices. On peut passer des unes aux autres mais ... attention à la marche !

Une norme vectorielle est une fonction $\mathbb{R}^m \to \mathbb{R}$ qui satisfait :

1. $\|x\| \neq 0$ pour tout vecteur $x \neq 0$;
2. $\|\alpha\,x\| = |\alpha|\,\|x\|$;
3. $\|x + y\| \leqslant \|x\| + \|y\|$ (inégalité triangulaire).

Les normes les plus utilisées en pratique sont les suivantes. La norme 2, appelée aussi *norme euclidienne*, est la plus connue.

$$\|x\|_1 = \sum_{i=1}^{m} |x_i|\,, \quad \|x\|_2 = \sqrt{\sum_{i=1}^{m} x_i^2}\,, \quad \|x\|_\infty = \max_{i=1}^{m} |x_i|\,.$$

Une norme matricielle est une fonction $\mathbb{R}^{m\times n} \to \mathbb{R}$ qui satisfait les conditions suivantes. Les trois premières sont analogues aux conditions pour les normes vectorielles.

1. $\|A\| \neq 0$ pour toute matrice $A \neq 0$;
2. $\|\alpha\,A\| = |\alpha|\,\|A\|$;
3. $\|A + B\| \leqslant \|A\| + \|B\|$ (inégalité triangulaire) ;
4. $\|A\,B\| \leqslant \|A\|\,\|B\|$.

Enfin, une norme vectorielle $\|\cdot\|_{\mathrm{v}}$ est dite *consistante avec une norme matricielle* $\|\cdot\|_{\mathrm{M}}$, si elle vérifie :

$$\|A\,x\|_{\mathrm{M}} \leqslant \|A\|_{\mathrm{M}}\,\|x\|_{\mathrm{v}}\,.$$

La consistance est une propriété qui semble très naturelle et souhaitable. Malheureusement, elle nous empêche de généraliser les normes vectorielles de la « façon évidente ».

Question 12. Supposons qu'on définisse la norme infinie d'une matrice A comme le maximum des valeurs absolues de ses éléments a_{ij} (généralisation « évidente » de la norme infinie des vecteurs). Dans le cas de la matrice suivante, que vaudraient $\|A^2\|$ et $\|A\|^2$? Que peut–on en déduire ?

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\,.$$

Voici les généralisations, pour les matrices, des trois normes vectorielles données précédemment. Attention : la norme matricielle qui généralise la norme 2 vectorielle change de nom [1] : on l'appelle la norme de *Frobenius*.

$$\|A\|_1 = \max_{j=1}^{m} \sum_{i=1}^{m} |a_{ij}|\,, \quad \|A\|_F = \sqrt{\sum_{i,j\leqslant m} a_{ij}^2}\,, \quad \|A\|_\infty = \max_{i=1}^{m} \sum_{j=1}^{m} |a_{ij}|\,.$$

1. Il existe une norme 2 pour les matrices, qui n'est pas la norme de Frobenius, qui a une utilité importante. On ne la présente pas ici.

Bibliographie

[1] Gilbert W. Stewart. *Afternotes on Numerical Analysis*. SIAM, 1996.

Chapitre 3

Systèmes linéaires triangulaires

On s'intéresse à la résolution numérique d'un système d'équations linéaires

$$A\,x = b\,,$$

où A est une matrice triangulaire inférieure (*lower triangular* en Anglais), c'est-à-dire une matrice carrée de la forme suivante. Les éléments diagonaux a_{ii} sont non nuls. Les éléments au-dessus de la diagonale sont nuls.

$$A = \begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{pmatrix}.$$

Pour faire court, on appellera parfois de telles matrices des matrices L. La plupart des considérations faites dans ce chapitre s'appliquent aussi aux matrices triangulaires supérieures (*upper triangular* en Anglais) qui sont des transposées de matrices L. Pour faire court, ces matrices seront parfois appelées des matrices U.

La résolution de systèmes triangulaires inférieurs se fait du haut vers le bas (on parle d'algorithmes de descente). La résolution de systèmes triangulaires supérieurs se fait du bas vers le haut (on parle d'algorithmes de remontée).

Question 13. [Utile au chapitre 7]. Montrer que la matrice A est inversible.

3.1 Substitutions avant et arrière

On dispose de deux algorithmes naturels pour implanter une descente : par substitution *avant* ou *arrière* (*forward* substitution ou *back* substitution en Anglais). Voir la section 3.2 pour des implantations en FORTRAN. Dans les deux cas, les paramètres sont m (un entier), la matrice A, de dimension $m \times m$ et le vecteur b, de dimension m. Le vecteur x n'est pas passé en paramètre. Après appel à l'une de ces fonctions, la solution se trouve dans b.

Dans le cas de la substitution avant, une fois la valeur de x_i calculée et stockée dans b_i, l'algorithme passe immédiatement à la ligne suivante, pour calculer x_{i+1}. Il lui reste donc à résoudre un système de $m - i$ équations (lignes) en m inconnues (colonnes).

Dans le cas de la substitution arrière, une fois la valeur de x_i calculée et stockée dans b_i, l'algorithme commence par reporter cette valeur dans les coordonnées restantes du vecteur b, avant de passer à la ligne suivante. Il lui reste donc à résoudre un système de $m-i$ équations en $m-i$ inconnues (les valeurs de $x_1, \ldots, x_i$ ayant été reportées dans le second membre, on peut oublier les colonnes qui leur correspondent).

Ces deux algorithmes sont *backward stable* (voir chapitre 5).

3.2 Implantations

3.2.1 En FORTRAN

Les codes ci-dessous sont codent les substitutions avant et arrière dans le cas d'une descente. Deux variantes peuvent aussi être écrites pour le cas d'une remontée. La ligne qui commence par une étoile est un commentaire qui n'est utile que dans le cadre d'une utilisation avec l'utilitaire `f2py3` (voir section 3.2.3). En sortie, le vecteur b contient la solution du système.

```
      SUBROUTINE FORWARD_L (M, A, B)
* f2py3 intent(inplace) B
      INTEGER M
      DOUBLE PRECISION A (M, M)
      DOUBLE PRECISION B (M)
      INTEGER I, J
      DO I = 1,M
         DO J = 1,I-1
            B(I) = B(I) - A(I,J)*B(J)
         END DO
         B(I) = B(I)/A(I,I)
      END DO
      END
```

```
      SUBROUTINE BACKWARD_L (M, A, B)
* f2py3 intent(inplace) B
      INTEGER M
      DOUBLE PRECISION A (M, M)
      DOUBLE PRECISION B (M)
      INTEGER I, J
      DO J = 1,M
         B(J) = B(J)/A(J,J)
         DO I = J+1,M
            B(I) = B(I) - A(I,J)*B(J)
         END DO
      END DO
      END
```

3.2.2 En Python

Le paquetage `scipy.linalg` offre la fonction `solve_triangular` qui comporte de nombreux paramètres.

Le paramètre `lower=True` indique à la fonction d'utiliser les données présentes dans la partie inférieure (la sous-matrice L) de la matrice A passée en paramètre. Si on ne précise rien de plus, ce paramètre indique qu'on souhaite exécuter une descente mais, si on passe, en plus `trans='T'` alors la fonction exécute une remontée avec, pour matrice U, la transposée de L. Cette fonctionnalité est utile en conjonction avec l'algorithme de Cholesky.

```
>>> import numpy as np
>>> from scipy.linalg import solve_triangular
>>> A = np.array([[3, 0, 0, 0], [2, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 1]])
>>> b = np.array([4, 2, 4, 2])
>>> x = solve_triangular(A,b,lower=True)
>>> x
array([ 1.33333333, -0.66666667,  2.66666667, -1.33333333])
>>> np.dot (A,x)
array([4., 2., 4., 2.])
```

On peut aussi souhaiter que Python appelle des implantations efficaces (voir section 3.5) programmées en FORTRAN et modifie le vecteur b plutôt que de retourner un nouveau vecteur x comme ci-dessus. Pour cela, il faut que les données à traiter soient des flottants sur 64 bits (le type `DOUBLE PRECISION` du langage FORTRAN) et aussi que la matrice A soit stockée en mémoire colonne par colonne (comme en FORTRAN) et non ligne par ligne (comme en C). Il suffit de spécifier pour cela le paramètre `order='F'`. On aboutit au code suivant.

```
>>> import numpy as np
>>> from scipy.linalg import solve_triangular
>>> A = np.array([[3, 0, 0, 0], [2, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 1]], \
                 dtype=np.float64, order='F')
>>> b = np.array([4, 2, 4, 2], dtype=np.float64)
>>> solve_triangular(A,b,lower=True,overwrite_b=True)
>>> b
array([ 1.33333333, -0.66666667,  2.66666667, -1.33333333])
```

3.2.3 Appeler du code FORTRAN depuis Python

On montre rapidement dans cette section qu'il est possible d'appeler du code FORTRAN depuis Python. L'utilitaire à utiliser d'appelle `f2py3`. La plus grande difficulté consiste à mettre au point les commandes de compilation, qui peuvent fortement varier d'une installation de machine à une autre.

Supposons que les deux subroutines FORTRAN soient présentes dans des fichiers nommés `backward_l.f` et `forward_l.f`. Le `Makefile` donné Figure 3.1 les compile et permet de les appeler, depuis Python, par l'intermédiaire d'un paquetage nommé `mon_code_fortran`. L'utilitaire `f2py3` a analysé les codes FORTRAN et fait en sorte que la dimension m du système linéaire soit automatiquement extraite des structures de données Python.

```
>>> import numpy as np
>>> import mon_code_fortran as mcf
>>> A = np.array([[3, 0, 0, 0], [2, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 1]], \
...                  dtype=np.float64, order='F')
>>> b = np.array([4, 2, 4, 2], dtype=np.float64)
>>> mcf.backward_l(A,b)
>>> b
array([ 1.33333333, -0.66666667,  2.66666667, -1.33333333])
```

3.3 Complexité

Les algorithmes pour la substitution avant et la substitution arrière ont-ils la même complexité ? Dans le support du cours *Structures de Données*, il est écrit que la complexité d'un algorithme est une fonction f qui, à un entier m représentant la taille de la donnée, associe un nombre $f(m)$ d'opérations, considéré comme représentatif du comportement de l'algorithme.

Dans le cas qui nous intéresse, le paramètre m est la dimension du système[1] et le nombre $f(m)$, le nombre d'opérations en virgule flottante effectuées par l'algorithme.

1. On remarque que la taille de la donnée n'est pas de l'ordre de m mais de m^2 ! C'est la raison pour laquelle la phrase du polycopié de *Structures de Données* est rédigée de façon un peu vague et qu'on y écrit que m *représente* la taille de la donnée. D'autres auteurs écrivent que m est une *mesure naturelle* de la taille de la donnée.

Bien que les calculs de complexité soient faciles à mener à la main, on montre comment les effectuer avec un logiciel de calcul symbolique tel que MAPLE. On s'intéresse à la substitution avant. On commence par calculer le nombre $s(i)$ égal au nombre d'opérations flottantes de la boucle intérieure puis on se sert du résultat pour calculer $f(m)$.

```
> si := sum (2, j=1..i-1);
                                 si := 2 i - 2

> fm := sum (si, i = 1 .. m);
                                          2
                            fm := (m + 1)  - 3 m - 1

# Plus simplement :
> fm := expand (fm);
                                           2
                                  f(m) := m  - m
```

On s'intéresse maintenant à la substitution arrière. On procède de façon similaire.

```
> sj := sum (2, i = j+1 .. m);
                                sj := 2 m - 2 j

> fm := sum (sj, j = 1 .. m);
                                                  2
                     fm := 2 m (m + 1) - (m + 1)  - m + 1

> expand (fm);
                                      2
                                     m  - m
```

On constate que les deux algorithmes ont même complexité.

3.4 Temps d'exécution réels

Sur ma machine, j'ai écrit un code FORTRAN qui exécute les deux sous-programmes sur des matrices aléatoires de dimension croissante. Les résultats sont donnés Figure 3.2.

Dans le cas de la substitution avant, la boucle intérieure modifie l'indice de colonne j. Entre deux exécutions successives de l'instruction

```
B(I) = B(I) - A(I,J)*B(J)
```

l'indice de colonne a été modifié or, les éléments a_{ij} et $a_{i(j+1)}$ sont distants, en mémoire, de m fois la taille d'un flottant. Dans le cas de la substitution arrière, la boucle intérieure modifie l'indice de ligne i. Entre deux exécutions successives de cette même instruction, c'est donc l'indice de ligne qui est modifié or, les éléments a_{ij} et $a_{(i+1)j}$ se suivent, dans la mémoire de l'ordinateur.

Dans un ordinateur courant de 2012, l'accès du processeur à la mémoire est un mécanisme assez compliqué. Pour accélérer les accès, l'ordinateur utilise un mécanisme de *mémoire cache* qui pénalise les accès à des zones distantes les unes des autres en mémoire et avantage les accès à des zones proches les unes des autres. Ainsi, passée une certaine dimension, la substitution avant provoque un *défaut de cache*, qui la rend moins performante que la substitution arrière, en FORTRAN.

Question 14. En langage C, les tableaux à deux dimensions sont rangés en mémoire par ligne. Si on implante les deux algorithmes en C plutôt qu'en FORTRAN, quel phénomène devrait-on observer ? Vérifiez !

De ce qui précède, on peut tirer plusieurs conclusions :

— la complexité d'un algorithme est une mesure théorique d'efficacité des algorithmes. Il est difficile de s'en servir pour prédire le temps de calcul d'un programme. En calcul numérique, elle prédit surtout l'accroissement [2] du temps d'exécution d'un programme en fonction de l'accroissement de la taille de sa donnée. Par exemple, si un algorithme en m^2 prend s secondes et qu'on double m, il faut s'attendre à un temps d'exécution de l'ordre de $4\,s$ secondes ;

— entre deux algorithmes de même complexité, il est très difficile de déterminer le plus rapide. L'exemple précédent montre que la question n'a d'ailleurs pas vraiment de sens, puisque la rapidité peut dépendre du langage de programmation utilisé pour coder l'algorithme !

— réciproquement, il faut se méfier des analyses d'algorithmes fondées sur des comparaisons de temps de calcul. Comparer des temps de calcul, c'est comparer des implantations. La complexité d'un algorithme, c'est ce qui reste quand on a oublié tous les « détails ». C'est la partie de l'analyse qui ne sera pas périmée au prochain changement de machine, ou de langage.

3.5 Complément sur les BLAS

On peut tirer un dernier enseignement des analyses précédentes, très concret : plutôt que de coder les algorithmes de calcul numérique « au niveau des scalaires », comme nous l'avons fait, il vaut mieux les coder « au niveau des vecteurs et des matrices ».

Le langage FORTRAN est muni d'une bibliothèque standard contenant des implantations des opérations élémentaires d'algèbre linéaire (*Basic Linear Algebra Subroutines*, ou « BLAS » en Anglais), qui permettent (au moins partiellement) de faire abstraction de détails d'implantation liés à l'organisation de la mémoire. Par exemple, une boucle telle que

```
DO I = J+1,M
   B(I) = B(I) - B(J)*A(I,J)
END DO
```

2. C'est déjà beaucoup. Dans d'autres domaines, où les données élémentaires sont plus compliquées que des flottants de taille fixe, elle ne prédit même pas cela.

peut s'interpréter en termes d'opérations sur des vecteurs. Notons $\hat{b}$ et $\hat{a}_j$ les vecteurs colonne, de dimension $m-j$, formés des éléments $b_{j+1}, \ldots, b_m$ et $a_{(j+1)j}, \ldots, a_{mj}$. La boucle affecte à $\hat{b}$ le vecteur $\hat{b} - b_j \hat{a}_j$. La fonction BLAS correspondant à cette opération se nomme `AXPY` (son nom vient de l'opération $y = \alpha x + y$ qu'elle réalise). Comme les données sont des flottants double précision, on préfixe son nom avec la lettre « `D` », ce qui donne `DAXPY`. La boucle ci-dessus se réécrit :

```
CALL DAXPY (M-J, - B(J), A(J+1,J), 1, B(J+1), 1)
```

Le premier paramètre donne la dimension, $m-j$, des deux vecteurs. Le deuxième paramètre est le scalaire $\alpha = -b_j$. Le troisième paramètre est le premier élément du vecteur $x = \hat{a}_j$ (en fait, c'est l'adresse de ce premier élément qui est passée en paramètre). Le quatrième paramètre est l'incrément pour x. Ici, il vaut 1, ce qui signifie que chaque élément du vecteur x s'obtient en ajoutant 1 à l'adresse de son prédecesseur. Un incrément de 1 correspond donc à un vecteur colonne. Pour un vecteur ligne, l'incrément aurait été de m. Les deux derniers paramètres sont l'adresse du premier élément de $y = \hat{b}$ et l'incrément associé à ce vecteur.

Question 15. Réécrire la fonction FORTRAN qui implante la substitution avant avec les BLAS, sachant que l'appel de fonction BLAS suivant retourne le produit scalaire $x^T y$ de deux vecteurs de dimension n,

```
* Les fonctions appelées (leur types de retour) doivent être déclarées
      DOUBLE PRECISION DDOT, RESULT
      RESULT = DDOT (N, X, INCX, Y, INCY)
```

Question 16. [Utile pour la décomposition LU]. Comment modifier les deux sous-programmes pour résoudre des systèmes $A\,x = b$ avec A triangulaire supérieure ?

La fonction BLAS `DTRSV` résout les systèmes d'équations linéaires triangulaires.

```
# Construit un module python3 appelé 'mon_code_fortran.py'
# à partir de fichiers FORTRAN.
#
# Sont fabriqués : un fichier source '.py' et un fichier compilé '.so'

# Le nom du module
MODULE=mon_code_fortran

# Les fichiers FORTRAN
FORTRAN_FILES=forward_l.f backward_l.f

# L'utilitaire de compilation
F2PY3=f2py3
# CPPFLAGS et CFLAGS évitent des warnings dus à des codes C mal écrits.
F2PY3_CPPFLAGS=-DNPY_NO_DEPRECATED_API=NPY_1_7_API_VERSION
F2PY3_CFLAGS=-Wno-unused-function -Wno-misleading-indentation
# LDFLAGS et LIBS permettent d'appeler BLAS et LAPACK depuis FORTRAN
F2PY3_LDFLAGS=-L/usr/lib/x86_64-linux-gnu/lapack -L/usr/lib/x86_64-linux-gnu/blas
F2PY3_LIBS=-llapack -lblas
# Autorise le remplacement de l'ancien '.py'
F2PY3_H_FLAGS=--overwrite-signature

.PHONY: all clean

# f2py3 -c construit le '.so'
# f2py3 -h construit le '.py'
all:
    CPPFLAGS="$(F2PY3_CPPFLAGS)" CFLAGS="$(F2PY3_CFLAGS)" $(F2PY3) -c -m $(MODULE) \
            $(F2PY3_LDFLAGS) $(FORTRAN_FILES) $(F2PY3_LIBS)
    $(F2PY3) -h $(MODULE).py $(F2PY3_H_FLAGS) $(FORTRAN_FILES)

clean:
    -rm $(MODULE)*.so
    -rm $(MODULE).py
```

FIGURE 3.1 – Fichier `Makefile` fourni à toutes fins utiles. Il construit un paquetage Python appelé `mon_code_fortran`, permettant d'appeler les fonctions présentes dans les fichiers `backward_l.f` et `forward_l.f` grâce à l'utilitaire `f2py3`. Bien que ce ne soit pas nécessaire dans le cadre de ce chapitre, le `Makefile` est prévu pour faire l'édition des liens non seulement avec la bibliothèque BLAS mais aussi avec la bibliothèque LAPACK, qui contient des implantations des algorithmes d'algèbre linéaire numérique étudiés dans ce cours.

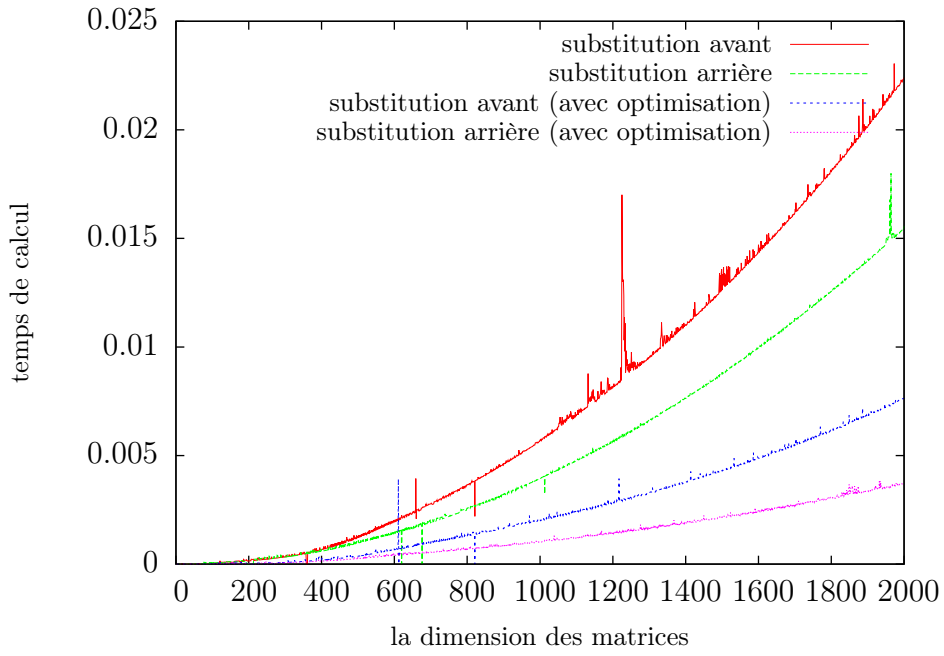


FIGURE 3.2 – Comparaison des temps de calcul (fonction `CPU_TIME` de FORTRAN) des susbstitutions avant et arrière, sans et avec optimisation (option `-O2`). Dans les deux cas, la substitution arrière a de meilleures performances.

Bibliographie

Chapitre 4

Factorisation triangulaire

Ce chapitre doit beaucoup à [2, Lecture 14]. Une façon simple de résoudre les systèmes d'équations linéaires $A\,x = b$ avec A inversible, de dimension $m \times m$ (systèmes carrés), consiste à factoriser la matrice A en un produit de matrices

$$A = L\,U\,,$$

où L est triangulaire inférieure et U est triangulaire supérieure (factorisation triangulaire).

4.1 Factoriser pour résoudre

En effet, la solution du système d'équations s'obtient ensuite en enchaînant deux résolutions de systèmes triangulaires, grâce aux méthodes du chapitre précédent :

1. calculer y tel que $L\,y = b$,
2. calculer x tel que $U\,x = y$.

On illustre ce principe avec Python

```
>>> import scipy.linalg as nla
>>> import numpy as np
```

On définit une matrice A

```
>>> A = np.array([[2,-1],[1,3]])
>>> A
array([[ 2, -1],
       [ 1,  3]])
```

et un vecteur b

```
>>> b = np.array([-1,6])
>>> b
array([-1,  6])
```

On cherche à résoudre $A\,x = b$. La fonction `lu` de `scipy.linalg` retourne une factorisation $A = P\,L\,U$ de A où P est une matrice de permutation.

```
>>> P, L, U = nla.lu (A)
>>> P
array([[1., 0.],
       [0., 1.]])
>>> L
array([[1. , 0. ],
       [0.5, 1. ]])
>>> U
array([[ 2. , -1. ],
       [ 0. ,  3.5]])
```

La matrice P est la matrice identité. Oublions la et vérifions que le produit $L\,U$ donne A

```
>>> np.dot (L,U)
array([[ 2., -1.],
       [ 1.,  3.]])
```

Utilisons maintenant les deux facteurs L et U pour résoudre le système d'équations linéaires en utilisant les méthodes du chapitre précédent

```
>>> y = nla.solve_triangular(L, b, lower=True)
>>> y
array([-1. ,  6.5])

>>> x = nla.solve_triangular(U, y, lower=False)
>>> x
array([0.42857143, 1.85714286])
```

Vérification finale. On retrouve bien b.

```
>>> np.dot (A, x)
array([-1.,  6.])
```

Pourquoi ? Une matrice code une application linéaire. Le produit de deux matrices code la composition des deux applications. Ce point de vue est illustré par l'exemple suivant. On considère le système d'équations $L\,U\,x = b$:

$$\begin{pmatrix} 1 & 0 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \end{pmatrix} .$$

Comme le produit de matrices est associatif, on peut commencer par calculer $U\,x$. On obtient :

$$\begin{pmatrix} 1 & 0 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} 2\,x_1 + x_2 \\ -x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \end{pmatrix} .$$

Effectuons la multiplication par L. On obtient :

$$\begin{cases} (2\,x_1 + x_2) & = & 5\,, \\ 3\,(2\,x_1 + x_2) + 2\,(-x_2) & = & 1\,. \end{cases}$$

Nommons y_1 et y_2 les expressions entre parenthèses. Le système de deux équations ci-dessus a la même solution que le système de quatre équations ci-dessous (les deux dernières équations ne servent qu'à exprimer le « nommage » que nous venons de faire). Ces quatre équations peuvent être résolues en résolvant d'abord le système de gauche ($L\,y = b$), puis le système de droite ($U\,x = y$).

$$\begin{cases} y_1 & = & 5\,, \\ 3\,y_1 + 2\,y_2 & = & 1\,, \end{cases} \qquad \begin{cases} 2\,x_1 + x_2 & = & y_1\,, \\ -x_2 & = & y_2\,. \end{cases}$$

Question 17. Terminer la résolution.

Si la matrice A n'a pas de structure particulière, on peut la factoriser grâce au pivot de Gauss. Si elle est symétrique définie positive, il vaut mieux utiliser l'algorithme de Cholesky [1], qui a une meilleure complexité, et de meilleures propriétés de stabilité. On commence par le cas particulier des matrices symétriques définies positives.

4.2 Digression sur les matrices symétriques définies positives

Une matrice est symétrique si elle est égale à sa transposée. La symétrie d'une matrice est une propriété qui peut s'observer à l'œil nu. Ce n'est pas le cas de la propriété suivante :

Définition 2 *Soit A une matrice symétrique. A est dite* définie positive *si, quel que soit le vecteur $x \neq 0$, on a $x^T\,A\,x > 0$.*

La première fois qu'on rencontre ces deux définitions, on se demande fréquemment le sens qu'elles ont. Après tout, si on tire aléatoirement les coefficients d'une matrice, on n'a aucune chance d'obtenir une matrice symétrique. Sans parler de la propriété de positivité définie, qui semble terriblement arbitraire. Dans les quelques lignes ci-dessous, on survole très rapidement une notion qui sera revue en détail au chapitre 7, dans le but de montrer au moins un exemple où des matrices symétriques définies positives apparaissent naturellement.

On suppose donnés m points dans le plan $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$. On cherche les coefficients α, β d'une droite d'équation $y = \alpha\,x + \beta$ qui passe au plus près des m points. Bien sûr, pour que le problème ait une solution, il faut que $m \geqslant 1$ et même, plus précisément, qu'au moins deux points aient des abscisses distinctes.

1. Le « Commandant Cholesky » est né en 1875 et mort en 1918. Voir [1] pour l'histoire de l'algorithme qui porte son nom.

Les deux méthodes étudiées au chapitre 7 commencent par écrire un système d'équations un peu bizarre [2], où on pose que la droite passe par les m points :

$$\begin{aligned} x_1\,\alpha + \beta &= y_1 \\ x_2\,\alpha + \beta &= y_2 \\ &\vdots \\ x_m\,\alpha + \beta &= y_m \end{aligned}$$

Sous forme matricielle, ce système s'écrit $A\,x = b$ avec :

$$\underbrace{\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \alpha \\ \beta \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}}_{b}$$

La méthode historique consiste ensuite à former le système suivant, dont la solution fournit les coefficients α, β recherchés :

$$\underbrace{\begin{pmatrix} \times & \times \\ \times & \times \end{pmatrix}}_{A^T A} \underbrace{\begin{pmatrix} \alpha \\ \beta \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} \times \\ \times \end{pmatrix}}_{A^T b}$$

La matrice $A^T A$ est nécessairement symétrique (voir question 4, page 17). On peut montrer (l'argument clef est énoncé dans la proposition 14, page 76) qu'elle est définie positive si et seulement si le problème posé a une solution (au moins deux points ont des abscisses distinctes).

4.3 L'algorithme de Cholesky

On pourrait espérer que, dans le cas d'une matrice symétrique A, on puisse trouver une factorisation $A = L\,U$ qui soit elle-même symétrique, c'est-à-dire telle que $A = L\,L^T$. Ce n'est pas toujours le cas : il y a une condition supplémentaire.

Proposition 3 *Soit A une matrice inversible telle que $A = L\,L^T$. Alors, A est symétrique et définie positive.*

> **Preuve.** Le fait que A soit symétrique est montré dans la question 4, page 17. Montrons la seconde partie de la proposition. On considère un vecteur $x \neq 0$. On montre que $x^T\,A\,x > 0$. Comme A est inversible et $x \neq 0$ on a $A\,x = L\,L^T\,x \neq 0$ [théorème 1, (4)]. Donc $y = L^T\,x \neq 0$. En utilisant la formule pour la transposée d'un produit, on voit que $x^T\,A\,x = (x^T\,L)\,(L^T\,x) = y^T\,y > 0$ (cf. exercice 9, page 20). □

2. Ce qui est bizarre, c'est d'écrire un système dont on sait à l'avance qu'il n'a pas de solution.

Proposition 4 *Si A est symétrique définie positive, alors A admet une factorisation $A = L\,L^T$. Si, de plus, on impose que les éléments diagonaux de L soient positifs, alors cette décomposition est unique. Elle est appelée factorisation de Cholesky.*

La proposition se prouve en produisant un algorithme de calcul de la factorisation de Cholesky. Soit A une matrice symétrique définie positive. Considérons un partitionnement de A

$$A = \begin{pmatrix} \alpha & a^T \\ a & A_2 \end{pmatrix} .$$

On établit tout d'abord que $\alpha > 0$ (si cette condition n'est pas vérifiée, l'algorithme échoue et on a montré que la matrice symétrique A n'est pas définie positive).

Preuve. Prenons $x = (\ 1 \quad 0 \quad \cdots \quad 0\ )^T$. Alors

$$x^T\,A\,x = (\ 1 \quad 0\ ) \begin{pmatrix} \alpha & a^T \\ a & A_2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \alpha\,.$$

Comme A est définie positive, $\alpha > 0$. □

On cherche maintenant une factorisation de la forme

$$A = \begin{pmatrix} \alpha & a^T \\ a & A_2 \end{pmatrix} = \begin{pmatrix} \beta & 0 \\ b & B_2 \end{pmatrix} \begin{pmatrix} \beta & b^T \\ 0 & B_2^T \end{pmatrix} = \begin{pmatrix} \beta^2 & \beta\,b^T \\ \beta\,b & b\,b^T + B_2\,B_2^T \end{pmatrix} .$$

On trouve $\beta = \sqrt{\alpha}$, qui est bien défini puisque $\alpha > 0$ puis, $b = (1/\beta)\,a$ et $B_2\,B_2^T = A_2 - b\,b^T$. La matrice $B_2\,B_2^T$ est symétrique. Sa dimension est $(m-1)\times(m-1)$. Elle n'est pas triangulaire inférieure mais, si on arrive à calculer une matrice triangulaire inférieure L_2 telle que $B_2\,B_2^T = L_2\,L_2^T$, il suffit de remplacer B_2 par L_2 dans la formule précédente pour obtenir la factorisation recherchée. Calculatoirement, c'est facile : il suffit d'appliquer récursivement notre algorithme sur $B_2\,B_2^T$. Pour la justification, il reste quand même à démontrer que $B_2\,B_2^T$ est définie positive.

Preuve. Considérons le produit

$$(\ \eta \quad v^T\ )\,A \begin{pmatrix} \eta \\ v \end{pmatrix} = (\ \eta \quad v^T\ ) \begin{pmatrix} \alpha & a^T \\ a & A_2 \end{pmatrix} \begin{pmatrix} \eta \\ v \end{pmatrix} = \alpha\,\eta^2 + 2\,\eta\,v^T\,a + v^T\,A_2\,v\,.$$

Comme A est définie positive, cette expression est strictement positive, quels que soient le scalaire η et le vecteur v de dimension $m-1$ non nul. Remplaçons A_2 par $b\,b^T + B_2\,B_2^T$ et rappelons-nous que $b = (1/\sqrt{\alpha})\,a$. On trouve

$$\alpha\,\eta^2 + 2\,\eta\,v^T\,a + v^T\,A_2\,v = \underbrace{\alpha\,\eta^2 + 2\,\eta\,v^T\,a + \frac{1}{\alpha}(v^T\,a)^2}_{=0 \text{ pour } \eta = -\frac{1}{\alpha}\,v^T\,a} + v^T\,B_2\,B_2^T\,v\,.$$

On en conclut que, quel que soit le vecteur v non nul, $v^T\,B_2\,B_2^T\,v > 0$, c'est-à-dire que $B_2\,B_2^T$ est définie positive. □

4.3.1 Exemple

Partons de la matrice

$$L = \begin{pmatrix} 2 & 0 & 0 \\ 3 & 4 & 0 \\ -5 & -8 & 6 \end{pmatrix}.$$

Elle est triangulaire inférieure. Ses éléments diagonaux sont positifs. Par conséquent, en appliquant l'algorithme de Cholesky sur $A = L\,L^T$, on doit la retrouver.

$$A = \begin{pmatrix} 4 & 6 & -10 \\ 6 & 25 & -47 \\ -10 & -47 & 125 \end{pmatrix}.$$

À la première itération, on trouve

$$\left(\begin{array}{c|cc} 2 & 0 & 0 \\ \hline 3 & 16 & -32 \\ -5 & -32 & 100 \end{array}\right) \qquad \begin{pmatrix} 16 & -32 \\ -32 & 100 \end{pmatrix} = \begin{pmatrix} 25 & -47 \\ -47 & 125 \end{pmatrix} - \begin{pmatrix} 3 \\ -5 \end{pmatrix} \cdot \begin{pmatrix} 3 & -5 \end{pmatrix}.$$

À l'itération suivante, on trouve

$$\left(\begin{array}{cc|c} 2 & 0 & 0 \\ 3 & 4 & 0 \\ \hline -5 & -8 & 36 \end{array}\right) \qquad 36 = 100 - (-8) \times (-8)\,.$$

Enfin, à la dernière itération, on trouve la matrice L :

$$\begin{pmatrix} 2 & 0 & 0 \\ 3 & 4 & 0 \\ -5 & -8 & 6 \end{pmatrix}.$$

4.3.2 L'algorithme

Une implantation FORTRAN de l'algorithme de Cholesky est donnée ci-dessous. Comme A est symétrique, seule sa partie triangulaire inférieure est utilisée. En sortie, la matrice L résultat se trouve à la place de A. L'algorithme fournit aussi un outil pour décider si une matrice symétrique est définie positive.

```
      SUBROUTINE CHOLESKY (M, A)
      INTEGER M
      DOUBLE PRECISION A(M,M), BETA
      INTEGER I, J, K, L
      DO J = 1,M
* A(J,J) = alpha est nécessairement strictement positif
         IF (A(J,J) .LE. 0D0) STOP 'A non définie positive'
* beta = racine carrée de alpha
```

```
         BETA = SQRT(A(J,J))
         A(J,J) = BETA
* b = (1/beta) * a
         DO K = J+1,M
            A(K,J) = A(K,J)/BETA
         END DO
* B2 . B2**T = A2 - b . b**T
         DO L = J+1,M
            DO K = L,M
               A(K,L) = A(K,L) - A(K,J)*A(L,J)
            END DO
         END DO
* L'itération suivante, applique l'algorithme sur B2 . B2**T
      END DO
      END
```

En Python

La fonction `cho_factor` permet de calculer la matrice L de la factorisation de Cholesky. On a ici précisé le type de flottants ainsi que l'organisation en mémoire de la matrice (colonne par colonne) de façon à ce qu'une implantation efficace (codée dans LAPACK) de l'algorithme soit appelé. En particulier, ces précisions permettent d'activer le paramètre `overwrite_a=True` de telle sorte que la matrice résultat soit stockée à la place de la matrice initiale. En sortie, on remarque que la matrice L recherchée est bien présente dans A mais que la partie triangulaire supérieure n'est pas mise à zéro. En effet, cette mise à zéro est une perte de temps puisqu'il est ensuite possible, dans le cadre d'une résolution de système linéaire, de demander à `solve_triangular` de n'accéder qu'à la partie triangulaire inférieure de A.

```
>>> import scipy.linalg as nla
>>> import numpy as np
>>> A = np.array([[4,6,-10],[6,25,-47],[-10,-47,125]],dtype=np.float64,order='F')
>>> nla.cho_factor(A, lower=True, overwrite_a=True)
(array([[  2.,   6., -10.],
       [  3.,   4., -47.],
       [ -5.,  -8.,   6.]]), True)
>>> A
array([[  2.,   6., -10.],
       [  3.,   4., -47.],
       [ -5.,  -8.,   6.]])
```

Supposons maintenant qu'on souhaite utiliser cette factorisation $A = L\,L^T$ pour résoudre un système $A\,x = b$.

```
>>> b = np.array([42,175,-401],dtype=np.float64)
```

Il suffit maintenant d'enchaîner les deux résolutions de systèmes triangulaires. On remarque que l'optimisation qui consiste à remplacer le vecteur b par la solution du système est très pratique. Au final, aussi bien pour factoriser A que pour résoudre le système, on n'a utilisé aucune matrice et aucun vecteur supplémentaire.

```
>>> nla.solve_triangular (A, b, lower=True, overwrite_b=True)
array([ 21.,  28., -12.])
```

Dans l'appel ci-dessous, on indique à `solve_triangular` d'appliquer une remontée avec, pour matrice U, la transposée de L.

```
>>> nla.solve_triangular (A, b, lower=True, trans='T', overwrite_b=True)
array([ 1.,  3., -2.])
```

Le vecteur ci-dessous est bien la solution recherchée.

```
>>> b
array([ 1.,  3., -2.])
```

Stabilité et complexité

L'algorithme de Cholesky est stable[3] [3, Lecture 23, Theorem 23.2]. Voir le chapitre 5.

On peut calculer sa complexité avec MAPLE. On commence par calculer le nombre $h(\ell, m)$ d'opérations effectuées à chaque itération de la boucle la plus intérieure. Puis, on calcule le nombre $g(j, m)$ d'opérations effectuées à chaque itération de la boucle principale. Enfin, on calcule le nombre $f(m)$ d'opérations effectuées par l'algorithme de Cholesky. On trouve que $f(m)$ est de l'ordre de $(1/3)\, m^3$.

```
> hlm := sum (2, k = l .. m);
                              hlm := 2 m - 2 l + 2

# une racine carrée + le calcul de b + le calcul de M
> gjm := expand (1 + sum (1, k = j+1..m) + sum (hlm, l = j+1..m));
                                          2            2
                    gjm := 1 + 2 m - 2 j + m  - 2 m j + j

> fm := expand (sum (gjm, j = 1 .. m));
                                    3        2
                          fm := 1/3 m  + 1/2 m  + 1/6 m
```

3. Si on veut être précis, il faut faire attention : notons $\tilde{L}$ la matrice retournée par l'algorithme. Le produit $\tilde{L}\,\tilde{L}^T$ est proche de A mais la matrice $\tilde{L}$ n'est pas nécessairement proche de L. Voir [3, Lecture 23] et [3, Lecture 16] pour une démonstration numérique, dans un cas analogue.

Complément sur l'utilisation des BLAS

Le code FORTRAN peut être simplifié en utilisant les BLAS. Par exemple, la boucle

```
DO K = J+1,M
   A(K,J) = A(K,J)/BETA
END DO
```

aurait pu être remplacée par un appel à `DSCAL`, qui multiplie un vecteur par un scalaire :

```
CALL DSCAL (M - J, 1D0/BETA, A(J+1,J), 1)
```

Des BLAS auraient aussi pu être utilisées pour le calcul de M, et remplacer les deux boucles imbriquées mais ce n'est pas immédiat ici. La fonction à utiliser s'appelle `DSPR` mais elle suppose que la matrice A est stockée en mémoire d'une façon compacte : d'abord les m éléments de la première colonne de A, puis les $m-1$ éléments de la deuxième colonne, en commençant à a_{22}, puis les $m-2$ éléments de la troisième colonne, en commençant à a_{33} etc.

Complément sur la bibliothèque LAPACK

La bibliothèque LAPACK est une bibliothèque d'algèbre linéaire numérique, construite au-dessus des BLAS. Elle contient plusieurs implantations de l'algorithme de Cholesky.

```
SUBROUTINE CHOLESKY (M, A)
INTEGER M
DOUBLE PRECISION A(M,M)
INTEGER INFO
CALL DPOTRF ('L', M, A, M, INFO)
IF (INFO .NE. 0) STOP 'Erreur DPOTRF'
END
```

Dans l'identificateur `DPOTRF`, la première lettre `'D'` indique que les flottants sont en double précision ; les deux suivantes `'PO'` indiquent que la matrice est symétrique définie positive et stockée dans un tableau rectangulaire. Les trois dernières lettres `'TRF'` indiquent que la fonction effectue une factorisation dont les facteurs sont triangulaires.

Le premier paramètre `'L'` indique à `DPOTRF` d'utiliser la partie triangulaire inférieure de A. Les deux suivants sont la dimension m et la matrice A. Le quatrième paramètre est le nombre de lignes de A en tant que tableau FORTRAN (ce paramètre serait utile si A était plongée dans un tableau à deux dimensions surdimensionné). Le dernier paramètre permet à `DPOTRF` de retourner un code d'erreur au programme appelant. La valeur 0 signifie qu'il n'y a pas eu d'erreur.

4.4 Le pivot de Gauss

4.4.1 La version simple

Les opérations suivantes ne changent pas les solutions d'un système d'équations linéaires $A\,x = b$:

— multiplier une ligne par un scalaire non nul ;
— permuter deux lignes ;
— ajouter à une ligne, un multiple d'une autre ligne.

Comme ces opérations doivent être faites à la fois sur les membres gauche et droit des équations, on peut les réaliser en manipulant la matrice $(A \mid b)$. On considère les colonnes les unes après les autres. À chaque itération, on cherche un pivot (un élément non nul) sur la colonne courante (cette opération peut conduire à permuter deux lignes), puis on s'arrange pour annuler tous les éléments de la colonne courante, sous le pivot (cette opération conduit à ajouter un multiple de la ligne du pivot aux lignes qui sont situées au dessous). À la fin, on a un système triangulaire qu'on peut résoudre par l'algorithme de substitution avant ou arrière.

Proposition 5 *Si la matrice A est inversible, on est certain de trouver un pivot à chaque itération.*

> **Preuve.** Comme A est inversible, le système a une unique solution [théorème 1, (5)]. L'algorithme du pivot de Gauss ne change pas les solutions du système d'équations linéaires. À chaque itération, le système considéré a donc une unique solution et est donc de rang m [théorème 1, (2)]. Or l'absence de pivot à une itération k impliquerait que la $k$ième colonne soit une combinaison linéaire des colonnes précédentes, et que le rang du système soit strictement inférieur à m. □

4.4.2 La version élaborée

Dans cette section, on se concentre sur la justification mathématique de l'algorithme. Les considérations numériques sont reportées à plus tard.

L'avantage de la version élaborée sur la version simple, c'est qu'on calcule une décomposition $A = L\,U$ de A, qui permet de résoudre le système $A\,x = b$ pour plusieurs vecteurs b, et même des vecteurs b qu'on ne connaît pas encore au moment du traitement de A. On rencontrera une telle situation au chapitre 8, avec la méthode de la puissance inverse.

Formulation matricielle des opérations de lignes

Comme la version simple, la version élaborée n'effectue que des opérations de lignes, opérations qui peuvent être codées par des multiplications à gauche, par des matrices M_i. Considérons la matrice

$$A = A_1 = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}.$$

Supposons que le premier pivot soit a_{11} et notons

$$m_{k1} = \frac{a_{k1}}{a_{11}}\,, \qquad 2 \leqslant k \leqslant m\,.$$

Ces nombres sont appelés les *multiplicateurs*. À la première itération, l'algorithme du pivot de Gauss soustrait m_{k1} fois la première ligne à la $k$ième, pour $2 \leqslant k \leqslant m$. Le résultat est de la forme

$$A_2 = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{pmatrix}.$$

Mathématiquement, ce résultat s'obtient en multipliant A_1, à gauche, par une matrice M_1, de la façon suivante :

$$A_2 = M_1\, A_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -m_{21} & 1 & 0 & 0 \\ -m_{31} & 0 & 1 & 0 \\ -m_{41} & 0 & 0 & 1 \end{pmatrix} A_1\,.$$

Supposons que le deuxième pivot soit a'_{22} et notons $m_{k2} = a'_{k2}/a'_{22}$, pour $3 \leqslant k \leqslant m$. À la deuxième itération, l'algorithme du pivot de Gauss soustrait m_{k2} fois la deuxième ligne de A_2 à la $k$ième, pour $3 \leqslant k \leqslant m$. Le résultat est de la forme

$$A_3 = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & a''_{43} & a''_{44} \end{pmatrix}.$$

Mathématiquement, il s'obtient en multipliant A_2, à gauche, par une matrice M_2, de la façon suivante :

$$A_3 = M_2\, A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -m_{32} & 1 & 0 \\ 0 & -m_{42} & 0 & 1 \end{pmatrix} A_2\,.$$

Supposons enfin que le dernier pivot soit a''_{33} et notons $m_{43} = a''_{43}/a''_{33}$. À la dernière itération, l'algorithme du pivot de Gauss soustrait m_{43} fois la troisième ligne de A_3 à la quatrième. Le résultat est une matrice triangulaire supérieure

$$U = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a''_{33} & a''_{34} \\ 0 & 0 & 0 & a'''_{44} \end{pmatrix}.$$

Mathématiquement, il s'obtient en multipliant A_3, à gauche, par une matrice M_3, de la façon suivante :

$$U = M_3\, A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -m_{43} & 1 \end{pmatrix} A_3\,.$$

En résumé, on a calculé $M_3\, M_2\, M_1\, A = U$.

Premier coup de chance

Les matrices M_i sont inversibles. Il y a de nombreuses façons simples de s'en convaincre. Par exemple : chaque M_i est inversible parce que l'opération de lignes qu'elle code est réversible.

On a donc une factorisation $A = (M_3\, M_2\, M_1)^{-1}\, U = M_1^{-1}\, M_2^{-1}\, M_3^{-1}\, U$.

Bien que $M_3\, M_2\, M_1$ ait une expression compliquée, la matrice $L = M_1^{-1}\, M_2^{-1}\, M_3^{-1}$ s'écrit très simplement. En particulier elle est triangulaire inférieure, avec des 1 sur la diagonale :

$$L \;=\; \begin{pmatrix} 1 & & & \\ m_{21} & 1 & & \\ m_{31} & m_{32} & 1 & \\ m_{41} & m_{42} & m_{43} & 1 \end{pmatrix}. \tag{4.1}$$

Pour s'en convaincre, on peut commencer par remarquer (il suffit de faire les produits) que les inverses des matrices M_i ont des expressions simples :

$$M_1^{-1},\, M_2^{-1},\, M_3^{-1} \;=\; \begin{pmatrix} 1 & & & \\ m_{21} & 1 & & \\ m_{31} & & 1 & \\ m_{41} & & & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & & & \\ & 1 & & \\ & m_{32} & 1 & \\ & m_{42} & & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & m_{43} & 1 \end{pmatrix}.$$

Le diagramme suivant explique ensuite le phénomène qui se produit lorsqu'on développe le produit de ces trois matrices. Concentrons-nous sur les colonnes délimitées par des traits pleins : la première colonne de M_1^{-1} est recopiée dans la matrice résultat $M_1^{-1}\, M_2^{-1}$ parce que la première colonne de M_2^{-1} est celle d'une matrice identité. Concentrons-nous maintenant sur les sous-matrices délimitées par des traits en pointillés : la sous-matrice en bas à droite de M_2^{-1} est recopiée dans la sous-matrice correspondante de la matrice résultat parce qu'elle est multipliée par une sous-matrice de M_1^{-1} qui est une matrice identité. Par un raisonnement similaire, on se convainc qu'en multipliant la matrice résultat par M_3^{-1} on obtient bien la matrice L donnée en (4.1).

$$M_2^{-1} = \begin{pmatrix} \boxed{1} & \\ & \boxed{\begin{matrix} 1 & \\ m_{33} & 1 \\ m_{42} & & 1 \end{matrix}} \end{pmatrix}$$

$$M_1^{-1} = \begin{pmatrix} \boxed{\begin{matrix} 1 \\ m_{21} \\ m_{31} \\ m_{41} \end{matrix}} & \boxed{\begin{matrix} 1 & & \\ & 1 & \\ & & 1 \end{matrix}} \end{pmatrix} \begin{pmatrix} \boxed{\begin{matrix} 1 \\ m_{21} \\ m_{31} \\ m_{41} \end{matrix}} & \boxed{\begin{matrix} 1 & \\ m_{32} & 1 \\ m_{42} & & 1 \end{matrix}} \end{pmatrix} = M_1^{-1}\, M_2^{-1}$$

Lors du déroulement du pivot de Gauss, il suffit donc de mémoriser les multiplicateurs pour calculer la matrice L. On peut même stocker les multiplicateurs à la place des zéros introduits par le pivot de Gauss, sous la diagonale de A.

Second coup de chance

On a négligé une difficulté : à chaque étape, l'élément diagonal peut être nul et on peut être amené à échanger la ligne courante avec une autre ligne, située plus bas.

Lors du pivot de Gauss, les permutations interviennent au cours de l'algorithme. Ne vont-elles pas casser la simplicité de l'algorithme décrit à la fin de la section précédente ? Non.

Mathématiquement, on peut coder dans une matrice P les permutations de lignes effectuées par l'algorithme. Le principe est le suivant. Soit P_{ij} la matrice obtenue en permutant les lignes i et j de la matrice identité de dimension $m \times m$. Permuter les lignes i et j de la matrice A revient alors, mathématiquement, à calculer $P_{ij}\,A$. Notons $P_1, P_2, \ldots, P_{m-1}$ les matrices codant les permutations aux itérations $1, 2, \ldots, m-1$ (certaines matrices peuvent être égales à l'identité, si aucune permutation n'était nécessaire). L'ensemble des permutations effectuées par l'algorithme est alors codé par la matrice de permutation $P = P_{m-1} \cdots P_2\, P_1$.

On admet la proposition suivante.

Proposition 6 *Si, durant le déroulement du pivot de Gauss, à chaque fois qu'une permutation doit être effectuée, on permute à la fois les éléments de A et ceux de L, alors le résultat est une décomposition $L\,U$ de la matrice $P\,A$.*

On remarque que, si on a stocké les multiplicateurs à la place des zéros introduits par le pivot de Gauss sous la diagonale de A, il suffit d'effectuer les permutations de lignes sur toute leur largeur (pour $j = 1, 2, \ldots, m$) pour obtenir la matrice L associée à $P\,A$!

Enfin, pour résoudre après coup un système $A\,x = b$, il suffit de reproduire les permutations effectuées au cours de l'algorithme sur le vecteur b, avant d'appliquer la méthode décrite en section 4.1.

Preuve. En effet $A\,x = b$ implique $P\,A\,x = P\,b$ et donc $L\,U\,x = P\,b$. □

Pour reproduire ces permutations, il suffit (voir ci-dessous) de mémoriser, à chaque itération, l'indice de ligne du pivot.

4.4.3 L'algorithme

En FORTRAN

Le code ci-dessous implante l'algorithme décrit dans la section précédente, avec deux modifications :

1. au lieu de calculer P explicitement, l'algorithme mémorise l'indice de la ligne du pivot dans un tableau `IPIV` ;
2. à chaque étape k, on choisit comme pivot, le plus grand élément de la colonne k, en valeur absolue (stratégie du *partial pivoting* en Anglais).

```
SUBROUTINE LUDCMP (M, A, IPIV)
INTEGER M
```

```
      DOUBLE PRECISION A (M,M)
      INTEGER IPIV (M-1)
      DOUBLE PRECISION MAXELT, TMP
      INTEGER I, J, K
      DO K = 1,M-1
* Recherche du pivot (indice dans IPIV(K))
         MAXELT = ABS(A(K,K))
         IPIV(K) = K
         DO I = K+1,M
            IF (ABS (A(I,K)) .GT. MAXELT) THEN
               MAXELT = ABS (A(I,K))
               IPIV(K) = I
            END IF
         END DO
         IF (MAXELT .EQ. 0D0) STOP 'Matrice singulière'
* Échange de la ligne courante avec celle du pivot
* Les multiplicateurs sont échangés en même temps que les éléments de A
         DO J = 1,M
            TMP = A(K,J)
            A(K,J) = A(IPIV(K),J)
            A(IPIV(K),J) = TMP
         END DO
* Mémorisation des multiplicateurs en dessous du pivot
         DO I = K+1,M
            A(I,K) = A(I,K)/A(K,K)
         END DO
* À chaque ligne de A, on soustrait le multiplicateur fois la ligne du pivot
         DO J = K+1,M
            DO I = K+1,M
               A(I,J) = A(I,J) - A(I,K)*A(K,J)
            END DO
         END DO
      END DO
      END
```

En Python

```
>>> import scipy.linalg as nla
>>> import numpy as np
```

On entre le même système que pour l'algorithme de Cholesky : bien que A soit symétrique, on choisit d'appliquer l'algorithme de factorisation LU général.

```
>>> A = np.array([[4,6,-10],[6,25,-47],[-10,-47,125]],dtype=np.float64,order='F')
>>> b = np.array([42,175,-401],dtype=np.float64)
```

On applique l'algorithme. La matrice A a bien été modifiée. La fonction `lu_factor` retourne un couple formé de A (modifiée) et du tableau `piv` qui code la matrice de permutation (tableau `IPIV` du code FORTRAN).

```
>>> lu,piv = nla.lu_factor(A, overwrite_a=True)
>>> A
array([[-10.  , -47.  , 125.  ],
       [ -0.4 , -12.8 ,  40.  ],
       [ -0.6 ,   0.25,  18.  ]])
>>> piv
array([2, 2, 2], dtype=int32)
```

Comment tenir compte de `piv` ? On cherche à résoudre $A\,x = P\,b$. Par conséquent, il suffit de résoudre $L\,U\,x = P\,b$ et donc de remplacer b par $P\,b$ avant de lancer les résolutions de systèmes triangulaires. C'est ce que fait la boucle suivante. L'expression `piv.shape[0]` fournit la dimension de `piv`, c'est-à-dire m.

```
>>> for i in range(piv.shape[0]):
...     if i != piv[i]:
...         b[i],b[piv[i]] = b[piv[i]],b[i]
...
>>> b
array([-401.,   42.,  175.])
```

On résout $L\,y = P\,b$ (résultat dans b). La matrice L est incomplète, puisqu'il manque, dans A, les 1 situés sur la diagonale. On résout ce problème en passant à `solve_triangular` le paramètre `unit_diagonal=True`.

```
>>> nla.solve_triangular (A, b, lower=True, unit_diagonal=True, overwrite_b=True)
array([-401. , -118.4,  -36. ])
```

Il ne reste plus qu'à résoudre $U\,x = y$. Avant l'appel, le vecteur y est enregistré dans b. Après l'appel, c'est le vecteur x solution qui se trouve dans b.

```
>>> nla.solve_triangular (A, b, lower=False, overwrite_b=True)
array([ 1.,  3., -2.])
```

Complexité

On peut calculer la complexité de cet algorithme avec MAPLE. On commence par calculer le nombre $h(k,m)$ d'opérations arithmétiques exécutées dans la boucle la plus intérieure. On en déduit le nombre $g(k,m)$ d'opérations arithmétiques exécutées dans une itération de la boucle principale. Enfin, on calcule le nombre $f(m)$ d'opérations effectuées par l'algorithme complet. On trouve que $f(m)$ est de l'ordre de $(2/3)\,m^3$, ce qui est le double de l'algorithme de Cholesky.

```
> hkm := sum (2, i=k+1 .. m);
                                hkm := 2 m - 2 k
```

```
# Le calcul des multiplicateurs + la dernière boucle
> gkm := sum (1, i=k+1 .. m) + sum (hkm, j=k+1 .. m);
                      gkm := m - k + (m - k) (2 m - 2 k)

> fm := expand (sum (gkm, k = 1 .. m-1));
                                    2        3
                        fm := -1/2 m  + 2/3 m  - 1/6 m
```

On pourrait penser que le décompte est un peu rapide et souhaiter compter aussi les calculs de valeur absolue, les comparaisons de flottants ainsi que les permutations. Cela ne change pas le terme dominant de $f(m)$.

```
> gkm :=   1                      # première valeur absolue
         + sum (2, i=k+1 .. m)    # recherche du pivot
         + 1                      # test matrice singulière
         + sum (1, j=1 .. m)      # permutation
         + sum (1, i=k+1 .. m)    # mémorisation des multiplicateurs
         + sum (hkm, j=k+1 .. m); # la dernière boucle
                  gkm := 2 + 4 m - 3 k + (m - k) (2 m - 2 k)

> fm := expand (sum (gkm, k = 1 .. m-1));
                                          2        3
                      fm := -1/6 m + 3/2 m  + 2/3 m  - 2
```

Complément sur l'utilisation des BLAS

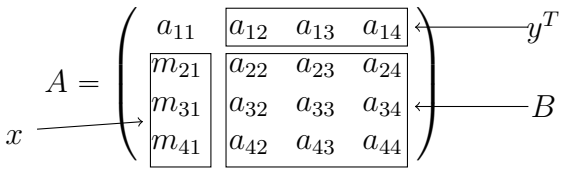
L'algorithme FORTRAN peut être codé de façon beaucoup plus compacte en utilisant les BLAS (voir ci-dessous). La fonction `IDAMAX` retourne l'indice du plus grand élément d'un vecteur (en valeur absolue). La fonction `DSWAP` permute deux vecteurs (les incréments valent m puisqu'on permute des vecteurs lignes). La fonction `DSCAL` multiplie un vecteur par un scalaire. Enfin, la fonction `DGER` prend en entrée un scalaire α, deux vecteurs x et y ainsi qu'une matrice A. Elle affecte à A la matrice $A + \alpha\, x\, y^T$ (matrice de rang 1).

```
      SUBROUTINE LUDCMP (M, A, IPIV)
      INTEGER M
      DOUBLE PRECISION A (M,M)
      INTEGER IPIV (M-1)
      INTEGER K, IDAMAX
      DO K = 1,M-1
* Recherche du pivot (indice dans IPIV(K))
         IPIV(K) = IDAMAX (M-K+1, A(K,K), 1) + K - 1
* Échange de la ligne courante avec celle du pivot
* Les multiplicateurs sont échangés en même temps que les éléments de A
         CALL DSWAP (M, A(K,1), M, A(IPIV(K),1), M)
* Mémorisation des multiplicateurs en dessous du pivot
         CALL DSCAL (M-K, 1D0/A(K,K), A(K+1,K), 1)
```

```
* À chaque ligne de A, on soustrait le multiplicateur fois la ligne du pivot
        CALL DGER (M-K, M-K, -1D0, A(K+1,K), 1, A(K,K+1), M,
     $              A(K+1,K+1), M)
      END DO
      END
```

L'emploi de la fonction `DGER` est expliqué dans la figure suivante. Les multiplicateurs sont mémorisés sous le pivot. Ils constituent un vecteur, noté x. La ligne du pivot forme un vecteur ligne, noté y^T. Il reste à soustraire à chaque ligne de la matrice désignée par B sur la figure, le multiplicateur placé sur cette ligne, fois y^T. Matriciellement, la nouvelle valeur de B est égale à l'ancienne moins le produit tensoriel $x\,y^T$:

$$B - x\,y^T\,.$$



Enfin, la bibliothèque LAPACK contient une implantation de la décomposition $L\,U$ d'une matrice rectangulaire, pas nécessairement de rang maximal. Cette fonction s'appelle `DGETRF` (c'est cette fonction qui est appelée par `lu_factor` en Python). Combinée avec `DGETRS`, elle permet de résoudre facilement un système d'équations linéaires carré.

4.4.4 Considérations numériques

Le pivot de Gauss sans *partial pivoting* est instable. Cet algorithme est déconseillé.

Le pivot de Gauss avec *partial pivoting* est considéré comme un très bon algorithme. Pourtant, en toute rigueur, c'est un algorithme instable mais, les matrices pour lesquelles « ça se passe mal » sont considérées comme rarissimes en pratique. C'est le cas des matrices ayant la structure suivante [3, Lecture 22] :

$$\begin{pmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{pmatrix}.$$

Question 18. À quoi ressemble cette matrice après application du pivot de Gauss ?

Bibliographie

[1] Jean-Luc Chabert, Évelyne Barbin, Michel Guillemot, Anne Michel-Pajus, Jacques Borowczyk, Ahmed Djebbar, and Jean-Claude Martzloff. *Histoire d'algorithmes. Du caillou à la puce*. Belin, Paris, 1994.

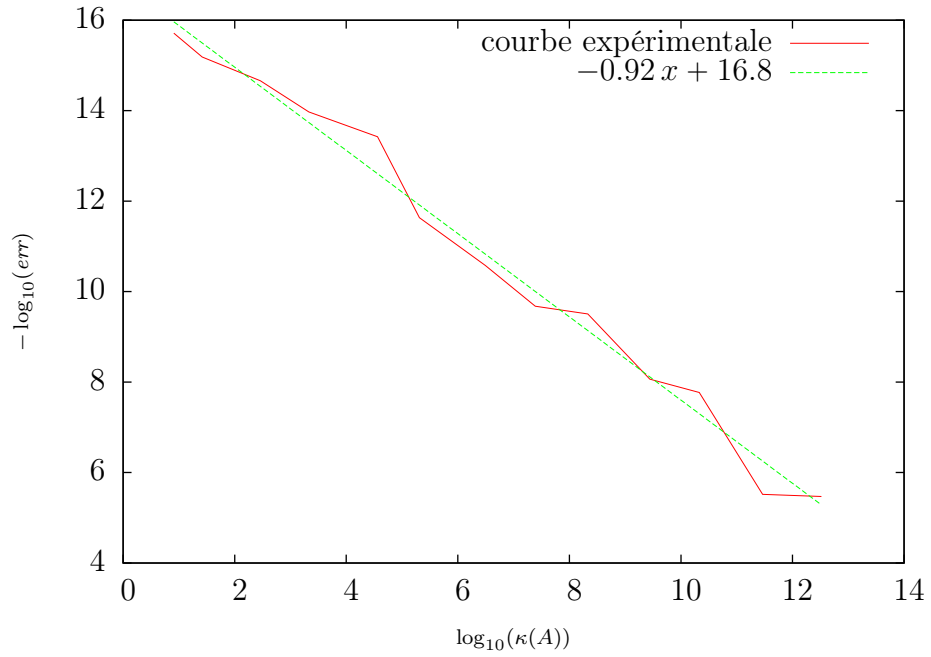
[2] Gilbert W. Stewart. *Afternotes on Numerical Analysis*. SIAM, 1996.

[3] Lloyd Nicholas Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

Chapitre 5

Conditionnement et stabilité

À toute matrice carrée A, on peut associer un réel $\kappa(A)$ (défini plus bas), appelé *condition* de A. Ce réel mesure la difficulté qu'il y a à résoudre un système d'équations $A\,x = b$, comme le montre le graphique ci-dessous.



La courbe expérimentale a été obtenue de la façon suivante : on a fixé un vecteur aléatoire x_0 et calculé des matrices aléatoires A pour différentes valeurs de $\kappa(A)$ (de 1 à 10^{14}). Pour chaque matrice A, on a calculé $b = A\,x_0$, résolu le système $A\,x = b$ par l'implantation LAPACK du pivot de Gauss et calculé l'erreur relative sur x, définie par $err = \|x - x_0\|/\|x_0\|$. En abscisse, on a mis le logarithme en base 10 de $\kappa(A)$; en ordonnée, le nombre de chiffres exacts (en base 10) de $\|x\|$. La droite a été obtenue en résolvant un problème de moindres carrés. Le graphique confirme l'idée intuitive que « si $\kappa(A)$ est de l'ordre de 10^k, il faut s'attendre à perdre k chiffres de précision lors du calcul de la solution ».

5.1 Notion de problème et de conditionnement

Un *problème* est une fonction mathématique $f : X \to Y$ qui, à une donnée $x \in X$, associe un résultat $f(x) \in Y$. Voici deux exemples de problèmes :

$$\begin{array}{rrcl} f_1 : & \mathbb{R}^m & \to & \mathbb{R}^m \\ & b & \mapsto & \text{la solution du système } A\,x = b, \text{ la matrice } A \text{ étant fixée} \end{array}$$

$$\begin{array}{rrcl} f_2 : & \mathbb{R}^{m\times m} & \to & \mathbb{R}^m \\ & A & \mapsto & \text{la solution du système } A\,x = b, \text{ le vecteur } b \text{ étant fixé} \end{array}$$

Définition 3 *Le* conditionnement *d'un problème f, pour une donnée x fixée, est le plus petit nombre $\kappa(x)$ tel que*

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leqslant \varepsilon \quad \Rightarrow \quad \frac{\|f(x) - f(\tilde{x})\|}{\|f(x)\|} \leqslant \kappa(x)\,\varepsilon\,.$$

Dans la définition, le réel ε est supposé petit. En toute rigueur, le conditionnement dépend de la norme utilisée mais on ne s'en souciera pas ici. Le conditionnement dépend du problème f et de la donnée x. Pour insister sur l'influence de la donnée, on parle du conditionnement du problème $f(x)$ (c'est un abus de langage classique [2, Lecture 12, page 89]) et on note $\kappa(x)$ plutôt que κ. Si le conditionnement $\kappa(x)$ est petit (inférieur à 100), le problème $f(x)$ est dit *bien conditionné*; s'il est grand (supérieur à 10^6), le problème est *mal conditionné*. Avec des mots, si l'erreur relative sur les données est inférieure à ε, alors l'erreur relative sur le résultat est inférieure au conditionnement du problème fois ε.

Comme cas particulier de perturbation, on peut considérer l'arrondi effectué lorsqu'on convertit le réel x (ou ses coefficients dans le cas d'un vecteur ou d'une matrice) en un nombre en virgule flottante : la perturbation sur les données provoque une erreur relative, sur les données, de l'ordre de $\varepsilon_{\text{machine}}$. D'après la définition du conditionnement, l'erreur sur le résultat vérifie :

$$\frac{\|f(x) - f(\tilde{x})\|}{\|f(x)\|} \leqslant \kappa(x)\,\varepsilon_{\text{machine}}\,.$$

En un sens $\kappa(x)\,\varepsilon_{\text{machine}}$ est l'erreur minimale faite par le meilleur algorithme imaginable qui calcule avec une arithmétique flottante. On verra plus loin que les algorithmes *backward stable* vérifient quasiment cette condition.

5.2 Condition d'une matrice

On appelle *condition* d'une matrice inversible A, le nombre

$$\kappa(A) = \|A\|\,\|A^{-1}\|\,.$$

Ici aussi, la condition d'une matrice A dépend de la norme utilisée. La définition est valable pour n'importe quelle norme. Toutefois, pour pouvoir faire le lien entre la condition de A

et le conditionnement d'un problème qui fait intervenir A il faut utiliser la même norme partout. La condition $\kappa(A)$ d'une matrice fournit une borne pour deux problèmes naturels liés à la résolution de systèmes d'équations linéaires $A\,x = b$: les problèmes f_1 et f_2 introduits ci-dessus. Quel est l'effet sur x d'une perturbation de b (problème f_1) ? On peut montrer [2, Theorem 12.1] que :

$$\kappa(x) \leqslant \kappa(A)\,.$$

Quel est l'effet sur x d'une perturbation de A (problème f_2) ? On peut montrer [2, Theorem 12.2] que :

$$\kappa(x) = \kappa(A)\,.$$

5.2.1 Considérations numériques

La bibliothèque LAPACK fournit plusieurs fonctions liées aux normes et aux conditions des matrices. La fonction `DLANGE` permet de calculer les trois normes matricielles données plus haut. La fonction `DGECON` calcule une estimation de la condition d'une matrice carrée à partir d'une décomposition $L\,U$, mais sans calculer explicitement A^{-1}.

Pour calculer une matrice aléatoire A d'une condition κ donnée, on peut utiliser la méthode décrite dans [1, Lecture 17] : calculer deux matrices orthogonales aléatoires U et V de dimension $m \times m$ et prendre

$$A = U \begin{pmatrix} 1 & & & & \\ & \kappa^{\frac{-1}{m-1}} & & & \\ & & \kappa^{\frac{-2}{m-1}} & & \\ & & & \ddots & \\ & & & & \kappa^{-1} \end{pmatrix} V\,.$$

Pour calculer une matrice aléatoire orthogonale, on peut calculer une matrice aléatoire M, calculer une décomposition $M = Q\,R$ et prendre Q.

5.3 Notion d'algorithme et de stabilité

Dans la section précédente on a défini la notion de problème, comme une fonction mathématique $f : X \to Y$.

Un *algorithme* pour $f(x)$ est une autre fonction $\tilde{f} : X \to Y$, conçue pour calculer $f(x)$, mais mise en œuvre avec une arithmétique flottante : la donnée x est donc arrondie avant d'être fournie à $\tilde{f}$; les opérations élémentaires sont ensuite effectuées avec une précision limitée. Le résultat de ce calcul est noté $\tilde{f}(x)$.

Définition 4 *Un algorithme $\tilde{f}$ pour un problème f est dit* backward stable *si il existe une constante $c > 0$ telle que, pour tout $x \in X$, on ait :*

$$\tilde{f}(x) = f(\tilde{x}) \qquad \textit{pour un certain } \tilde{x} \textit{ vérifiant} \qquad \frac{\|x - \tilde{x}\|}{\|x\|} \leqslant c\,\varepsilon_{\text{machine}}\,.$$

Avec des mots, un algorithme *backward stable* est un algorithme $\tilde{f}$ dont la sortie $\tilde{f}(x)$ est le résultat exact du problème f pour une donnée légèrement perturbée $\tilde{x}$.

La notion de stabilité inverse (*backward stability*) est importante en algèbre linéaire numérique parce qu'elle est souvent beaucoup plus facile à prouver que la stabilité simple, et que de nombreux algorithmes classiques la vérifient.

Elle est liée au conditionnement d'un problème par le théorème suivant [2, Theorem 15.1] :

Théorème 2 *Soit $\tilde{f}$ un algorithme* backward stable*, conçu pour résoudre un problème f. Alors, il existe une constante $c > 0$ telle que, pour tout $x \in X$, on ait :*

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \leqslant c\,\kappa(x)\,\varepsilon_{\text{machine}}\,,$$

où $\kappa(x)$ désigne le conditionnement du problème f en la donnée x.

Attention à bien comprendre la définition et le théorème : si on considère que $\varepsilon_{\text{machine}}$ est une constante, il sont vides de sens. Il faut voir ici $\varepsilon_{\text{machine}}$ comme une variable qui tend vers zéro, ce qui donne, dans le cas du théorème (attention à l'ordre des quantificateurs) : il existe une constante c telle que, quel que soit $\varepsilon_{\text{machine}}$ (paramètre qui suffit à définir une arithmétique flottante), l'erreur relative sur le résultat est inférieure à $c\,\kappa(x)\,\varepsilon_{\text{machine}}$. Voir la section 1.2.3, page 9.

Si on oublie la constante c, on voit que les algorithmes *backward stable* sont les « meilleurs algorithmes possibles ».

Il existe une notion d'algorithme *stable*, plus générale que *backward stable* (dans le sens où tout algorithme *backward stable* est *stable*) mais elle est plus difficile à comprendre. Voir [2, Lecture 14] pour une définition précise. On se contentera de la définition imprécise suivante : un algorithme stable est un algorithme $\tilde{f}$ dont la sortie $\tilde{f}(x)$ est le résultat légèrement perturbé du problème f pour une donnée légèrement perturbée $\tilde{x}$.

Il est écrit dans [2, Lecture 20, page 153] que, si un algorithme G fait appel à un sous-algorithme S *stable* mais pas *backward stable* pour résoudre un sous-problème, alors la stabilité de G est mise en danger.

Un algorithme qui n'est pas stable est dit *instable*.

5.4 La soustraction

Comme toutes les opérations en virgule flottante, la soustraction de l'arithmétique flottante est *backward stable* [2, Lecture 15]. Pour autant, le problème de la soustraction de deux réels peut être mal conditionné [1].

Le problème est la fonction $f : \mathbb{R}^2 \to \mathbb{R}$, qui à un couple $x = (x_1, x_2)$ associe $x_1 - x_2$. On commence par calculer le conditionnement. Soit $\varepsilon > 0$ un petit nombre réel. On cherche le

1. Si cette phrase vous surprend, vous devriez relire la section précédente.

plus petit nombre $\kappa(x)$ tel que

$$|\varepsilon_i| < \varepsilon \qquad \Rightarrow \qquad \frac{|(x_1\,(1+\varepsilon_1) - x_2\,(1+\varepsilon_2)) - (x_1 - x_2)|}{|x_1 - x_2|} = \frac{|x_1\,\varepsilon_1 - x_2\,\varepsilon_2|}{|x_1 - x_2|} \leqslant \kappa(x)\,\varepsilon\,.$$

On peut prendre

$$\kappa(x) = \frac{|x_1| + |x_2|}{|x_1 - x_2|}.$$

On voit que le conditionnement du problème varie beaucoup en fonction de x_1 et de x_2. Si les deux réels sont de signes différents, $\kappa(x) = 1$ et le problème est très bien conditionné. Par contre, si $x_1 \simeq x_2$, alors $\kappa(x)$ peut être très grand et le problème est très mal conditionné.

L'algorithme que nous allons considérer est $\tilde{f}(x) = \mathrm{fl}\,(x_1) \ominus \mathrm{fl}\,(x_2)$, où $\mathrm{fl}\,(x_i)$ désigne l'arrondi à $\varepsilon_{\text{machine}}$ près de x_i, et $\ominus$ la soustraction de l'arithmétique flottante. On montre que $\tilde{f}$ est *backward stable*. On a $\mathrm{fl}\,(x_i) = x_i\,(1+\varepsilon_i)$ où $|\varepsilon_i| \leqslant \varepsilon_{\text{machine}}$, pour $i = 1, 2$. Pour tenir compte de l'arrondi final effectué par $\ominus$, on multiplie le résultat de la soustraction par un facteur $1 + \varepsilon_3$, avec $|\varepsilon_3| \leqslant \varepsilon_{\text{machine}}$. Au final,

$$\begin{aligned} \mathrm{fl}\,(x_1) \ominus \mathrm{fl}\,(x_2) &= (x_1\,(1+\varepsilon_1) - x_2\,(1+\varepsilon_2))\,(1+\varepsilon_3) \\ &= x_1\,(1 + \varepsilon_1 + \varepsilon_3 + \varepsilon_1\,\varepsilon_3) - x_2\,(1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_2\,\varepsilon_3) \end{aligned}$$

Comme les ε_i sont supposés petits, les produits $\varepsilon_i\,\varepsilon_j$ le sont encore davantage. On en conclut l'existence de $\varepsilon_4, \varepsilon_5$, avec $|\varepsilon_4|, |\varepsilon_5| < 3\,\varepsilon_{\text{machine}}$, tels que

$$\mathrm{fl}\,(x_1) \ominus \mathrm{fl}\,(x_2) = x_1\,(1+\varepsilon_4) - x_2\,(1+\varepsilon_5)\,.$$

En d'autres termes, $\tilde{f}(x) = f(\tilde{x}) = \tilde{x}_1 - \tilde{x}_2$ pour certains $\tilde{x}_i$ vérifiant

$$\frac{|\tilde{x}_i - x_i|}{|x_i|} \leqslant 3\,\varepsilon_{\text{machine}}\,.$$

La soustraction flottante est donc bien *backward stable*.

Bibliographie

[1] Gilbert W. Stewart. *Afternotes on Numerical Analysis*. SIAM, 1996.

[2] Lloyd Nicholas Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

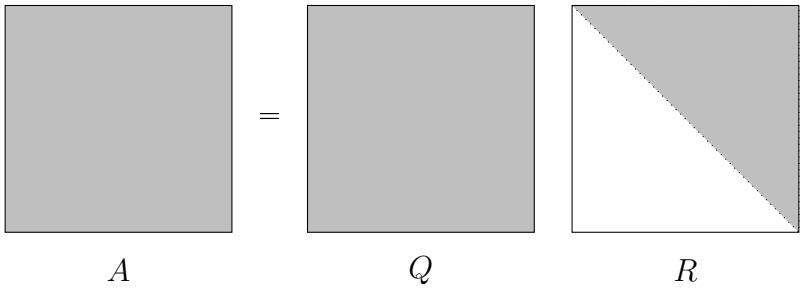
Chapitre 6

Factorisation $Q\,R$

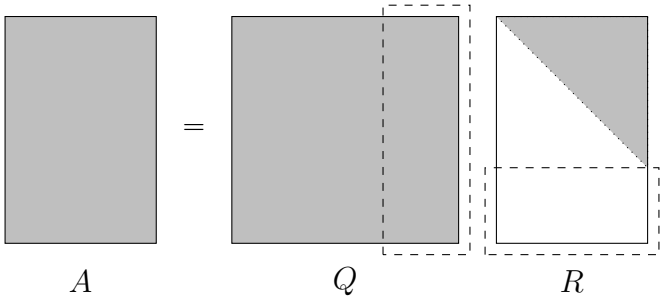
Calculer une factorisation $L\,U$ d'une matrice A permet de résoudre facilement les systèmes $A\,x = b$, parce que L et U sont triangulaires. Dans ce chapitre, on étudie la factorisation $Q\,R$, qui permet aussi de résoudre les systèmes $A\,x = b$, parce que Q^{-1} s'obtient sans calcul (Q est orthogonale) et R est quasiment une matrice triangulaire (elle est trapézoïdale supérieure). Il suffit donc de résoudre $R\,x = Q^{-1}\,b$ par l'algorithme de substitution avant ou arrière.

Une factorisation $Q\,R$ est deux fois plus coûteuse à calculer qu'une décomposition $L\,U$ mais on dispose d'algorithmes qui sont stables dans tous les cas (Householder). Elle a de nombreuses applications, bien au delà de la résolution de systèmes $A\,x = b$ avec A inversible.

Dans le cas d'une matrice A inversible, de dimension $m \times m$, une factorisation $Q\,R$ de A ressemble au schéma suivant :



Dans ce cours, on considèrera le cas plus général et très important en pratique d'une matrice A de dimension $m \times n$ avec $m > n$. La factorisation ressemble au schéma suivant :



Oublions temporairement les pointillés. La factorisation $Q\,R$ de A est dite alors *complète*. La matrice R est trapézoïdale supérieure. La matrice Q est carrée. On verra qu'elle est orthogonale et donc inversible. Toutefois, les parties en pointillé ne sont pas très intéressantes puisqu'elles correspondent à des lignes de zéros dans le cas de R et à des colonnes quasiment aléatoires dans le cas de Q. Pour cette raison, la plupart des algorithmes ne les calculent pas. La factorisation $Q\,R$ de A est dite alors *réduite*. Inconvénient de la factorisation réduite : la matrice Q n'étant plus carrée, n'est donc plus inversible, ce qui complique les énoncés. Par contre, l'ensemble de ses colonnes reste un ensemble orthonormal.

Voici un exemple avec Python. On commence par calculer la factorisation $Q\,R$ d'une matrice carrée. On calcule ensuite la factorisation réduite et la factorisation complète d'une matrice rectangulaire.

```
>>> import scipy.linalg as nla
>>> import numpy as np
>>> A = np.array([[4,-6,11],[2,15,7],[4,12,-43]])
```

La fonction `qr` de `scipy.linalg` retourne un couple formé de la matrice Q et de la matrice R.

```
>>> Q,R = nla.qr (A)
>>> Q
array([[-0.66666667,  0.66666667, -0.33333333],
       [-0.33333333, -0.66666667, -0.66666667],
       [-0.66666667, -0.33333333,  0.66666667]])
>>> R
array([[ -6.,  -9.,  19.],
       [  0., -18.,  17.],
       [  0.,   0., -37.]])
```

On vérifie que Q est orthogonale : le produit $Q^T\,Q$ est égal à l'identité, si on considère qu'un flottant de l'ordre de 10^{-16} vaut zéro.

```
>>> np.dot (np.transpose(Q),Q)
array([[ 1.00000000e+00, -1.23358114e-17,  6.16790569e-17],
       [-1.23358114e-17,  1.00000000e+00, -1.72701359e-16],
       [ 6.16790569e-17, -1.72701359e-16,  1.00000000e+00]])
```

On vérifie que le produit $Q\,R$ donne bien A.

```
>>> np.dot(Q,R)
array([[  4.,  -6.,  11.],
       [  2.,  15.,   7.],
       [  4.,  12., -43.]])
```

Pour traiter le cas d'une matrice rectangulaire, on ne garde que les deux premières colonnes de A

```
>>> A = A[:,0:2]
>>> A
array([[ 4, -6],
       [ 2, 15],
       [ 4, 12]])
```

Par défaut, la fonction `qr` retourne la factorisation complète de A

```
>>> Q,R = nla.qr (A)
>>> Q
array([[-0.66666667,  0.66666667, -0.33333333],
       [-0.33333333, -0.66666667, -0.66666667],
       [-0.66666667, -0.33333333,  0.66666667]])
>>> R
array([[ -6.,  -9.],
       [  0., -18.],
       [  0.,   0.]])
```

On obtient la factorisation réduite en passant le paramètre `mode='economic'`.

```
>>> Q,R = nla.qr (A,mode='economic')
>>> Q
array([[-0.66666667,  0.66666667],
       [-0.33333333, -0.66666667],
       [-0.66666667, -0.33333333]])
>>> R
array([[ -6.,  -9.],
       [  0., -18.]])
```

Même dans le cas de la factorisation réduite, on a bien $A = Q\,R$.

```
>>> np.dot(Q,R)
array([[ 4., -6.],
       [ 2., 15.],
       [ 4., 12.]])
```

6.1 Matrices orthogonales

Deux vecteurs x et y sont dits *orthogonaux* si $x^T\,y = 0$. Une famille de vecteurs S est dite *orthogonale* si ses éléments sont non nuls et orthogonaux deux-à-deux. Elle est dite *orthonormale* si, en plus, la norme euclidienne (norme 2) de chacun de ses éléments vaut 1.

Une matrice A est dite *orthogonale* si elle est inversible et $A^T = A^{-1}$. La transposée d'une matrice orthogonale est donc aussi une matrice orthogonale.

Proposition 7 *L'ensemble des colonnes d'une matrice orthogonale est une famille orthonormale.*

Preuve. Supposons A orthogonale et notons $S = \{a_1, a_2, \ldots, a_m\}$ la famille de ses vecteurs colonnes. Comme A est orthogonale, $A^T\,A = I$. On a donc $I_{ij} = a_i^T\,a_j$. Comme $I_{ij} = 0$ si $i \neq j$, la famille S est orthogonale. Comme $I_{ii} = a_i^T\,a_i = 1$ et que $a_i^T\,a_i = (\|a_i\|_2)^2$, on voit que la norme euclidienne de chaque colonne vaut 1 et que la famille S est orthonormale. □

Question 19. [Utile au chapitre 7]. Montrer que si x est un vecteur quelconque et Q est une matrice orthogonale, alors $\|x\|_2 = \|Q\,x\|_2$ (utiliser la formule qui lie la norme euclidienne d'un vecteur au produit scalaire de ce vecteur par lui-même).

6.2 Le procédé d'orthogonalisation de Gram-Schmidt

Cette section doit beaucoup à [1, Lecture 7]. Le procédé de Gram-Schmidt est déconseillé en pratique parce que instable. Il a l'avantage d'être simple. La proposition suivante [1, Lecture 2] donne l'idée clef.

Proposition 8 *Soit $S = \{q_1, q_2, \ldots, q_m\}$ une famille orthonormale et v un vecteur quelconque. Alors le vecteur (les expressions $q_i^T\,v$ sont des produits scalaires)*

$$r \;=\; v - (q_1^T\,v)\,q_1 - (q_2^T\,v)\,q_2 - \cdots - (q_m^T\,v)\,q_m$$

est orthogonal à S.

Preuve. Il suffit de montrer que, quel que soit $1 \leqslant i \leqslant m$, on a $q_i^T\,r = 0$. Comme le produit par un scalaire est commutatif, on a :

$$r \;=\; v - q_1\,(q_1^T\,v) - q_2\,(q_2^T\,v) - \cdots - q_m\,(q_m^T\,v)\,.$$

Soit $1 \leqslant i \leqslant m$ un indice. Multiplions les deux membres de cette égalité par q_i^T. Comme S est orthogonale, les expressions $q_i^T\,q_j$ sont nulles pour $i \neq j$. Comme S est orthonormale, le produit scalaire $q_i^T\,q_i$ vaut 1. Par conséquent, $q_i^T\,r = q_i^T\,v - q_i^T\,v = 0$. □

Le vecteur r n'est pas forcément de longueur 1 mais, s'il est non nul, on peut calculer $q_{m+1} = r/\|r\|_2$ et l'ajouter à S pour obtenir une nouvelle famille orthonormale. On cherche un algorithme qui calcule une factorisation :

$$\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} = \begin{pmatrix} q_1 & q_2 & \cdots & q_m \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix}.$$

On a donc :

$$\begin{aligned} a_1 &= r_{11}\, q_1 \,, \\ a_2 &= r_{12}\, q_1 + r_{22}\, q_2 \,, \\ &\vdots \\ a_n &= r_{1n}\, q_1 + r_{2n}\, q_2 + \cdots + r_{nn}\, q_n \,. \end{aligned} \tag{6.1}$$

On en déduit des expressions pour les n premières colonnes de Q :

$$\begin{aligned} q_1 &= a_1 / r_{11} \,, \\ q_2 &= (a_2 - r_{12}\, q_1) / r_{22} \,, \\ &\vdots \\ q_n &= (a_n - r_{1n}\, q_1 - r_{2n}\, q_2 - \cdots - r_{n-1,n}\, q_{n-1}) / r_{nn} \,. \end{aligned} \tag{6.2}$$

En combinant les formules ci-dessus avec la proposition 8, on conclut que $r_{ij} = q_i^T\, a_j$ quand $i \neq j$ et que r_{ii} est le coefficient qui sert à normaliser les longueurs des vecteurs. Une fois mis en forme, ces raisonnements conduisent à l'algorithme de Gram-Schmidt, dont la version classique est donnée Figure 6.1. Elle calcule une factorisation $Q\,R$ réduite.

```
function Gram-Schmidt (instable)
  for j = 1 to n do
    v = a_j
    for i = 1 to i − 1 do
      r_ij = q_i^T a_j
      v = v − r_ij q_i
    end do
    r_jj = ‖v‖_2
    q_j = v/r_jj
  end do
end
```

FIGURE 6.1 – La version classique de l'algorithme de Gram-Schmidt

La seule chose qui puisse faire échouer cet algorithme, c'est d'obtenir un vecteur v nul à l'une des itérations. La proposition suivante nous éclaire sur ce point.

Proposition 9 *Si la matrice A est de rang n (par exemple, si A est carrée et inversible), alors le vecteur v n'est jamais nul.*

> **Preuve.** Un examen rapide des formules (6.1) et (6.2) montre que, quel que soit j, le vecteur q_j est une combinaison linéaire des vecteurs $\{a_1, \ldots, a_{j-1}\}$ et que le vecteur a_j est une combinaison linéaire des vecteurs $\{q_1, \ldots, q_{j-1}\}$. Par conséquent, si v est nul

à une itération j, c'est que a_j est une combinaison linéaire des vecteurs $\{a_1, \ldots, a_{j-1}\}$ et que le rang de A est strictement inférieur à n. Cette contradiction avec l'hypothèse montre que v n'est jamais nul. □

Question 20. Supposons A carrée et inversible. La matrice Q construite par l'algorithme de Gram-Schmidt est alors carrée. Il reste à démontrer qu'elle est orthogonale. Pour cela, il suffit de montrer que si $\{q_1, q_2, \ldots, q_m\}$ est une famille orthonormale de m vecteurs de dimension m, alors Q est orthogonale. Cette réciproque de la proposition 7 est laissée en exercice (elle se démontre avec exactement les mêmes arguments).

En adaptant un peu l'algorithme de Gram-Schmidt pour traiter le cas où v est nul, on peut montrer que toute matrice A de dimension $m \times n$, avec $m \geqslant n$ admet une factorisation $Q\,R$ [1, Theorem 7.1].

Proposition 10 *Si A est de rang n alors A a une unique factorisation $Q\,R$ réduite telle que $r_{ii} > 0$.*

Voir [1, Theorem 7.2]. La matrice A peut aussi avoir des factorisations $Q\,R$ avec $r_{ii} < 0$. Si A n'est pas de rang n, certains r_{ii} sont nuls.

6.3 La méthode de Householder

D'un point de vue mathématique, la méthode de Householder consiste à multiplier A, à gauche, par une suite de n matrices orthogonales Q_k, de telle sorte que le résultat

$$Q_n\, Q_{n-1} \cdots Q_1\, A = R$$

soit une matrice trapézoïdale supérieure. La matrice Q recherchée est définie par

$$Q^T = Q_n\, Q_{n-1} \cdots Q_1\,.$$

Proposition 11 *Un produit de matrices orthogonales est une matrice orthogonale.*

Preuve. Soit $Q = Q_n\, Q_{n-1} \cdots Q_1$ un produit de matrices orthogonales. La transposée de ce produit est $Q^T = Q_1^T \cdots Q_{n-1}^T\, Q_n^T$. Comme les matrices Q_i sont orthogonales, chaque produit $Q_i\, Q_i^T$ est egal à l'identité, le produit $Q\, Q^T$ est donc égal l'identité et Q est orthogonale. □

Proposition 12 *Soit v un vecteur non nul. La matrice F suivante est orthogonale :*

$$F \;=\; I - 2\,\frac{v\, v^T}{v^T\, v}\cdot \tag{6.3}$$

Le facteur 2 est très important ; au numérateur, le produit tensoriel $v\, v^T$ donne une matrice ; au dénominateur, le produit $v^T\, v$ est un scalaire : c'est le carré de la norme euclidienne de v, qui est non nul puisque v est non nul.

Preuve. Pour montrer que F est orthogonale, il suffit de montrer que $F^T F$ est l'identité. La matrice F est égale à sa transposée (utiliser les règles pour la transposée et le fait que le dénominateur est un scalaire). On a donc :

$$F^T F = F F = I - 4\,\frac{v\,v^T}{v^T\,v} + \left(2\,\frac{v\,v^T}{v^T\,v}\right)^2 .$$

En se rappelant à nouveau que $v^T\,v$ est un scalaire et que la multiplication par un scalaire est commutative, on trouve que $F^T F$ est l'identité et donc que F est orthogonale. □

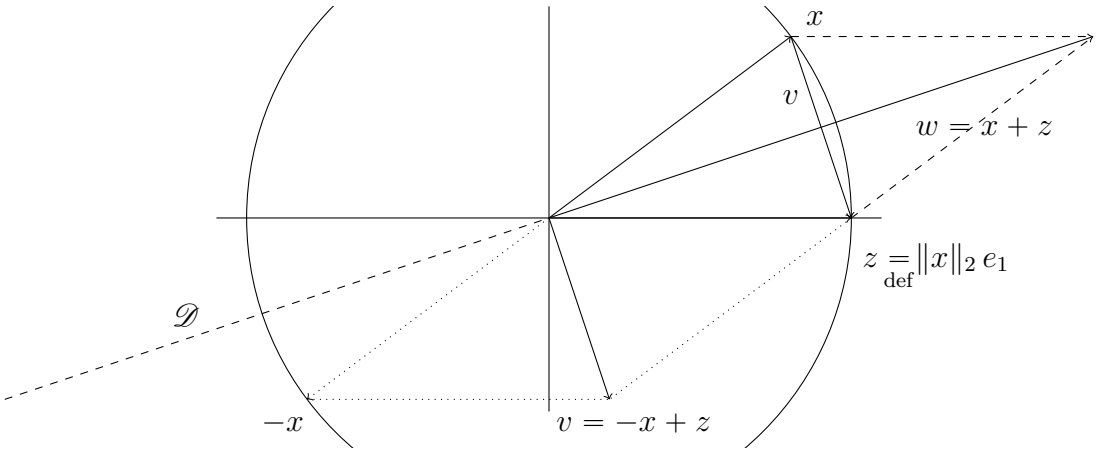


FIGURE 6.2 – Les vecteurs x et $\|x\|_2\,e_1$ (nommons-le z) ont même longueur. Le vecteur $v = -x+z$ s'obtient graphiquement en ajoutant $-x$ à z. La droite $\mathscr{D}$ a pour vecteur directeur le vecteur $w = z + x$. Dans le plan, des arguments de géométrie élémentaire montrent que v et w sont orthogonaux. En dimension m, on vérifie facilement que $v^T\,w = 0$. La matrice F de la formule (6.4) envoie chaque point sur son image, par rapport au « miroir » que constitue l'hyperplan $\mathscr{D}$ des vecteurs orthogonaux à v, d'où son nom de « matrice de réflexion ».

Proposition 13 *Soient x un vecteur non nul quelconque, e_1 le vecteur ayant un 1 en première position et des zéros partout ailleurs, $v = \pm\|x\|_2\,e_1 \pm x$ et F la matrice suivante, qui est donc un cas particulier de* (6.3) *:*

$$F \;=\; I - 2\,\frac{v\,v^T}{v^T\,v}\cdot \tag{6.4}$$

Alors,

$$F\,x = \begin{pmatrix} \|x\|_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Preuve. On fait la démonstration dans le cas illustré Figure 6.2. Notons $z = \|x\|_2\, e_1$ et montrons que $F\,x = z$. On introduit les vecteurs $v = -x+z$ et $w = x+z$. On commence par remarquer que v et w sont orthogonaux. En effet, $v^T\,w = (-x^T + z^T)\,(x+z) = -x^T\,x - x^T\,z + z^T\,x + z^T\,z = 0$ puisque les deux termes du milieu s'annulent l'un l'autre et que les vecteurs x et z ont même longueur. Ensuite, en utilisant le fait que $x = -v+z$, on remarque que $v^T\,w = v^T\,(-v+2\,z) = 0$. On en déduit que $v^T\,z/(v^T\,v) = 1/2$. En développant le produit $F\,x$, on obtient :

$$\left(I - 2\,\frac{v\,v^T}{v^T\,v}\right)\,(-v+z) = -v + z + 2\,v\,\underbrace{\frac{v^T\,v}{v^T\,v}}_{=1} - 2\,v\,\underbrace{\frac{v^T\,z}{v^T\,v}}_{=\frac{1}{2}} = z\,.$$

□

Si on multiplie un vecteur quelconque y par F, on obtient un vecteur $F\,y$ qui correspond à la « réflexion » du vecteur y par rapport à l'hyperplan des vecteurs orthogonaux à v (voir Figure 6.2). La matrice F est appelée un *réflecteur de Householder*. L'idée de Householder consiste à prendre pour matrices Q_k des matrices de la forme

$$Q_k \;=\; \begin{pmatrix} I & 0 \\ 0 & F \end{pmatrix} \tag{6.5}$$

avec I identité de dimension $(k-1)\times(k-1)$ et F de la forme considérée ci-dessus, de façon à transformer A d'après le diagramme de la figure 6.3.

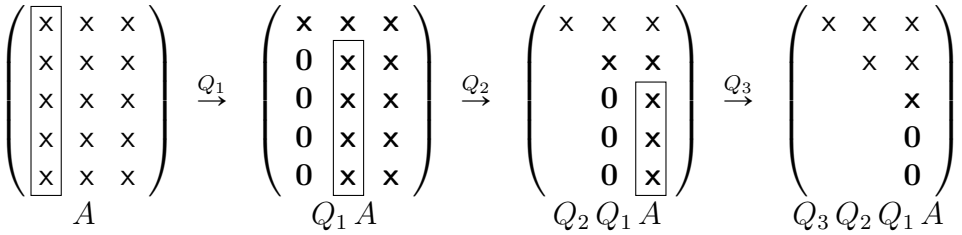


FIGURE 6.3 – Transformations effectuées par l'algorithme de Householder. Les croix sont des nombres non nécessairement nuls, les blancs sont nuls. À chaque itération, la matrice F est formée avec le vecteur x encadré. Les caractères gras indiquent des entrées qui ont été modifiées par la dernière multiplication. La multiplication par Q_2, par exemple, ne modifie pas la première ligne de la matrice $Q_1\,A$. Ces considérations sur les entrées modifiées ou pas sera importante lorsqu'on adaptera la méthode pour obtenir un algorithme de mise sous forme de Hessenberg d'une matrice.

6.3.1 Version mathématique de l'algorithme

On illustre les propositions précédentes ainsi que l'algorithme de Householder en Python. À chaque étape, on construit les matrices Q_i explicitement, ce que les vraies implantations ne font pas.

```
>>> import scipy.linalg as nla
>>> import numpy as np
>>> A1 = np.array([[0,-20,-14],[3,27,-4],[4,11,-2]])

>>> A1
array([[  0, -20, -14],
       [  3,  27,  -4],
       [  4,  11,  -2]])
```

On extrait la première colonne de $A = A_1$.

```
>>> x1 = A1[:,0]
>>> x1
array([0, 3, 4])
```

On forme v_1 en ajoutant la norme deux de x_1 à sa première coordonnée.

```
>>> v1 = x1 + np.array([nla.norm(x1,2),0,0])
>>> v1
array([5., 3., 4.])
```

On peut maintenant former la première matrice de réflexion. La fonction `eye` de `numpy` permet de former une matrice identité. La fonction `outer` permet de calculer le produit tensoriel $v^T\, v$.

```
>>> Q1 = np.eye(3) - (2 / np.dot(v1,v1)) * np.outer(v1,v1)
>>> Q1
array([[ 0.  , -0.6 , -0.8 ],
       [-0.6 ,  0.64, -0.48],
       [-0.8 , -0.48,  0.36]])
```

On obtient la matrice $A_2 = Q_1\, A_1$.

```
>>> A2 = np.dot (Q1,A1)
>>> A2
array([[ -5. , -25. ,   4. ],
       [  0. ,  24. ,   6.8],
       [  0. ,   7. ,  12.4]])
```

À l'itération suivante, le vecteur x_2 est un vecteur de dimension 2 extrait de la deuxième colonne de A_2.

```
>>> x2 = A2[1:,1]
>>> x2
array([24.,  7.])
```

On forme v_2 en ajoutant la norme deux de x_2 à sa première coordonnée.

```
>>> v2 = x2 + np.array([nla.norm(x2,2),0])
>>> v2
array([49.,  7.])
```

On forme la matrice de réflexion F

```
>>> F = np.eye(2) - (2 / np.dot(v2,v2)) * np.outer(v2,v2)
>>> F
array([[-0.96, -0.28],
       [-0.28,  0.96]])
```

Pour obtenir la matrice Q_2, il suffit d'insérer F en bas à droite d'une matrice identité 3×3.

```
>>> Q2 = np.eye (3)
>>> Q2[1:,1:] = F
>>> Q2
array([[ 1.  ,  0.  ,  0.  ],
       [ 0.  , -0.96, -0.28],
       [ 0.  , -0.28,  0.96]])
```

On obtient la matrice $R = Q_2\, A_2$ qui est bien triangulaire.

```
>>> R = np.dot (Q2, A2)
>>> R
array([[-5.0000000e+00, -2.5000000e+01,  4.0000000e+00],
       [ 0.0000000e+00, -2.5000000e+01, -1.0000000e+01],
       [ 0.0000000e+00, -8.8817842e-16,  1.0000000e+01]])
```

Grâce aux matrices Q_i on peut former $Q = (Q_2\, Q_1)^T$.

```
>>> Q = np.transpose(np.dot(Q2,Q1))
>>> Q
array([[ 0.  ,  0.8 , -0.6 ],
       [-0.6 , -0.48, -0.64],
       [-0.8 ,  0.36,  0.48]])
```

On vérifie que Q est orthogonale et que $A = Q\, R$.

```
>>> np.dot(np.transpose(Q),Q)
array([[ 1.0000000e+00, -8.2600593e-17,  8.8817842e-19],
       [-8.2600593e-17,  1.0000000e+00, -2.1938007e-17],
       [ 8.8817842e-19, -2.1938007e-17,  1.0000000e+00]])
>>> np.dot(Q,R)
array([[  0., -20., -14.],
       [  3.,  27.,  -4.],
       [  4.,  11.,  -2.]])
```

Les éléments diagonaux de R sont non nuls parce que A est de rang maximal. En utilisant l'opposé de certaines des formules qui donnent v à partir de x, on aurait pu assurer qu'ils soient positifs.

6.3.2 Version praticable de l'algorithme

La figure 6.4 contient le pseudo-code [1, Algorithm 10.1] d'un algorithme de factorisation $Q\,R$ fondé sur l'utilisation des matrices de réflexion de Householder. Une implantation en

Factorisation $Q\,R$ de Householder
En entrée : la matrice A de dimension $m \times n$.
En sortie : A contient R et les vecteurs $v_1, v_2, \ldots, v_n$ contiennent un codage de Q.
Notation : $A_{k:m,k}$ désigne le vecteur des éléments de A, lignes k à n et colonne k.
De même, $A_{k:m,k:n}$ désigne la sous-matrice de A, lignes k à m et colonnes k à n.
begin
for $k = 1$ to n do
$x = A_{k:m,k}$
$v_k = \mathrm{sign}\,(x_1)\,\|x\|_2\,e_1 + x$
$v_k = v_k / \|v_k\|_2$
$A_{k:m,k:n} = A_{k:m,k:n} - 2\,v_k\,(v_k^T\,A_{k:m,k:n})$
end do
end

FIGURE 6.4 – L'algorithme de factorisation $Q\,R$ de Householder

FORTRAN est donnée ci-dessous.

```
      SUBROUTINE HOUSEHOLDER (M, N, A, V)
      INTEGER M, N
      DOUBLE PRECISION A(M,N), V(M,N)
      DOUBLE PRECISION DNRM2, NRMV, W(N)
      INTEGER K, I, J
* Pour k variant de 1 à N
      DO K = 1,N
* Stocker Vk dans V(K..M,K). Initialisation à x = A(K..M,K)
         DO I = K,M
            V(I,K) = A(I,K)
         END DO
* Ajouter +/- ||Vk|| au 1er élément de Vk
         NRMV = DNRM2 (M-K+1, V(K,K), 1)
         V(K,K) = V(K,K) + SIGN (NRMV, V(K,K))
* Vk = Vk / ||Vk||
         NRMV = DNRM2 (M-K+1, V(K,K), 1)
         DO I = K,M
            V(I,K) = V(I,K) / NRMV
         END DO
* W = le vecteur ligne Vk**T . A(K..M,K..N)
         DO J = K,N
            W(J) = 0D0
```

```
            DO I = K,M
               W(J) = W(J) + V(I,K)*A(I,J)
            END DO
         END DO
* A(K..M,K..N) = A(K..M,K..N) - 2 * Vk . W
*              = A(K..M,K..N) - 2 * Vk . Vk**T . A(K..M,K..N)
         DO J = K,N
            DO I = K,M
               A(I,J) = A(I,J) - 2DO*V(I,K)*W(J)
            END DO
         END DO
      END DO
      END
```

Voici quelques remarques sur le code FORTRAN.

L'expression `SIGN(A,B)` retourne A si $B \geqslant 0$ et $-A$ sinon. Mathématiquement, il est inutile de se préoccuper du signe mais, numériquement, on fait en sorte d'ajouter $\|x\|_2$ à x_1 si x_1 est positif et de soustraire $\|x\|_2$ à x_1 sinon, afin d'éviter les pertes de précision dues de la soustraction de deux nombres de valeurs comparables (cf. chapitre 1).

La fonction BLAS `DNRM2` permet de calculer la norme euclidienne d'un vecteur.

On ne construit pas explicitement les matrices Q_k mais on mémorise le vecteur v_k à chaque étape dans la colonne k d'un tableau V, entre les indices k et m.

À la fin, de l'exécution, la matrice A contient R.

Mais où est Q ? On ne l'a pas explicitement mais, grâce à V, on peut facilement calculer $Q\,x$ pour tout vecteur x. Par conséquent, il est facile d'obtenir Q en calculant $Q\,I$, colonne par colonne ! Un algorithme en pseudo-code pour calculer le produit $Q\,x$ est donné figure 6.5 [1, Algorithm 10.3].

Calcul de $Q\,x$
En entrée : les vecteurs $v_1, v_2, \ldots, v_n$ contenant un codage de Q et un vecteur x.
begin
  for $k = n$ downto 1 do
    $x_{k:m} = x_{k:m} - 2\,v_k\,(v_k^T\,x_{k:m})$
  end do
end

FIGURE 6.5 – Algorithme pour calculer $Q\,x$. Le vecteur résultat est dans x.

Une implantation FORTRAN est donnée ci-dessous

```
      SUBROUTINE MULQX (M, N, V, X)
      INTEGER M, N
      DOUBLE PRECISION V(M,N), X(M)
      DOUBLE PRECISION W
```

```
      INTEGER I, K
* pour k variant de N à 1 (pas = -1)
      DO K = N,1,-1
* X(K..M) = X(K..M) - 2 * Vk . Vk**T . X(K..M)
         W = 0D0
         DO I = K,M
            W = W + V(I,K)*X(I)
         END DO
         DO I = K,M
            X(I) = X(I) - 2D0*V(I,K)*W
         END DO
      END DO
      END
```

Les mêmes multiplications, mais en sens inverse, permettent de calculer $Q^T x$. Un algorithme en pseudo-code est donné figure 6.6 [1, Algorithm 10.2].

Calcul de $Q^T x$
En entrée : les vecteurs $v_1, v_2, \ldots, v_n$ contenant un codage de Q et un vecteur x.
begin
  for $k = 1$ to n do
    $x_{k:m} = x_{k:m} - 2\, v_k\, (v_k^T\, x_{k:m})$
  end do
end

FIGURE 6.6 – Algorithme pour calculer $Q^T x$.

Question 21. Réécrire les algorithmes de cette section en utilisant les BLAS. Utiliser les fonctions `DGEMV` et `DGER`.

Stabilité et complexité

L'algorithme de Householder est *backward stable*[1] [1, Theorem 16.1].

La complexité de l'algorithme de Householder peut se calculer avec MAPLE. On trouve que $f(m,n)$ est de l'ordre de $(5/2)\, m\, n^2 - (5/6)\, n^3$ en général et $(5/3)\, m^3$ dans le cas particulier d'une matrice carrée. C'est plus de deux fois plus que le pivot de Gauss.

```
# Le coût des deux boucles les plus intérieures, ensemble
> h := sum (5, i=k..m);
                              h := 5 m - 5 k + 5
```

1. Même remarque que pour l'algorithme de Cholesky : soient $\tilde{Q}$ et $\tilde{R}$ les matrices calculées. La matrice $\tilde{Q}\,\tilde{R}$ est proche de A mais les matrices $\tilde{Q}$ et $\tilde{R}$ ne sont pas nécessairement proches de Q et de R. Voir [1, Lecture 16].

```
# Le coût de la norme 2 d'un vecteur de dimension m-k+1
> n2 := 2*(m-k+1);
                              n2 := 2 m - 2 k + 2

# Le coût de la boucle principale (deux normes 2, une normalisation
# plus les deux boucles finales)
> g := 2*n2 + sum (1, i=k..m) + sum (h, j=k..n);
               g := 5 m - 5 k + 5 + (n - k + 1) (5 m - 5 k + 5)

> f := expand (sum (g, k=1..n));
                                              2          2        3
             f := 15/2 n m + 10/3 n + 5/2 n  m - 5/2 n  - 5/6 n

# dans le cas m = n
> subs (n=m, f);
                                2                 3
                            5 m  + 10/3 m + 5/3 m
```

Voici le calcul de la complexité de la multiplication par Q d'un vecteur x. On trouve $f(m,n)$ de l'ordre de $5\,n\,m - (5/2)\,n^2$ en général et $(5/2)\,m^2$ dans le cas particulier d'une matrice carrée.

```
# Le coût des deux boucles intérieures, ensemble
> g := sum (5, i=k..m);
                              g := 5 m - 5 k + 5

> f := expand (sum (g, k=1..n));
                                          2
                         f := 5 n m - 5/2 n  + 5/2 n

> subs (n=m, f);
                                    2
                                5/2 m  + 5/2 m
```

En LAPACK

La fonction `DGEQRF` de LAPACK applique une variante de la méthode de Householder : les coefficients diagonaux r_{jj} sont positifs alors que dans la version donnée ci-dessus, ils peuvent être négatifs. À la fin de l'exécution de ce sous-programme, la matrice R est rangée dans la partie supérieure de A et les vecteurs v_k sont stockés dans la partie inférieure de A et dans un tableau supplémentaire (il n'y a pas suffisamment de place dans A).

Après application de `DGEQRF`, la fonction `DORGQR` permet de calculer la matrice Q et la fonction `DORMQR` fournit l'algorithme de multiplication par Q ou par sa transposée.

En Python

La fonction LAPACK `DGEQRF` peut être appelée depuis Python en passant le paramètre `mode='raw'` à la fonction `qr`. Malheureusement, aucune interface n'est prévue pour permettre la réutilisation de son résultat via une fonction telle que `DORMQR`.

La solution de facilité consiste donc à utiliser les fonctionmalités moins optimisées de la fonction `qr` de `scipy.linalg`.

Mais si on le souhaite vraiment, on peut aussi programmer une fonction d'interface FORTRAN et l'appeler depuis Python grâce à `f2py3`. Un exemple d'une telle fonction est donné ci-dessous :

```
* Affecte au vecteur B le produit Q**T . B où Q est la matrice orthogonale
* issue de la factorisation QR complète d'une matrice A
*
* ABAR et TAU contiennent le résultat de DGEQRF appliqué à A
* En Python : (Abar,tau),R = nla.qr(A,mode='raw')

      SUBROUTINE MUL_QT (M, K, ABAR, TAU, B)
* f2py3 intent(inplace) B
      INTEGER M, K
      DOUBLE PRECISION ABAR(M,K), TAU(K), B(M)
* variable locale
      DOUBLE PRECISION WORK(M)
      INTEGER INFO
*
      CALL DORMQR ('L', 'T', M, 1, K, ABAR, M, TAU, B, M, WORK, M, INFO)
      IF (INFO .NE. 0) STOP 'Erreur dans DORMQR'
      END SUBROUTINE
```

Pour l'utiliser il suffit de placer ce code dans un fichier `mul_QT.f`, d'ajouter le nom de ce fichier à la variable `FORTRAN_FILES` du `Makefile` de la figure 3.1, page 31 et de lancer le `Makefile`. Cela fait, le code suivant montre comme résoudre un système d'équations linéaires en utilisant la méthode décrite dans la section 6.3.3. On reprend l'exemple de la Section 4.3.2, page 40.

```
>>> import scipy.linalg as nla
>>> import numpy as np
```

La premières des commandes ci-dessous importe le paquetage `mon_code_fortran` compilé par `f2py3`.

```
>>> import mon_code_fortran as mcf
>>> A = np.array([[4,6,-10],[6,25,-47],[-10,-47,125]],dtype=np.float64,order='F')
>>> b = np.array([42,175,-401],dtype=np.float64)
```

Exemple d'utilisation de l'option `mode='raw'` qui appelle `DGETRF` en LAPACK. La matrice A contient la matrice R dans la partie triangulaire supérieure. La partie inférieure, combinée avec le contenu du tableau `tau`, contient un codage des matrices de réflexion.

```
>>> (A,tau),R = nla.qr(A,overwrite_a=True,mode='raw')
>>> A
array([[-12.32882801, -52.2352976 , 127.50603701],
       [  0.36744829, -11.89427107,  37.80679018],
       [ -0.61241382,  -0.73154604,  15.7116881 ]])
>>> tau
array([1.32444284, 1.30279613, 0.        ])
>>> R
array([[-12.32882801, -52.2352976 , 127.50603701],
       [  0.        , -11.89427107,  37.80679018],
       [  0.        ,   0.        ,  15.7116881 ]])
```

On remplace b par $Q^T\,b$ en appelant `DORMQR`.

```
>>> mcf.mul_qt(A,tau,b)
>>> b
array([-424.04679484, -111.29639355,  -31.42337619])
```

Il ne reste plus qu'à résoudre $R\,x = Q^T\,b$. Le résultat est dans b.

```
>>> nla.solve_triangular(A, b, lower=False, overwrite_b=True)
array([ 1.,  3., -2.])
```

6.3.3 Utilisation pour résoudre $A\,x = b$

Supposons A inversible (et donc carrée). Soit $A = Q\,R$ une factorisation de A. En multipliant les deux membres de l'équation par Q^T, on trouve que mathématiquement, il suffit de résoudre :

$$R\,x = Q^T\,b\,.$$

Algorithmiquement, il suffit donc de :

1. appliquer le sous-programme `HOUSEHOLDER` sur A (on obtient R et les vecteurs v_k),
2. multiplier b, à gauche, par Q^T, en utilisant les vecteurs v_k (sous-programme `MULQTX`),
3. en déduire x, en utilisant R, avec l'algorithme de substitution avant ou arrière.

Cette méthode de résolution de systèmes d'équations linéaires [1, Algorithm 16.1] est *backward stable* [1, Theorem 16.2]. Notons $\tilde{x}$ la solution calculée, par opposition à la solution mathématique x. Il existe une constante $c > 0$ telle que [1, Theorem 16.3] :

$$\frac{\|\tilde{x} - x\|}{\|x\|} \leqslant c\,\kappa(A)\,\varepsilon_{\text{machine}}\,,$$

où $\kappa(A)$ désigne la condition de A.

Bibliographie

[1] Lloyd Nicholas Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

Chapitre 7

Problèmes de moindres carrés

7.1 Faire passer une courbe par un ensemble de points

On se donne un ensemble de m points (x_i, y_i). On cherche les coefficients p, q, r de la parabole

$$y = p\,x^2 + q\,x + r$$

qui passe « au plus près » des points. On suppose qu'on dispose d'un nombre suffisant de points : $m \geqslant 3$. Que veut-on dire exactement par « au plus près » ? Pour chaque triplet de valeurs p, q, r, on considère les m écarts verticaux entre la courbe et les points (x_i, y_i)

$$\textit{écart}_i \underset{\text{def}}{=} p\,x_i^2 + q\,x_i + r - y_i\,, \qquad 1 \leqslant i \leqslant m$$

et on définit l'erreur comme la somme des carrés des écarts :

$$\textit{erreur} = \textit{écart}_1^2 + \textit{écart}_2^2 + \cdots + \textit{écart}_m^2\,.$$

On cherche les coefficients p, q, r pour lesquels l'erreur (la somme des carrés des écarts verticaux) est minimale. La recette est la suivante. On pose que la parabole passe par les m points :

$$\begin{aligned} p\,x_1^2 + q\,x_1 + r &= y_1\,, \\ p\,x_2^2 + q\,x_2 + r &= y_2\,, \\ &\vdots \\ p\,x_m^2 + q\,x_m + r &= y_m\,. \end{aligned}$$

Les équations ci-dessus forment un système d'équations linéaires qui n'a pas de solution, en général :

$$\underbrace{\begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots \\ x_m^2 & x_m & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} p \\ q \\ r \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}}_{b}.$$

Les coefficients p, q, r qui minimisent l'erreur sont la solution du système suivant (système des *équations normales*) :

$$A^T\,A\,x = A^T\,b\,.$$

Ce système peut se résoudre grâce pivot de Gauss. D'après la proposition suivante, il peut aussi se résoudre par la méthode de Cholesky.

Proposition 14 *Soit A une matrice $m \times n$ avec $m \geqslant n$. Si le rang de A est n alors la matrice $A^T\,A$ est définie positive.*

> **Preuve.** Pour commencer, on rappelle que le produit d'une matrice par sa transposée est une matrice symétrique (question 4, page 17). On suppose que le rang de A est n. On montre que, quel que soit le vecteur x non nul, $x^T\,A^T\,A\,x > 0$. Comme $x^T\,A^T\,A\,x = (\|A\,x\|_2)^2$, il suffit de montrer que le vecteur $A\,x$ est non nul. Or comme le rang de A vaut n, les colonnes de A sont linéairement indépendantes et $A\,x$ ne peut être nul que si x est nul (cf. question 7, page 20). □

Question 22. Qu'est-ce qui peut empêcher le rang de valoir n ?

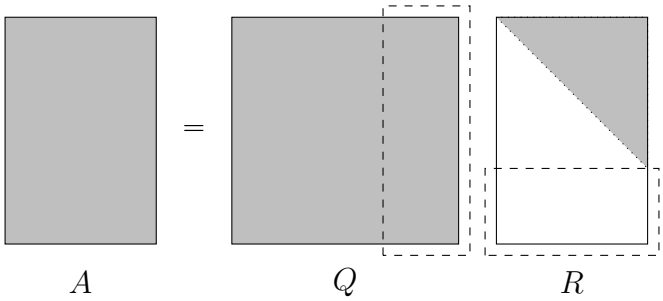
7.2 Moindres carrés et factorisation $Q\,R$

La factorisation $Q\,R$ fournit non seulement un meilleur algorithme pour résoudre les problèmes de moindres carrés (via la méthode de Householder), mais elle permet de prouver la recette donnée plus haut.

On considère un système linéaire surdéterminé $A\,x = b$ (avec $m \geqslant n$, donc). Dans le cas général, ce système n'a aucune solution mais, à tout vecteur x, on peut associer une erreur qui s'écrit matriciellement :

$$\textit{erreur} = (\|b - A\,x\|_2)^2\ .$$

On cherche, comme ci-dessus, un vecteur x qui minimise cette somme de carrés. La matrice A admet une factorisation $Q\,R$ de la forme suivante :



En partitionnant les matrices suivant les pointillés, on a donc :

$$A = Q\,R = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix}\ .$$

En multipliant les deux membres de l'équation par la transposée de Q (qui est aussi son inverse, puisque Q est orthogonale), on obtient :

$$Q^T\,A\,x = Q^T\,Q\,R\,x = R\,x = Q^T\,b\,.$$

En faisant apparaître le partitionnement :

$$\begin{pmatrix} R_1 \\ 0 \end{pmatrix} x = \begin{pmatrix} Q_1^T \\ Q_2^T \end{pmatrix} b\,.$$

Si on développe ce système d'équations un peu bizarre, on observe qu'il comporte deux parties :

$$R_1\,x = Q_1^T\,b\,, \qquad \text{et} \qquad 0 = Q_2^T\,b\,.$$

C'est un système sans solution (c'est normal, puisqu'il est équivalent à $A\,x = b$, qui n'en a pas non plus). On ne peut rien faire au sujet du système $0 = Q_2^T\,b$ mais on peut résoudre $R_1\,x = Q_1^T\,b$ en utilisant l'algorithme décrit en section 6.3.3.

Proposition 15 *Le vecteur x qui minimise l'erreur $(\|b - A\,x\|_2)^2$ est la solution du système $R_1\,x = Q_1^T\,b$.*

> **Preuve.** Multiplier un vecteur par une matrice orthogonale ne change pas sa norme euclidienne (question 19, page 61). Par conséquent, minimiser $(\|b - A\,x\|_2)^2$ revient à minimiser $(\|Q^T\,(b - A\,x)\|_2)^2 = (\|Q^T\,b - R\,x\|_2)^2$. D'après la question 10, page 20,
>
> $$(\|Q^T\,b - R\,x\|_2)^2 = \underbrace{(\|Q_1^T\,b - R_1\,x\|_2)^2 + (\|Q_2^T\,b\|_2)^2}_{S}\,.$$
>
> Le terme de droite de S ne dépend pas de x. La somme de carrés est donc minimale quand le terme de gauche de S est nul. Le terme de droite représente l'erreur minimale, ou l'erreur en la solution optimale, si on préfère. □

Proposition 16 *Résoudre le système des équations normales $A^T\,A\,x = A^T\,b$ est équivalent à résoudre $R_1\,x = Q_1^T\,b$.*

> **Preuve.** Le système $A^T\,A\,x = A^T\,b$ s'écrit $R^T\,Q^T\,Q\,R\,x = R^T\,Q^T\,b$. En simplifiant le terme de gauche, on trouve $R^T\,R\,x = R^T\,Q^T\,b$. Faisons apparaître le partitionnement :
>
> $$\begin{pmatrix} R_1^T & 0 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} x = \begin{pmatrix} R_1^T & 0 \end{pmatrix} \begin{pmatrix} Q_1^T \\ Q_2^T \end{pmatrix} b\,.$$
>
> Ce système comporte deux parties :
>
> $$R_1^T\,R_1\,x = R_1^T\,Q_1^T\,b \qquad \text{et} \qquad 0 = 0\,.$$
>
> La partie de droite est inutile. La matrice R_1^T est inversible : en effet, parce que A est de rang n, les éléments diagonaux sont non nuls (voir la proposition 10, page 63 et la question 13, page 24). On peut donc multiplier les deux membres du système d'équations de gauche par son inverse et déduire que $R_1\,x = Q_1^T\,b$. □

7.3 Considérations numériques

La méthode qui consiste à former explicitement le système des équations normales et à appliquer la méthode de Cholesky peut être utile si la question de la rapidité de le méthode est primordiale, mais la méthode généralement conseillée est celle qui passe par la factorisation $Q\,R$ calculée avec la technique de Householder [2, Lecture 11, Comparison of Algorithms].

L'algorithme de résolution des problèmes de moindres carrés par la méthode de Householder est *backward stable* [2, Theorem 19.1] alors que la résolution par la méthode des équations normales est instable [2, Theorem 19.3].

La bibliothèque LAPACK offre la fonction `DGELS` pour résoudre les problèmes de moindres carrés.

Question 23. [1, Chapitre IV, exemple 6.2, page 100]. Pour étudier le phénomène de la thermo-électricité, on fait l'expérience suivante. On soude un fil de cuivre avec un fil de constantan de manière à obtenir une boucle fermée. Un point de soudure est maintenu à température fixe ($T_0 \simeq 24$ degrés Celsius) alors que l'on fait varier la température de l'autre. Ceci génère une tension U, qu'on mesure en fonction de T. On suppose que cette dépendance obéit à la loi :

$$U = a + b\,T + c\,T^2\,.$$

On cherche à déterminer les paramètres a, b, c, à partir des mesures données dans le tableau Figure 7.1. Mettre le problème en équations. Écrire un programme FORTRAN qui détermine les paramètres en fonction des mesures. Utiliser la méthode fondée sur la factorisation $Q\,R$.

i	T	U	i	T	U	i	T	U
1	0	−0.89	8	35	0.42	15	70	1.88
2	5	−0.69	9	40	0.61	16	75	2.10
3	10	−0.53	10	45	0.82	17	80	2.31
4	15	−0.34	11	50	1.03	18	85	2.54
5	20	−0.15	12	55	1.22	19	90	2.78
6	25	0.02	13	60	1.45	20	95	3.00
7	30	0.20	14	65	1.68	21	100	3.22

FIGURE 7.1 – Tension mesurée en fonction de la température T

Bibliographie

[1] Ernst Hairer and Gerhard Wanner. Introduction à l'Analyse Numérique. Accessible sur `http://www.unige.ch/~hairer`, juin 2005.

[2] Lloyd Nicholas Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

Chapitre 8

Valeurs propres

8.1 Pourquoi des valeurs propres ?

Les valeurs propres et les vecteurs propres sont des notions qui n'ont de sens que pour des matrices carrées. Elles apparaissent naturellement dans tous les problèmes qui font intervenir des puissances de la matrice du problème, plutôt que la matrice elle-même. Une exception importante : l'analyse en composantes principales.

8.1.1 Modélisation d'un câble coaxial

En physique, il est classique de décrire les caractéristiques d'un mètre de câble coaxial par une matrice 2×2, appelée matrice transférante. Cette matrice se définit à partir d'un quadripole passif, modélisant un circuit RLC (voir Figure 8.1). Les coefficients de la matrice dépendent de la capacité C et de la self L du circuit (pour faire simple, on n'a pas mis de résistance), ainsi que de la pulsation ω (pour fixer les idées, on peut prendre $L = 100\,10^{-2}$ Farad, $C = 10^{-7}$ Henry et $0 \leqslant \omega \leqslant 10^6$ radians par seconde) :

$$A = \begin{pmatrix} -C\,\omega^2\,L + 1 & 2\,i\,L\,\omega - i\,L^2\,\omega^3\,C \\ i\,C\,\omega & -C\,\omega^2\,L + 1 \end{pmatrix}.$$

La matrice A représente un mètre de câble. Pour modéliser n mètres, on l'élève à la puissance n. Bien sûr, il n'est pas difficile d'élever une matrice au carré ou au cube mais on aimerait bien avoir l'expression de A^n en fonction de n. Et on aimerait aussi pouvoir modéliser $n = 3.5$ mètres de câble ! On voit alors l'intérêt de diagonaliser A, c'est-à-dire de

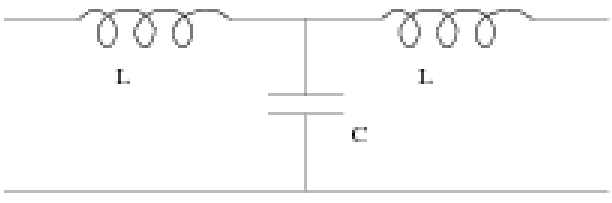


FIGURE 8.1 – Quadripole modélisant un mètre de câble coaxial

factoriser A sous la forme :

$$A = X\,\Lambda\,X^{-1}\,, \qquad \Lambda = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}.$$

En effet, on a alors :

$$A^n = \underbrace{X\,\Lambda\,X^{-1}\,X\,\Lambda\,X^{-1}\,\cdots\,X\,\Lambda\,X^{-1}}_{n \text{ fois}}\,.$$

Les produits $X^{-1}\,X$ se simplifient. On obtient :

$$A^n = X\,\Lambda^n\,X^{-1}\,.$$

Comme Λ est diagonale, la matrice Λ^n a une expression simple, et on trouve :

$$A^n = X \begin{pmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{pmatrix} X^{-1}\,.$$

Le raisonnement a été tenu en supposant implicitement n entier mais on peut montrer que la formule finale est valable pour n réel quelconque.

Synthétisons : pour obtenir une expression de A^n, il est utile de diagonaliser A, c'est-à-dire de factoriser A sous la forme $A = X\,\Lambda\,X^{-1}$ avec Λ diagonale. Les éléments diagonaux de Λ sont les valeurs propres de A. Les colonnes de X sont des vecteurs propres de A.

Le terme « diagonaliser » insiste sur la forme diagonale de Λ mais c'est trompeur : le fait que Λ soit multipliée à gauche et à droite par une matrice X et son inverse est encore plus important. Cela se voit lorsqu'on considère d'autres factorisations qui révèlent les valeurs propres, comme la factorisation de Schur.

8.1.2 Stabilité d'un système dynamique

On considère un système de deux équations différentielles

$$\frac{\mathrm{d}}{\mathrm{d}t}\,\chi_1(t) = a_{11}\,\chi_1(t) + a_{12}\,\chi_2(t)\,, \quad \frac{\mathrm{d}}{\mathrm{d}t}\,\chi_2(t) = a_{21}\,\chi_1(t) + a_{22}\,\chi_2(t)\,, \tag{8.1}$$

où les coefficients a_{ij} sont des réels. Une solution de ce système d'équations est un couple de deux fonctions $\chi_1(t)$ et $\chi_2(t)$. Ce système peut s'écrire sous forme matricielle. Pour faire court, on note $\dot{\chi}_i$ la dérivée par rapport au temps de la fonction $\chi_i(t)$:

$$\begin{pmatrix} \dot{\chi}_1 \\ \dot{\chi}_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} \chi_1 \\ \chi_2 \end{pmatrix},$$

ou encore plus court :

$$\dot{x} = A\,x \qquad \text{avec} \qquad x = \begin{pmatrix} \chi_1 \\ \chi_2 \end{pmatrix}.$$

Numériquement, il est possible de calculer une approximation du graphe d'une solution, sous la forme d'une suite de vecteurs (x_n), à partir d'un vecteur de conditions initiales $x_0 \in \mathbb{R}^2$ donné. La méthode la plus simple (méthode d'Euler) consiste à fixer un pas h et à calculer :

$$x_0 \text{ est fixé}\,, \quad x_1 = x_0 + A\,h\,x_0 = (I + A\,h)\,x_0\,, \quad x_2 = x_1 + A\,h\,x_1 = (I + 2\,A\,h + A^2\,h^2)\,x_0\,, \ldots$$

On voit que la formule fait apparaître des puissances de la matrice A. Parallèlement, on peut montrer que, dans le cas où la matrice A est diagonalisable, les solutions du système d'équations différentielles sont de la forme :

$$\chi_i(t) \;=\; c_{i1}\,e^{\lambda_1\,t} + c_{i2}\,e^{\lambda_2\,t}\,, \quad 1 \leqslant i \leqslant 2\,, \tag{8.2}$$

où λ_1 et λ_2 sont les valeurs propres de la matrice A et où les coefficients c_{ij} sont liés aux vecteurs propres de A.

Preuve. En effet, on a, en supposant A diagonalisable :

$$\begin{pmatrix} \dot\chi_1 \\ \dot\chi_2 \end{pmatrix} = X\,\Lambda\,X^{-1} \begin{pmatrix} \chi_1 \\ \chi_2 \end{pmatrix} \quad \text{et donc} \quad X^{-1} \begin{pmatrix} \dot\chi_1 \\ \dot\chi_2 \end{pmatrix} = \Lambda\,X^{-1} \begin{pmatrix} \chi_1 \\ \chi_2 \end{pmatrix}.$$

La diagonalisation fournit un changement de variables qui simplifie considérablement le système (on passe d'un système de deux équations différentielles couplées à un système de deux équations différentielles indépendantes l'une de l'autre). Remarquer l'importance, à nouveau, d'avoir, à gauche et à droite de la matrice diagonale, une matrice X et son inverse. Posons donc :

$$\begin{pmatrix} \zeta_1 \\ \zeta_2 \end{pmatrix} \underset{\text{def}}{=} X^{-1} \begin{pmatrix} \chi_1 \\ \chi_2 \end{pmatrix}.$$

Dans les nouvelles variables ζ_i, le système se réécrit (on utilise implicitement le fait que X est une matrice de nombres, et donc de constantes) :

$$\begin{pmatrix} \dot\zeta_1 \\ \dot\zeta_2 \end{pmatrix} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \zeta_1 \\ \zeta_2 \end{pmatrix}.$$

La solution de ce système, dans les nouvelles variables, est évidente. Une simple multiplication par X donne la solution (8.2), dans les anciennes variables :

$$\begin{pmatrix} \zeta_1 \\ \zeta_2 \end{pmatrix} = \begin{pmatrix} c_1\,e^{\lambda_1\,t} \\ c_2\,e^{\lambda_2\,t} \end{pmatrix} \quad \text{et donc} \quad \begin{pmatrix} \chi_1 \\ \chi_2 \end{pmatrix} = X \begin{pmatrix} c_1\,e^{\lambda_1\,t} \\ c_2\,e^{\lambda_2\,t} \end{pmatrix}.$$

□

Le système dynamique (8.1) est stable si les fonctions $\chi_1(t)$ et $\chi_2(t)$ tendent vers zéro quand t tend vers l'infini. Un raisonnement élémentaire montre donc que (8.1) est stable si les parties réelles des deux valeurs propres de A sont négatives.

8.1.3 L'algorithme du *Page rank*

Cette section est fortement inspirée de [6]. L'algorithme du *Page rank* est la base de la méthode qui permet à *Google* de classer les pages web en fonction de leur pertinence. L'idée consiste à attribuer à chaque page un score proportionnel au nombre de fois où cette page serait visitée par un utilisateur qui, partant d'une page web quelconque, suivrait aléatoirement les hyperliens qui apparaissent au fil de sa visite. Supposons que le web soit composé de m pages, numérotées de 1 à m. L'algorithme du *Page rank* calcule un vecteur de m rangs, c'est-à-dire un vecteur de m réels positifs. Les pages web dont les rangs sont les plus élevés sont listées en premier par le moteur de recherche.

À chaque page j, on associe le nombre d_j d'hyperliens qui partent de le page j vers d'autres pages (le nombre de liens sortants), et on définit la *matrice de transition* A par : $A_{ij} = 1/d_j$ s'il existe un hyperlien de la page j vers la page i, zéro sinon. Par exemple, le graphe de la figure 8.2 a pour matrice de transition :

$$A = \begin{pmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{pmatrix}.$$

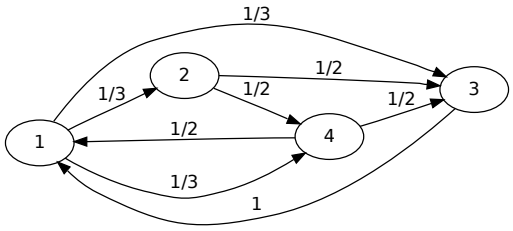


FIGURE 8.2 – Un web composé de $m = 4$ pages. Les hyperliens sont matérialisés par des flèches. Une flèche partant de la page j vers la page i est étiquetée par $1/d_j$, où d_j désigne le nombre de liens sortant de la page j.

Intuitivement, l'idée est que chaque page j transmet de façon équitable $1/d_j$ de son rang, à chacune des pages vers lesquelles elle pointe. Considérons maintenant un utilisateur explorant le web. Initialement, il peut partir de n'importe laquelle des m pages avec une probabilité $1/m$. Initialement, chaque page a donc le même rang et on part du vecteur de rangs :

$$v = \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{pmatrix}.$$

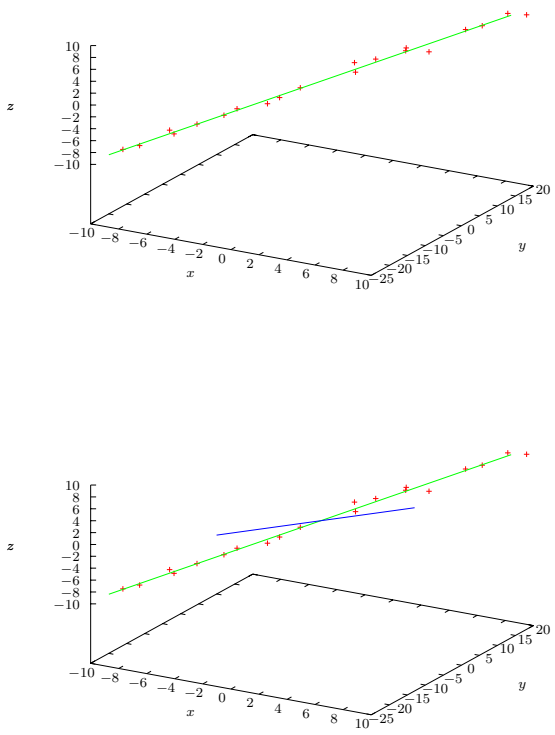


FIGURE 8.3 – Le nuage de points avec la première composante principale (en haut) et la deuxième (en bas)

Ensuite, on met à jour le vecteur v en ajoutant, à chaque page, la contribution des liens entrants. Mathématiquement, le vecteur de rangs mis-à-jour vaut $A\,v$. Si on répète n fois cette mise-à-jour, on obtient le vecteur de rangs $A^n\,v$, qui converge vers un certain vecteur :

$$v^* = \begin{pmatrix} 0.39 \\ 0.13 \\ 0.29 \\ 0.19 \end{pmatrix}.$$

Un moteur de recherche listerait donc dans l'ordre les pages 1, 3, 4 puis 2. On peut montrer [6] que le vecteur v^* est un vecteur propre de la matrice A, associé à la valeur propre 1. On remarque à nouveau qu'un calcul de vecteur propre est naturellement associé à l'étude des puissances de la matrice A.

Remarque : en pratique, on ne travaille pas à partir de la matrice A mais à partir de la matrice

$$\bar{A} = (1-p)\,A + p\,B$$

où p (le *damping factor*) est un réel compris entre 0 et 1 (typiquement $p = 0.15$) et où B est la matrice dont tous les coefficients valent $1/m$. Intuitivement, cette correction modélise la possibilité, pour l'utilisateur, de sauter aléatoirement d'une page vers une autre en entrant directement son URL dans le navigateur (sans suivre les liens, donc). Mathématiquement, cette correction contourne des difficultés liées aux graphes non connexes et aux pages web qui ne pointent vers aucune autre page. En quelque sorte, elle rend le graphe connexe et rajoute des hyperliens fictifs entre toutes les pages.

8.1.4 L'analyse en composantes principales

L'analyse en composantes principales est très importante en statistiques [3, 4]. Elle a des applications dans de nombreux domaines tels que la reconnaissance de visages, la compression d'images. On considère un nuage de $n = 20$ points dans $\mathbb{R}^3$. Ce nuage définit une matrice M de dimension $n \times m$ avec $m = 3$. Voir Figures 8.3 et 8.4. Pour simplifier les calculs, on s'est arrangé pour que les données soient centrées : la moyenne de chaque colonne est nulle. On

$$M = \begin{pmatrix} -8.742771 & -16.309141 & -9.183847 \\ -8.746139 & -20.932279 & -8.291558 \\ -7.562766 & -12.588108 & -7.324801 \\ -6.578025 & -15.187399 & -6.663474 \\ -5.504077 & -13.005027 & -5.270812 \\ -4.515701 & -9.368546 & -4.555080 \\ -3.730677 & -8.784442 & -3.315790 \\ -2.460550 & -5.288003 & -3.091396 \\ -1.575287 & -5.370151 & -1.647970 \\ -0.630553 & -3.308739 & -0.267847 \\ 0.478016 & 7.448750 & 0.844585 \\ 1.615917 & 3.305187 & 1.121139 \\ 2.372446 & 5.942013 & 2.764686 \\ 3.355619 & 10.586407 & 3.521738 \\ 4.496506 & 5.999105 & 5.067528 \\ 5.608201 & 8.136502 & 4.670766 \\ 6.406469 & 15.188349 & 6.412231 \\ 7.505404 & 15.534796 & 7.406800 \\ 8.472581 & 18.845804 & 8.799519 \\ 9.735386 & 19.154922 & 9.003584 \end{pmatrix}.$$

FIGURE 8.4 – Les données initiales

calcule la matrice de covariance A de la matrice M par la formule suivante (la formule est simple parce que les données sont centrées).

$$A = \frac{1}{n-1} M^T M \,.$$

Cette matrice est symétrique :

$$A = \begin{pmatrix} 34.8563048127361 & 72.7454532296073 & 34.3585893385741 \\ 72.7454532296073 & 158.165935897380 & 71.8970011140296 \\ 34.3585893385741 & 71.8970011140296 & 34.0527515138144 \end{pmatrix}.$$

Il suffit ensuite de calculer une factorisation de Schur de la matrice A. Comme A est symétrique, cette factorisation est une diagonalisation $A = Q\,\Lambda\,Q^T$ avec Q orthogonale. On obtient :

$$Q = \begin{pmatrix} 0.389471202353109 & -0.703316634916119 & 0.594691427202284 \\ 0.836742856867970 & 0.000336483686725517 & -0.547595907818126 \\ 0.384933207216593 & 0.710876640373697 & 0.588626221088521 \end{pmatrix},$$

$$\Lambda = \begin{pmatrix} 225.101436933838 & & \\ & 0.0935891289300329 & \\ & & 1.87996616116170 \end{pmatrix}.$$

La première composante principale est la droite $\mathscr{D}_1$ dont le vecteur directeur est le vecteur propre associé à la valeur propre la plus grande, en valeur absolue, c'est-à-dire $\lambda_1 \simeq 225$. Ce vecteur directeur est donc q_1, première colonne de Q. Géométriquement, $\mathscr{D}_1$ est telle que, si on projette chaque point orthogonalement sur $\mathscr{D}_1$, on obtient une dispersion maximale. La deuxième composante principale est la droite $\mathscr{D}_2$ dont le vecteur directeur est le vecteur propre associé à la deuxième plus grande valeur propre, en valeur absolue, c'est-à-dire $\lambda_3 \simeq 1.88$. Ce vecteur directeur est donc q_3, troisième colonne de Q. Voir Figure 8.3. La formule suivante calcule les coordonnées des points, projetés orthogonalement dans le plan défini par les deux premières composantes principales. Voir Figure 8.5.

$$\textit{données projetées} \;=\; M \begin{pmatrix} q_1 & q_3 \end{pmatrix}.$$

8.2 Rappels mathématiques

Cette section doit beaucoup à [7, Lecture 24]. Soit A une matrice carrée de dimension $m \times m$. Un vecteur non nul x est un *vecteur propre* de A (*eigenvector* en Anglais), et le nombre complexe $\lambda \in \mathbb{C}$ est la *valeur propre* (*eigenvalue* en Anglais) qui lui est associée, si

$$A\,x = \lambda\,x\,.$$

Un vecteur propre d'une matrice A est en quelque sorte défini à un facteur près. En effet, si x est un vecteur propre de A de valeur propre λ alors $\alpha\,v$ l'est aussi, quel que soit $\alpha \neq 0$.

Les valeurs propres de A sont les racines du *polynôme caractéristique* de A, d'indéterminée λ, défini par

$$C(\lambda) = \det(\lambda\,I - A)\,.$$

Preuve. Dire que λ est une valeur propre de A, c'est dire, par définition des valeurs propres, qu'il existe un vecteur x non nul tel que $\lambda\,x - A\,x = (\lambda\,I - A)\,x = 0$. D'après le théorème 1, point 4, c'est équivalent à dire que la matrice $\lambda\,I - A$ n'est pas inversible. D'après le théorème 1, point 8, c'est équivalent à dire que le déterminant de $\lambda\,I - A$ est nul. □

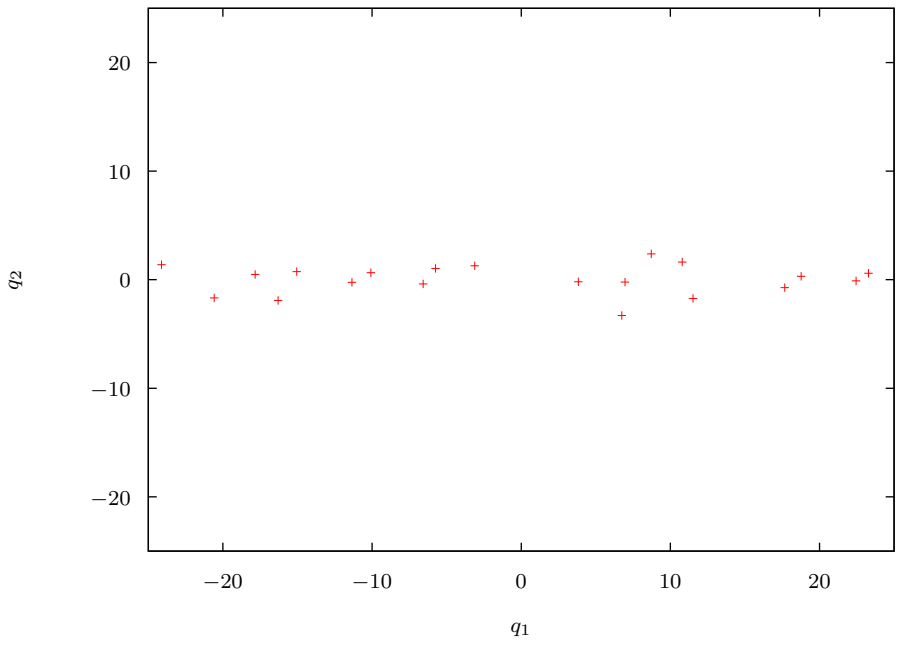


FIGURE 8.5 – Le nuage de points, projeté orthogonalement dans le plan défini par les deux premières composantes principales

Il y a deux conséquences importantes : la matrice A a au plus m valeurs propres ; même si A est une matrice à coefficients réels, ses valeurs propres peuvent être complexes.

Deux matrices A et B sont dites semblables (*similar* en Anglais) s'il existe une matrice inversible X telle que $A = X\,B\,X^{-1}$. Si A et B sont deux matrices semblables alors A et B ont mêmes valeurs propres [7, Theorem 24.3].

Diagonaliser A, c'est chercher une factorisation

$$A \;=\; X\,\Lambda\,X^{-1}\,, \tag{8.3}$$

où X est une matrice carrée inversible et Λ une matrice diagonale. Une telle factorisation n'existe pas toujours [1]. Les éléments diagonaux de Λ sont les valeurs propres de A.

> **Preuve.** Notons $\lambda_1, \lambda_2, \ldots, \lambda_m$ les éléments diagonaux de Λ. Le polynôme caractéristique de Λ s'écrit $C(\lambda) = (\lambda - \lambda_1)\,(\lambda - \lambda_2)\cdots(\lambda - \lambda_m)$. Les λ_i sont donc les valeurs propres de Λ. Les matrices A et Λ sont semblables. Les λ_i sont donc les valeurs propres de A. □

Les colonnes de X sont des vecteurs propres de A (le vecteur x_i étant associé à λ_i).

> **Preuve.** Comme $A = X\,\Lambda\,X^{-1}$, on a $A\,X = X\,\Lambda$. Développons le membre gauche :
>
> $$A\,X = \begin{pmatrix} A\,x_1 & A\,x_2 & \cdots & A\,x_m \end{pmatrix}.$$

1. Si A a m valeurs propres distinctes, alors A est diagonalisable. Comme un polynôme dont les coefficients sont pris aléatoirement n'a que des racines distinctes, on voit qu'une matrice carrée prise au hasard est diagonalisable.

Développons le membre droit :

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & & x_{2m} \\ \vdots & & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mm} \end{pmatrix} \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{pmatrix} = \begin{pmatrix} \lambda_1 x_{11} & \lambda_2 x_{12} & \cdots & \lambda_m x_{1m} \\ \lambda_1 x_{21} & \lambda_2 x_{22} & & \lambda_m x_{2m} \\ \vdots & & & \vdots \\ \lambda_1 x_{m1} & \lambda_2 x_{m2} & \cdots & \lambda_m x_{mm} \end{pmatrix}$$

$$= \begin{pmatrix} \lambda_1 x_1 & \lambda_2 x_2 & \cdots & \lambda_m x_m \end{pmatrix}$$

Pour tout $1 \leqslant i \leqslant m$, on a donc $A\,x_i = \lambda_i\,x_i$. □

D'autres factorisations sont disponibles. La plus importante, pour le calcul numérique, est la *factorisation de Schur*. C'est elle que les algorithmes que nous allons étudier vont calculer. Dans le cas où toutes les valeurs propres de A sont réelles, une factorisation de Schur de A est une factorisation

$$A \;=\; Q\,T\,Q^T\,, \tag{8.4}$$

où Q est une matrice orthogonale et T est une matrice triangulaire supérieure. Comme A et T sont semblables, les valeurs propres de A apparaissent nécessairement sur la diagonale de T. On dit de la factorisation de Schur que c'est une factorisation qui *révèle les valeurs propres* de A (*eigenvalue revealing factorization*).

Dans le cas où certaines valeurs propres de A sont complexes, la factorisation de Schur de A existe aussi mais la notion de matrice orthogonale doit être généralisée [2] au cas des matrices à coefficients dans $\mathbb{C}$. Les matrices orthogonales, au sens complexe, sont dites *unitaires* [7, Lecture 2]. Sous réserve de bien prendre en compte cette généralisation, on a [7, Theorem 24.9] :

Proposition 17 *Toute matrice carrée a une factorisation de Schur.*

Dans le cas très important en pratique des matrices symétriques, la matrice T de la forme de Schur se trouve être diagonale et la forme de Schur, une diagonalisation de A. C'est le moment d'énoncer la proposition suivante [3] [7, Exercise 2.3] :

Proposition 18 *Les valeurs propres d'une matrice symétrique sont réelles.*

2. La généralisation au cas complexe de la transposée Q^T d'une matrice Q est la transposée-conjuguée, notée Q^*, qui s'obtient en remplaçant chaque entrée de Q^T par son nombre complexe conjugué. La proposition 17 s'énonce alors : toute matrice A admet une factorisation de Schur $A = Q\,T\,Q^*$.

3. La proposition 18 est en fait un corollaire de la proposition 17 : supposons A réelle et symétrique. Alors

$$A = A^* = (Q\,T\,Q^*)^* = Q\,T^*\,Q^*\,.$$

La dernière égalité s'obtient avec la formule pour la transposée d'un produit, qui se généralise au cas de la transposée-conjuguée. On en conclut que $T = T^*$. Une matrice triangulaire supérieure égale à sa transposée est nécessairement diagonale. Comme les seuls nombres complexes égaux à leur conjugué sont les réels, les éléments diagonaux de T — les valeurs propres de A — sont nécessairement réels.

Une subtilité a pu vous échapper : une diagonalisation d'une matrice A est une factorisation $A = X\,\Lambda\,X^{-1}$ avec X inversible quelconque. Si cette diagonalisation est aussi une factorisation de Schur, alors X est non seulement inversible mais orthogonale. On obtient donc la proposition suivante, qui est très importante, dans le cadre de l'analyse en composantes principales par exemple :

Proposition 19 *Toute matrice symétrique A peut se factoriser sous la forme*

$$A = Q\,\Lambda\,Q^T$$

avec Λ diagonale et Q orthogonale. Cette factorisation est une diagonalisation de A et aussi une factorisation de Schur de A.

8.3 Considérations générales sur le calcul des valeurs propres

Cette section doit beaucoup à [7, Lecture 25]. Le problème de la résolution des équations polynomiales $p(z) = 0$ peut *se réduire* au problème du calcul des valeurs propres d'une certaine matrice A, appelée *matrice compagnon* du polynôme p. La matrice compagnon d'un polynôme $p(z) = z^m + a_{m-1}\,z^{m-1} + \cdots + a_1\,z + a_0$ s'écrit

$$\begin{pmatrix} 0 & & & & & -a_0 \\ 1 & 0 & & & & -a_1 \\ & 1 & 0 & & & -a_2 \\ & & 1 & \ddots & & \vdots \\ & & & \ddots & 0 & -a_{m-2} \\ & & & & 1 & -a_{m-1} \end{pmatrix}.$$

L'expression « se réduire » a ici un sens technique, classique en informatique [1, section 34.3], qui signifie qu'on peut coder un problème (la résolution d'une équation polynomiale quelconque) comme un cas particulier d'un autre problème (le calcul des valeurs propres d'une matrice carrée quelconque).

Or des mathématiciens du dix-neuvième siècle (Abel, Ruffini) ont montré que les équations polynomiales de degré 5 ou plus, ne peuvent pas (en général) se résoudre par radicaux. En d'autres termes, ces équations ont des racines mais ces racines ne peuvent pas (dans le cas général) s'exprimer à partir des coefficients a_i du polynôme à résoudre, par des formules finies, ne faisant intervenir que des sommes, des produits et des extractions de racines m-ièmes.

En raison de la réduction de problème mentionnée plus haut, on en déduit que les valeurs propres d'une matrice A quelconque ne peuvent pas s'exprimer à partir des coefficients a_{ij} de la matrice A, par des formules finies, ne faisant intervenir que des sommes, des produits et des extractions de racines m-ièmes. Or ces opérations sont exactement les opérations mises en œuvre dans les algorithmes numériques.

Conséquence importante : il est impossible d'écrire un algorithme général de calcul de valeurs propres qui soit *exact*, dans le sens où le pivot de Gauss est un algorithme exact, c'est-à-dire un algorithme qui produirait la solution exacte du problème à résoudre en un nombre fini d'opérations élémentaires, si on menait les calculs sans faire aucune erreur d'arrondi (par exemple avec des nombres algébriques, en MAPLE). Le mieux qu'un algorithme général de calcul de valeurs propres puisse faire, c'est calculer des suites de réels qui *convergent* vers les valeurs propres, mais sans jamais les calculer exactement, même si on menait les calculs sans aucune erreur d'arrondi. De tels algorithmes sont classiquement appelés *algorithmes itératifs*, ce qui est une appellation un peu malheureuse.

À première vue, les algorithmes itératifs semblent peu intéressants puisqu'il leur faudrait, en théorie, un temps infini pour produire leur résultat. En fait, c'est le contraire : il est fréquent qu'ils convergent extrêmement rapidement vers leur résultat (en doublant ou en triplant le nombre de chiffres exacts à chaque itération) et donc qu'ils atteignent la meilleure précision possible $\pm\varepsilon_{\text{machine}}$ très rapidement.

L'algorithme $Q\,R$, que nous allons étudier, enchaîne deux étapes :

1. mise sous forme de Hessenberg de la matrice A (sous-algorithme exact) ;
2. calcul des valeurs propres de A à partir de la forme de Hessenberg (sous-algorithme itératif).

Il est presque paradoxal de constater que la mise sous forme de Hessenberg, qui constitue la partie exacte de l'algorithme $Q\,R$, est plus coûteuse que la partie itérative. Même dans le domaine de la résolution des systèmes d'équations linéaires, domaine où on dispose d'algorithmes exacts, il existe des algorithmes itératifs plus efficaces que les algorithmes exacts, mais qui ne s'appliquent pas dans tous les cas. Ces méthodes, que nous n'aurons pas le temps d'étudier dans ce cours, sont développées dans [7, Lecture 32 and ff.].

8.4 Calculer un vecteur propre ou une valeur propre

Dans cette section, on décrit des méthodes simples qui peuvent parfois s'employer isolément pour calculer une valeur propre ou un vecteur propre. Combinées, elles donnent l'algorithme $Q\,R$. On commence par la proposition suivante.

Proposition 20 *Soit $A\,x = b$ un système d'équations linéaires avec A inversible. Notons $a_1, a_2, \ldots, a_m$ les colonnes de A et $\chi_1, \chi_2, \ldots, \chi_m$ les coordonnées de x. La solution de ce système permet d'exprimer b comme combinaison linéaire des colonnes de A :*

$$b = \chi_1\, a_1 + \chi_2\, a_2 + \cdots + \chi_m\, a_m\,.$$

Preuve. Le fait que la solution soit unique vient du fait que A est inversible (on a $x = A^{-1}\,b$). Il suffit ensuite de développer le produit $A\,x$:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{pmatrix} \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_m \end{pmatrix} = \begin{pmatrix} \chi_1\,a_{11} + \chi_2\,a_{12} + \cdots + \chi_m\,a_{1m} \\ \chi_1\,a_{21} + \chi_2\,a_{22} + \cdots + \chi_m\,a_{2m} \\ \vdots \\ \chi_1\,a_{m1} + \chi_2\,a_{m2} + \cdots + \chi_m\,a_{mm} \end{pmatrix}$$

$$= \quad \chi_1\,a_1 + \chi_2\,a_2 + \cdots + \chi_m\,a_m$$

□

8.4.1 La méthode de la puissance

Elle calcule un vecteur propre associé à la valeur propre la plus grande en valeur absolue. La méthode de la puissance peut être utilisée pour l'algorithme du *Page rank*.

Pour faire simple, on suppose que A a m valeurs propres distinctes (A est donc diagonalisable) et que l'une d'elles (notons-la λ_1) est strictement plus grande que les autres, en valeur absolue. En particulier, λ_1 est réelle [4].

Proposition 21 *Soit v un vecteur (presque) quelconque non nul. La suite de vecteurs $v_k = A^k\,v$ converge vers un vecteur propre associé à λ_1.*

Preuve. Soit $A = X\,\Lambda\,X^{-1}$ une diagonalisation de A. Notons $x_1, x_2, \ldots, x_m$ les vecteurs propres formant les colonnes de X et $\beta_1, \beta_2, \ldots, \beta_m$ les scalaires tels que (appliquer la proposition 20 au système $X\,b = v$ en notant $\beta_1, \beta_2, \ldots, \beta_m$ les coordonnées de b) :

$$v \quad = \quad \beta_1\,x_1 + \beta_2\,x_2 + \cdots + \beta_m\,x_m\,.$$

On a :

$$\begin{aligned} A^k\,v &= A^k\,(\beta_1\,x_1 + \beta_2\,x_2 + \cdots + \beta_m\,x_m) \\ &= \beta_1\,A^k\,x_1 + \beta_2\,A^k\,x_2 + \cdots + \beta_m\,A^k\,x_m \\ &= \beta_1\,\lambda_1^k\,x_1 + \beta_2\,\lambda_2^k\,x_2 + \cdots + \beta_m\,\lambda_m^k\,x_m\,. \end{aligned}$$

On passe de la deuxième à la troisième ligne en appliquant le fait que les x_i sont des vecteurs propres de A et donc que $A\,x_i = \lambda_i\,x_i$. Sous réserve que β_1 soit non nul (pour presque tous les vecteurs v, donc), le terme $\beta_1\,\lambda_1^k$ domine tous les autres termes $\beta_i\,\lambda_i^k$ et on conclut que $A^k\,v$ converge vers un multiple de x_1. □

En pratique, on normalise les vecteurs v_k à chaque itération. La vitesse de convergence est faible : à chaque itération, l'erreur, qu'on peut définir comme $\|x_1 - v\|_2$, n'est divisée

4. Comme A est réelle, son polynôme caractéristique a des coefficients réels et, pour chaque valeur propre complexe λ de A, le nombre complexe conjugué de λ (qui a même module que λ) est aussi une valeur propre de A.

que par une constante, de l'ordre de $|\lambda_1/\lambda_2|$, où λ_2 désigne la deuxième valeur propre la plus grande, en valeur absolue [7, Lecture 27]. Une telle vitesse de convergence est dite *linéaire*[5].

Dans l'exemple suivant, la matrice A a trois valeurs propres -4, 2 et 6. La méthode de la puissance converge vers un vecteur propre, de longueur 1, associé à la valeur propre 6. On observe en effet que le vecteur calculé est proche d'un multiple de la troisième colonne de X. À la fin du calcul, connaissant une aproximation du vecteur propre, on peut retrouver la valeur propre en calculant le *quotient de Rayleigh* du vecteur.

```
>>> import numpy as np
>>> import scipy.linalg as nla
```

Pour construire une matrice dont on connaît à l'avance les valeurs propres, l'astuce consiste à partir de la diagonalisation et d'en déduire la matrice.

```
>>> X = np.array([[2,-1,1],[0,-2,2],[-1,1,0]])
>>> X
array([[ 2, -1,  1],
       [ 0, -2,  2],
       [-1,  1,  0]])
>>> Xinv = nla.inv(X)
>>> Xinv
array([[ 0.5 , -0.25, -0.  ],
       [ 0.5 , -0.25,  1.  ],
       [ 0.5 ,  0.25,  1.  ]])
>>> Lambda = np.diag ([-4,2,6])
>>> Lambda
array([[-4,  0,  0],
       [ 0,  2,  0],
       [ 0,  0,  6]])
```

Comme $A = X\,\Lambda\,X^{-1}$ on connaît les valeurs propres de A.

```
>>> A = np.dot (X, np.dot (Lambda, Xinv))
>>> A
array([[-2. ,  4. ,  4. ],
       [ 4. ,  4. ,  8. ],
       [ 3. , -1.5,  2. ]])
```

5. Les algorithmes que nous étudions calculent des suites de matrices (M_k) qui convergent vers une certaine matrice résultat M. On peut donc, à chaque itération k, définir l'erreur e_k à l'étape k (par exemple, on pourrait prendre $e_k = \|M_k - M\|$). Pour beaucoup d'algorithmes, il arrive qu'il existe une constante c et un entier α tels que, pour tout k, on ait $e_{k+1} < c\,e_k^\alpha$. Lorsque $\alpha = 1$, on parle de vitesse de convergence *linéaire* (en prenant $c = 10^{-1}$, on voit qu'on gagne une décimale à chaque itération). Lorsque $\alpha = 2$, on parle de vitesse de convergence *quadratique* (en prenant $c = 1$ et $e_k = 10^{-p}$, on voit que $e_{k+1} = 10^{-2p}$ et donc que le nombre p de décimales exactes double à chaque itération). Lorsque $\alpha = 3$, on parle de vitesse de convergence *cubique*. Voir [7, Exercise 25.2].

On vérifie qu'on ne s'est pas trompé. En général, les valeurs propres d'une matrice réelle sont complexes. Il est donc normal que le résultat de la fonction `eigvals` (pour *eigenvalues*) soit composée de nombres complexes.

```
>>> nla.eigvals(A)
array([-4.+0.j,  6.+0.j,  2.+0.j])
```

On choisit un vecteur v aléatoire.

```
>>> v = np.array([2,3,-5])
```

Le vecteur $b = X^{-1}\,v$ (qu'on obtient en résolvant le système d'équations linéaires $X\,b = v$) contient les coordonnées du vecteur v dans la base des colonnes de X, qui sont des vecteurs propres de A.

```
>>> b = nla.solve(X,v)
>>> b
array([ 0.25, -4.75, -3.25])
```

On vérifie l'affirmation précédente, c'est-à-dire que $v = b_0\,x_0 + b_1\,x_1 + b_2\,x_2$ où x_i désigne la $i$ème colonne de X.

```
>>> x0=X[:,0]
>>> x1=X[:,1]
>>> x2=X[:,2]
>>> b[0]*x0 + b[1]*x1 + b[2]*x2
array([ 2.,  3., -5.])
```

Une information utile : $b_1 \neq 0$ (coefficient associé à la valeur propre la plus grande en valeur absolue). Dans la mesure où v était tiré au hasard, il était très peu probable que ce ne soit pas le cas. En pratique, c'est un test qu'on n'effectue pas. Appliquons donc la méthode de la puissance à partir de v

```
>>> v = np.array([2,3,-5])
>>> v = (1/nla.norm(v,2))*v
>>> for i in range (30) :
...     w = np.dot (A, v)
...     v = (1/nla.norm(w,2))*w
```

Voici le vecteur obtenu après 30 itérations

```
>>> v
array([-4.47213308e-01, -8.94427335e-01, -1.79404756e-07])
```

On vérifie qu'il est de longueur 1 (c'est dû aux divisions systématiques par la norme deux au cours de l'algorithme)

```
>>> np.dot (v,v)
1.0
```

On peut donc simplifier la formule du quotient de Rayleigh en évitant de diviser par $v^T\,v$. On retrouve la valeur propre la plus grande en valeur absolue.

```
>>> np.dot(v,np.dot(A,v))
6.000001604643565
```

8.4.2 Le quotient de Rayleigh

Définition 5 *Le quotient de Rayleigh d'un vecteur x est défini comme le scalaire :*

$$r(x) = \frac{x^T A\, x}{x^T\, x}\cdot$$

Si x est un vecteur propre de A alors $r(x) = \lambda$ est la valeur propre qui lui est associée. Si v est proche d'un vecteur propre x de A alors $r(v)$ est une bonne approximation de la valeur propre associée à x. L'expression « bonne approximation » peut être rendue précise : dans le cas des matrices symétriques, le quotient de Rayleigh de v converge quadratiquement vers λ, lorsque v tend vers x [7, Lecture 27]. En termes un peu imprécis, cela signifie que le nombre de décimales exactes de $r(v)$ est le double du nombre de décimales exactes de v. En termes plus précis [6],

$$r(v) - r(x) = O(\|v - x\|^2)\,.$$

Dans le cas des matrices non symétriques, le quotient de Rayleigh converge linéairement (et plus quadratiquement) vers λ [7, Exercise 27.3].

8.4.3 La méthode de la puissance inverse

C'est une amélioration de la méthode de la puissance : elle converge potentiellement beaucoup plus vite et il est possible de choisir le vecteur propre à calculer. C'est la méthode standard utilisée pour calculer un ou plusieurs vecteurs propres d'une matrice dont les valeurs propres sont connues [7] [7, Lecture 27].

L'idée importante (exprimée un peu informellement), c'est que si f est une fonction et $A = X\,\Lambda\,X^{-1}$ est une diagonalisation de A, alors

$$f(A) = X\, f(\Lambda)\, X^{-1}$$

est une diagonalisation de $f(A)$. En d'autres termes, appliquer une fonction f sur A transforme les valeurs propres de A par f mais laisse les vecteurs propres inchangés. Par exemple, prenons

$$A \;=\; \begin{pmatrix} -2 & 4 & 4 \\ 4 & 4 & 8 \\ 3 & -\frac{3}{2} & 2 \end{pmatrix}.$$

On a $A = X\,\Lambda\,X^{-1}$ avec

$$X\,, \quad \Lambda \;=\; \begin{pmatrix} 2 & -1 & 1 \\ 0 & -2 & 2 \\ -1 & 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} -4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{pmatrix}.$$

6. Comme toutes les normes sont équivalentes [7, Lecture 14], il est inutile d'en préciser une à l'intérieur du grand O.

7. C'est le cas, par exemple, si on calcule une factorisation de Schur d'une matrice non symétrique.

Prenons $f(x) = x - \mu$ où μ désigne un paramètre. Alors $f(A) = A - \mu\, I$ et on a la diagonalisation $f(A) = X\, f(\Lambda)\, X^{-1}$ avec

$$X\,, \quad f(\Lambda) \;=\; \begin{pmatrix} 2 & -1 & 1 \\ 0 & -2 & 2 \\ -1 & 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} -4-\mu & 0 & 0 \\ 0 & 2-\mu & 0 \\ 0 & 0 & 6-\mu \end{pmatrix}.$$

De même, prenons $f(x) = 1/(x - \mu)$. Alors $f(A) = (A - \mu\, I)^{-1}$ et on a la diagonalisation $f(A) = X\, f(\Lambda)\, X^{-1}$ avec

$$X\,, \quad f(\Lambda) \;=\; \begin{pmatrix} 2 & -1 & 1 \\ 0 & -2 & 2 \\ -1 & 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} \dfrac{1}{-4-\mu} & 0 & 0 \\ 0 & \dfrac{1}{2-\mu} & 0 \\ 0 & 0 & \dfrac{1}{6-\mu} \end{pmatrix}.$$

Supposons que $\mu \simeq 2$ soit proche de la valeur propre $\lambda = 2$ de A. Alors la méthode de la puissance, appliquée à la matrice $B = (A - \mu\, I)^{-1}$, va converger vers un vecteur propre associé à la valeur propre $1/(2 - \mu)$ de B, c'est-à-dire un vecteur propre associé à la valeur propre 2 de A. La vitesse de convergence reste linéaire mais la constante est d'autant plus élevée que μ est proche de λ.

```
>>> A = np.array([[-2,4,4],[4,4,8],[3,-1.5,2]])
>>> mu = 2.01
>>> B = nla.inv (A - mu*np.eye(3))
>>> B
array([[  49.95892393,  -24.85414868,  100.25062657],
       [ 100.25062657,  -49.87468672,  200.50125313],
       [ -49.91680532,   24.95840266, -100.        ]])
>>> v = np.array([2,3,-5])
>>> v = (1/nla.norm(v,2))*v
```

En trois itérations seulement !

```
>>> for i in range (3) :
...     w = np.dot (B,v)
...     v = (1/nla.norm(w,2))*w
```

On vérifie qu'on a bien trouvé un vecteur propre associé à la valeur propre 2 de A

```
>>> v
array([-0.40824829, -0.81649658,  0.40824829])
>>> np.dot(v,np.dot(A,v))
2.0000000351772242
```

On remarque que le quotient de Rayleigh de v est nettement plus précis que μ. Cela suggère une idée : améliorer μ à chaque itération en remplaçant son ancienne valeur par le quotient de Rayleigh de v. On obtient alors une méthode, appelée l'itération du quotient de Rayleigh, étudiée dans [7, Lecture 27, page 207], qui a un inconvénient : la matrice $(A - \mu\, I)$ change à chaque itération.

Revenons à la méthode de la puissance inverse : mathématiquement, calculer $w = (A - \mu\, I)^{-1}\, v$ est équivalent à résoudre le système d'équations linéaires $(A - \mu\, I)\, w = v$. Numériquement le seconde solution est bien meilleure. De plus, comme la matrice $(A - \mu\, I)$ ne change pas d'une itération sur l'autre, il est possible de précalculer une factorisation $L\, U$ ou $Q\, R$ de cette matrice. Cela donne :

```
>>> A = np.array([[-2,4,4],[4,4,8],[3,-1.5,2]])
>>> mu = 2.01
```

On calcule une factorisation $L\, U$ de $C = A - \mu\, I$

```
>>> C = A - mu*np.eye(3)
>>> P, L, U = nla.lu(C)
```

On vérifie que la matrice de permutation P est bien égae à l'identité et qu'on peut donc l'ignorer

```
>>> P
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> v = np.array([2,3,-5])
>>> v = (1/nla.norm(v,2))*v
```

À chaque itération, on calcule w, solution de $C\, w = v$ par deux résolutions de systèmes triangulaires

```
>>> for i in range (3) :
...     w = nla.solve_triangular(L, v, lower=True)
...     w = nla.solve_triangular(U, w, lower=False)
...     v = (1/nla.norm(w,2))*w
...
>>> v
array([-0.40824829, -0.81649658,  0.40824829])
>>> np.dot(v,np.dot(A,v))
2.000000035177225
```

On peut quand même faire une objection à la méthode de la puissance inverse : si μ est vraiment très proche de λ, la matrice $(A - \mu\, I)$ est très proche d'une matrice singulière, le système d'équations à résoudre est très mal conditionné et ne pourra pas être résolu avec précision. La réponse est : le système est effectivement très mal conditionné mais c'est sans conséquence dans ce cas-ci [5, Lecture 15, § 4], [7, Exercise 27.5].

8.5 L'algorithme $Q\,R$

The Q R algorithm, dating from the early 1960s, is one of the jewels of numerical analysis [7, Lecture 28]. La progression de cette section est inspirée de [7, Lectures 28-29]. En particulier, on se concentre sur le cas des matrices symétriques. Au lecteur intéressé, on conseille fortement la lecture de [2, chapitre V], qui suit une présentation différente et traite le cas des matrices carrées quelconques.

La formulation la plus simple de l'algorithme $Q\,R$ est donnée en pseudo-code, Figure 8.6. Sous certaines hypothèses, cet algorithme élémentaire converge vers la matrice T d'une factorisation de Schur de la matrice A, triangulaire supérieure si A est arbitraire, diagonale si A est symétrique.

Version élémentaire de l'algorithme $Q\,R$
$A_0 = A$
for $k = 1, 2, \ldots, N$ do
  Q_k, R_k = une factorisation $Q\,R$ de A_{k-1}
  $A_k = R_k\,Q_k$
end do

FIGURE 8.6 – La version la plus simple de l'algorithme $Q\,R$

Des approximations des valeurs propres sont obtenues sur la diagonale de la matrice calculée A_N. Pour chaque valeur propre, un vecteur propre peut être ensuite calculé par la méthode de la puissance inverse. Dans le cas symétrique, ces vecteurs propres fournissent la matrice orthogonale qui complète la factorisation de Schur de A. Dans les discussions ci-dessous, on introduit une autre façon d'obtenir cette matrice orthogonale (proposition 22). Cette autre méthode est surtout une méthode de principe, destinée à expliquer l'algorithme, mais pas une méthode utilisable en pratique.

```
>>> import numpy as np
>>> import scipy.linalg as nla
```

Pour éviter des passages à la ligne qui rendraient la matrice finale difficile à lire, on augmente la largeur d'une ligne

```
>>> np.set_printoptions(linewidth=160)
```

On construit une suite de matrices qu'on stocke (pour des raisons pédagogiques) dans une liste A. La matrice initiale A_0 est construite de façon à être symétrique avec des valeurs propres connues à l'avance (-74, 38, 2 et 42).

```
>>> M = np.array([[1,-1,4],[1,4,-2],[1,4,2],[1,-1,0]])
>>> Q, R = nla.qr (M)
>>> Lambda = np.diag([-74,38,2,42])
```

```
>>> A0 = np.dot (Q, np.dot (Lambda, np.transpose(Q)))
>>> A = [ A0 ]
>>> N = 20
>>> for k in range (1,N+1) :
...     Qk, Rk = nla.qr (A[k-1])
...     Ak = np.dot (Rk, Qk)
...     A.append (Ak)
```

Après $N = 20$ itérations, on observe que la suite de matrices converge vers une matrice diagonale. De bonnes approximations des valeurs propres figurent sur la diagonale.

```
>>> A[N]
array([[-7.40000000e+01,  1.82077198e-04, -1.39575568e-03, -4.09344697e-15],
       [ 1.82077198e-04,  3.80000000e+01,  2.26908838e-09,  3.65347808e-16],
       [-1.39575568e-03,  2.26907769e-09,  4.20000000e+01,  8.44495405e-15],
       [ 3.45983594e-30, -9.57702193e-25, -1.43771928e-25,  2.00000000e+00]])
```

Proposition 22 *Soit A une matrice carrée quelconque. Les matrices calculées par l'algorithme de la figure 8.6 vérifient la relation* (8.5)*. En particulier, les matrices A et A_k sont semblables.*

$$\begin{array}{rcl} A & = & P_k\, A_k\, P_k^T\,, \quad \textit{avec} \\ P_k & \underset{\text{def}}{=} & Q_1\, Q_2 \cdots Q_k\,. \end{array} \tag{8.5}$$

Preuve. La relation (8.5) peut se montrer au moyen d'un raisonnement par récurrence sur k dont l'argument clef est $A_{k-1} = Q_k\, A_k\, Q_k^T$. En effet, $Q_k\, A_k\, Q_k^T = Q_k\, R_k\, Q_k\, Q_k^T = Q_k\, R_k = A_{k-1}$. La similitude entre A et A_k vient du fait que la matrice P_k est orthogonale puisque les matrices $Q_1, Q_2, \ldots, Q_k$ le sont et donc que $P_k^T = P_k^{-1}$. □

Proposition 23 *Supposons que A soit symétrique et considérons $A = X\,\Lambda\,X^T$ une factorisation de Schur de A (la matrice X est donc orthogonale). Si P_k converge vers X alors A_k converge vers Λ.*

Preuve. D'après (8.5) on a $A = P_k\, A_k\, P_k^T$. Comme $A = X\,\Lambda\,X^T$, on a $X^T\,A\,X = \Lambda = X^T\,P_k\,A_k\,P_k^T\,X$. Si P_k converge vers X alors les produits $X^T\,P_k$ et $P_k^T\,X$ convergent vers la matrice identité. □

8.5.1 Un lien avec la méthode de la puissance

On suppose à nouveau, pour simplifier, que A est symétrique. Ses valeurs propres sont donc réelles. On suppose de plus que

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_m| > 0\,. \tag{8.6}$$

D'après la proposition 23, pour justifier l'algorithme $Q\,R$, il suffit de se convaincre que (P_k) converge vers une matrice $X = \begin{pmatrix} x_1 & x_2 & \cdots & x_m \end{pmatrix}$ orthogonale dont les colonnes sont des vecteurs propres de A. Cette conviction s'établit en deux temps :

1. On a $Q = P_k$, où Q résulte de la factorisation $Q\,R$ de A^k. Voir [7, Theorem 28.3] pour une preuve.
2. Plus N est grand, plus les colonnes de Q (résultant de la factorisation $Q\,R$ de A^N) sont proches des vecteurs propres de A. En effet, d'après l'hypothèse (8.6) et les propriétés de la méthode de la puissance, les m colonnes v_i de A^N sont de la forme suivante. Elles convergent toutes vers x_1 tout en restant linéairement indépendantes [8]. Le reste du raisonnement est illustré Figure 8.7.

$$v_i \;=\; \alpha_{1i}\,x_1 + \alpha_{2i}\,x_2 + \cdots + \alpha_{mi}\,x_m \quad (\alpha_{1i} \gg \alpha_{2i} \gg \cdots \gg \alpha_{mi})\,. \tag{8.7}$$

Pour conclure, l'algorithme $Q\,R$ naïf est une variante proche de la méthode de la puissance, combinée à une orthogonalisation. Numériquement, il est préférable d'effectuer les factorisations $Q\,R$ à chaque itération, pour des raisons de stabilité numérique.

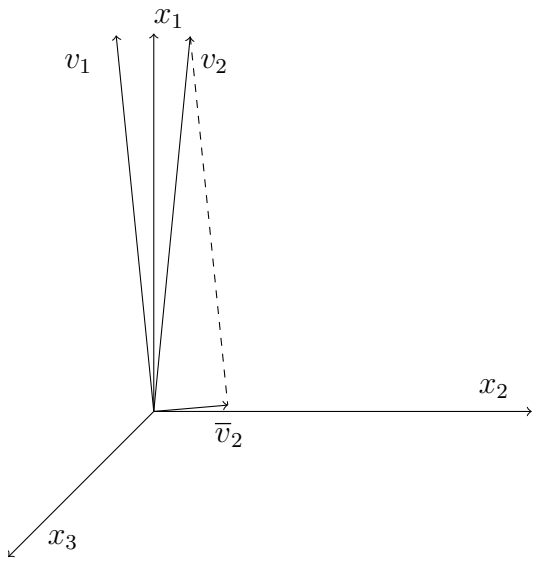


FIGURE 8.7 – Les trois vecteurs x_1, x_2 et x_3 sont orthogonaux deux–à–deux. Les vecteurs v_1 et v_2 sont linéairement indépendants mais tous les deux de la forme $\alpha_{1i}\,x_1 + \alpha_{2i}\,x_2 + \alpha_{3i}\,x_3$ avec $\alpha_{1i} \gg \alpha_{2i} \gg \alpha_{3i}$ $(i = 1, 2)$. Orthogonalisons la matrice $(v_1\ v_2)$ (par Gram-Schmidt ou Householder). La direction du vecteur v_1 reste inchangée. Le vecteur v_2 est projeté sur le plan orthogonal à v_1 (le vecteur $\overline{v}_2$ du dessin), puis renormalisé. Comme $\alpha_{12} \gg \alpha_{22}, \alpha_{32}$, le vecteur $\overline{v}_2$ est presque dans le plan défini par x_1 et x_2. Comme $\alpha_{22} \gg \alpha_{32}$, il est même presque parallèle à x_2.

8. La matrice A est inversible puisque toutes ses valeurs propres sont non nulles. Un produit de matrices inversibles est inversible. Les colonnes d'une matrice inversible sont linéairement indépendantes.

8.6 Optimisations de l'algorithme $Q\,R$

8.6.1 Décalages et déflation

Si on s'intéresse à la deuxième matrice de la suite de matrices calculée précédemment on observe que la plus petite des valeurs propres (2) est obtenue avec une très bonne précision, alors que les approximations des autres valeurs propres $(-74, 38, 42)$ sont encore très mauvaises :

```
>>> A[2]
array([[-5.71030322e+01,  2.38703783e+01, -3.28358173e+01,  5.39483329e-02],
       [ 2.38703783e+01,  3.20083626e+01,  8.24140245e+00, -1.16355950e-01],
       [-3.28358173e+01,  8.24140245e+00,  3.10942282e+01, -6.93109231e-02],
       [ 5.39483329e-02, -1.16355950e-01, -6.93109231e-02,  2.00044136e+00]])
```

Il est même possible d'accentuer ce phénomène en effectuant les $N = 2$ itérations sur la matrice $A_0 - \mu\, I$ (pour $\mu = 1.99 \simeq 2$) au lieu de A_0. À la fin des deux itérations, il suffit d'effectuer le décalage inverse pour retrouver les approximations des valeurs propres de A :

```
>>> mu = 1.99
>>> A = [ A0 - mu * np.eye(4) ]
>>> for k in range (1,3) :
...     Qk, Rk = nla.qr (A[k-1])
...     Ak = np.dot (Rk, Qk)
...     A.append (Ak)
...
>>> A[2] + mu * np.eye(4)
array([[-6.10816594e+01,  2.14411886e+01, -2.91869609e+01,  1.30485286e-06],
       [ 2.14411886e+01,  3.33601446e+01,  6.31603402e+00, -3.12286419e-06],
       [-2.91869609e+01,  6.31603402e+00,  3.37215148e+01, -2.05633839e-06],
       [ 1.30485285e-06, -3.12286419e-06, -2.05633839e-06,  2.00000000e+00]])
```

Ce phénomène surprenant est dû au fait que l'algorithme $Q\,R$ « contient de façon cachée » la méthode de la puissance inverse par blocs : tout se passe comme si on avait exécuté la méthode de la puissance sur $(A_0 - \mu\, I)^{-1}$. La dernière colonne de P_N est très proche d'un vecteur propre associé à la valeur propre 2 de A_0. L'élément en bas à droite de A_N est un quotient de Rayleigh de cette colonne : il est très proche de la valeur propre. Les autres éléments situés sur la dernière ligne et la dernière colonne de A_N sont les produits scalaires de vecteurs presque orthogonaux : ils sont très proches de zéro. Ces observations suggèrent deux optimisations :

1. décaler à chaque itération la matrice A_k d'une valeur μ avant la factorisation $Q\,R$ de A_{k-1} puis effectuer le décalage inverse après le produit de R_k par Q_k

   Q_k, R_k = une factorisation $Q\,R$ de $A_{k-1} - \mu\, I$
   $A_k = R_k\, Q_k + \mu\, I$

On peut souvent prendre l'élément en bas à droite de A_{k-1} pour μ. Cette stratégie s'appelle le *Rayleigh quotient shift* en Anglais. Il existe d'autres stratégies, comme le *Wilkinson shift* [7, Lecture 29].

2. chercher à chaque itération si un élément sous-diagonal de A_k est très proche de zéro. Si c'est le cas, la matrice A_k peut être découpée en deux sous-matrices carrées et la recherche des valeurs propres peut être menée séparément sur les deux sous-matrices. Il arrive fréquemment que l'une de ces deux sous-matrices soit de dimension 1×1. On a alors trouvé une des valeurs propres recherchées. Cette stratégie, qui consiste à réduire la dimension du problème à traiter au fil des itérations, s'appelle une *déflation*.

8.6.2 Mise initiale sous forme de Hessenberg

La dernière optimisation consiste à mettre la matrice A sous forme de Hessenberg avant d'effectuer les itérations de l'algorithme $Q\,R$. Cette optimisation présente deux avantages :

1. elle accélère la convergence de l'algorithme $Q\,R$;
2. elle réduit le coût des factorisations $Q\,R$. En effet, les itérations de l'algorithme $Q\,R$ préservent les zéros introduits par la mise sous forme de Hessenberg. Les vecteurs colonnes considérés par l'algorithme de factorisation $Q\,R$ n'ont que deux éléments non nuls !

Une matrice est dite sous forme de Hessenberg si elle est de la forme suivante :

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{pmatrix}.$$

La technique utilisée est celle introduite pour la factorisation $Q\,R$ (réflexions de Householder). La différence tient au fait que la transformation recherchée doit être une similitude (sinon, on perdrait les valeurs propres de A) : à chaque fois qu'on multiplie à gauche par une matrice orthogonale Q_i, on multiplie aussi à droite par sa transposée (son inverse). Enfin, dans le cas où la matrice A est symétrique, la forme de Hessenberg l'est aussi : la matrice résultat est dite *tridiagonale*.

Une idée vient naturellement à l'esprit. Supposons que A soit symétrique et considérons la première matrice de réflexion Q_1 utilisée, pour la factorisation $Q\,R$, par la méthode de Householder (voir section 6.3).

$$\underset{A}{\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix}} \xrightarrow{Q_1} \underset{Q_1\,A}{\begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{pmatrix}}$$

Comme A est symétrique, on a également :

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix} \xrightarrow{Q_1^T} \begin{pmatrix} \times & & & & \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{pmatrix}$$
$$A \qquad\qquad A\,Q_1^T$$

Mais alors, est-ce qu'on ne pourrait pas calculer $Q_1\,A\,Q_1^T$, qui serait semblable à A, pour introduire des zéros à la fois sur la première ligne et la première colonne de A ? En répétant ce procédé $m-1$ fois, on aurait une diagonalisation de A !

Malheureusement, ça ne marche pas. On peut d'ailleurs s'en douter puisqu'on aurait alors un algorithme exact de diagonalisation de A. En fait, la multiplication à droite par Q_1^T efface les zéros introduits par la multiplication à gauche par Q_1, comme l'indique le diagramme de la figure 6.3, page 65 : les croix situées sur la première ligne de $Q_1\,A$ sont en caractères gras.

Toutefois, en observant de près ce diagramme, on voit que les croix situées sur la première ligne de $Q_2\,Q_1\,A$ sont en caractères normaux (voir aussi la figure 6.3, page 65), ce qui suggère le schéma d'algorithme donné Figure 8.8, qui vise un objectif un peu moins élevé : une matrice tridiagonale plutôt que diagonale.

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ \boxed{\times} & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \boxed{\times} & \times & \times & \times & \times \end{pmatrix} \xrightarrow{Q_1} \begin{pmatrix} \times & \mathbf{\times} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{\times} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \mathbf{0} & \boxed{\mathbf{\times}} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \mathbf{0} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ \mathbf{0} & \boxed{\mathbf{\times}} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \end{pmatrix} \xrightarrow{Q_2} \begin{pmatrix} \times & \times & & & \\ \times & \times & \mathbf{\times} & \mathbf{0} & \mathbf{0} \\ & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ & \mathbf{0} & \boxed{\mathbf{\times}} & \mathbf{\times} & \mathbf{\times} \\ & \mathbf{0} & \boxed{\mathbf{\times}} & \mathbf{\times} & \mathbf{\times} \end{pmatrix}$$
$$A \qquad\qquad Q_1\,A\,Q_1^T \qquad\qquad Q_2\,Q_1\,A\,Q_1^T\,Q_2^T$$

$$\xrightarrow{Q_3} \begin{pmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \mathbf{\times} & \mathbf{0} \\ & & \mathbf{\times} & \mathbf{\times} & \mathbf{\times} \\ & & \mathbf{0} & \mathbf{\times} & \mathbf{\times} \end{pmatrix}$$
$$Q_3\,Q_2\,Q_1\,A\,Q_1^T\,Q_2^T\,Q_3^T$$

FIGURE 8.8 – Transformations effectuées par l'algorithme de mise sous forme de Hessenberg sur une matrice symétrique A. Le résultat est une matrice tridiagonale, semblable à A. Les croix sont des nombres non nécessairement nuls, les caractères gras indiquent des entrées modifiées par la dernière transformation. À chaque itération, la matrice de réflexion de Householder F, qui sert à construire Q_i, est formée à partir du vecteur x encadré.

La Figure 8.9 contient le pseudo-code [7, Algorithm 26.1] d'un algorithme de mise sous forme de Hessenberg d'une matrice carrée A, fondé sur l'utilisation des matrices de réflexion

de Householder. Cet algorithme est une variante très proche de l'algorithme de la Figure 6.4, page 68.

Mise sous forme de Hessenberg
En entrée : la matrice A de dimension $m \times m$.
En sortie : A est sous forme de Hessenberg (tridiagonale si A est symétrique), les vecteurs $v_1, v_2, \ldots, v_{m-2}$ contiennent un codage de Q.
begin
for $k = 1$ to $m - 2$ do
$x = A_{k+1:m,k}$
$v_k = \text{sign}(x_1) \|x\|_2 e_1 + x$
$v_k = v_k / \|v_k\|_2$
$A_{k+1:m,k:m} = A_{k+1:m,k:m} - 2 v_k (v_k^T A_{k+1:m,k:m})$
$A_{1:m,k+1:m} = A_{1:m,k+1:m} - 2 (A_{1:m,k+1:m} v_k) v_k^T$
end do
end

FIGURE 8.9 – L'algorithme de mise sous forme de Hessenberg d'une matrice carrée A, non nécessairement symétrique. L'avant-dernière ligne correspond à la multiplication à gauche par Q_i. La dernière correspond à la multiplication à droite par Q_i^T. La multiplication à gauche tire parti du fait que des zéros ont été introduits sous la première sous-diagonale, aux itérations précédentes.

Les fonctions LAPACK correspondantes sont `DSYTRD` pour la maise sous forme tridiagonale d'une matrice symétrique et `DGEHRD` pour la mise sous forme de Hessenberg d'une matrice carrée quelconque.

En Python, la fonction `hessenberg` est disponible. On reprend l'exemple précédent.

```
>>> A0
array([[  2., -18., -38., -20.],
       [-18.,   2., -20., -38.],
       [-38., -20.,   2., -18.],
       [-20., -38., -18.,   2.]])
>>> H, Q = nla.hessenberg (A0, calc_q=True)
>>> H
array([[ 2.00000000e+00,  4.65617869e+01, -6.82500591e-15,  1.97645953e-15],
       [ 4.65617869e+01, -3.58597786e+01,  2.71041909e+01,  7.99360578e-15],
       [ 0.00000000e+00,  2.71041909e+01,  2.61690372e+00, -4.79325234e+00],
       [ 0.00000000e+00,  0.00000000e+00, -4.79325234e+00,  3.92428749e+01]])
>>> Q
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        , -0.3865831 ,  0.66443026, -0.63959826],
       [ 0.        , -0.81611988, -0.56946153, -0.09829501],
       [ 0.        , -0.42953678,  0.48398967,  0.76239895]])
```

On vérifie que $A_0 = Q\,H\,Q^T$.

```
>>> np.dot (Q, np.dot (H, np.transpose(Q)))
array([[  2., -18., -38., -20.],
       [-18.,   2., -20., -38.],
       [-38., -20.,   2., -18.],
       [-20., -38., -18.,   2.]])
```

On vérifie aussi que A_0 et H ont mêmes valeurs propres. On peut utiliser la fonction `eigh`, spécialisée pour les matrices symétriques aussi bien sur A_0 que sur H bien que le caractère symétrique de H ne soit pas très apparent, en raison d'erreurs d'arrondis.

```
>>> nla.eigh (A0, eigvals_only=True)
array([-74.,   2.,  38.,  42.])
>>> nla.eigh (H, eigvals_only=True)
array([-74.,   2.,  38.,  42.])
```

8.6.3 Algorithme $Q\,R$ avec optimisations

On suppose à nouveau, pour simplifier, que A est symétrique. Le pseudo-code donné Figure 8.10 correspond à [7, Algorithm 28.2]. Les optimisations importantes sont : mettre ini-

Algorithme $Q\,R$ avec shifts
A_0 = une tridiagonalisation de A (algorithme donné Figure 8.9)
for $k = 1, 2, \ldots$ do
  Choisir un shift μ_k (par exemple l'élément (m, m) de A_{k-1})
  $Q_k\,R_k$ = une factorisation $Q\,R$ de $A_{k-1} - \mu_k\,I$
  $A_k = R_k\,Q_k + \mu_k\,I$
  if un élément non diagonal $(j, j+1)$ de A_k est suffisamment proche de zéro then
    Mettre à zéro les éléments $(j, j+1)$ et $(j+1, j)$ de A_k, afin d'obtenir une partition $\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} = A_k\,.$
    Appliquer séparément l'algorithme $Q\,R$ avec shifts aux deux blocs B_1 et B_2
  end if
end do

FIGURE 8.10 – L'algorithme $Q\,R$ avec shifts est plus un schéma d'optimisation pour l'algorithme $Q\,R$ qu'un véritable algorithme

tialement la matrice sous forme de Hessenberg, utiliser des shifts (des décalages en Français) et, en particulier, dès qu'une valeur propre est trouvée, casser la matrice A_k en sous-matrices (procédé de *déflation*).

Comme l'utilisation de matrices orthogonales le laisse présager, l'algorithme $Q\,R$ avec shifts peut être rendu (pour être rigoureux, il faudrait préciser de nombreux détails) *backward*

stable [7, Theorem 29.1] et il est possible de calculer, avec une complexité en $(4/3)\,m^3$ (les deux tiers de la complexité d'un produit de matrices !), des approximations $\tilde{\lambda}_j$ des valeurs propres λ_j de A qui satisfont :

$$\frac{\tilde{\lambda}_j - \lambda_j}{\|A\|} = O(\varepsilon_{\text{machine}})\,.$$

La fonction LAPACK `DSYEV` permet de calculer une factorisation de Schur d'une matrice réelle symétrique.

Le code suivant contient un début de traduction FORTRAN de l'algorithme donné Figure 8.10. La matrice A est de dimension $m \times m$. Le tableau `LAMBDA` reçoit les valeurs propres. Le flottant `ESPLN` contient un petit réel $\varepsilon > 0$ qui représente la précision à atteindre.

La fonction utilise deux indices k_0 et k_1. À chaque itération, elle applique le mécanisme de déflation en cherchant un élément sous-diagonal inférieur à ε en valeur absolue. Elle isole ainsi des sous-matrices de A entre les indices k_0 et k_1. Deux cas peuvent alors se produire : soit $k_0 = k_1$, le bloc est de dimension 1×1 et une valeur propre est trouvée ; soit $k_0 < k_1$, le bloc est plus grand et on applique une itération de l'algorithme $Q\,R$. Invariant de boucle : toutes les valeurs propres situées sur la diagonale, entre les indices $k_1 + 1$ et m sont déjà calculées et stockées dans le tableau `LAMBDA`.

```
      SUBROUTINE SHIFT_QR (M, A, LDA, LAMBDA, EPSLN)
      IMPLICIT NONE
      INTEGER LDA, M
      DOUBLE PRECISION A(LDA,M), LAMBDA(M)
      DOUBLE PRECISION EPSLN
*
      INTEGER K0, K1
*
      Mettre A sous forme tridiagonale avec la fonction LAPACK DSYTRD
      K1 = M
      DO WHILE (K1 .GE. 1)
         K0 = K1
         DO WHILE (K0 .GE. 2 .AND. ABS(A(K0,K0-1)) .GT. EPSLN)
            K0 = K0-1
         END DO
         IF (K0 .EQ. K1) THEN
            WRITE (*,*) 'Valeur propre', A(K1,K1)
            LAMBDA (K1) = A(K1,K1)
            K1 = K1-1
         ELSE
            MU = A(K1,K1)
            Soustraire MU à la diagonale de A(K0..K1,K0..K1)
            Calculer une factorisation QR du bloc A(K0..K1,K0..K1)
                avec la fonction LAPACK DGEQRF
            Calculer A(K0..K1,K0..K1) = RQ avec la fonction LAPACK DORMQR
            Rajouter MU à la diagonale de A(K0..K1,K0..K1)
         END IF
      END DO
      END SUBROUTINE
```

FIGURE 8.11 – Squelette d'implantation FORTRAN de l'algorithme $Q\,R$ pour matrices symétriques avec mise initiale sous forme de Hessenberg, décalages et déflation. On pourrait optimiser le code davantage en remplaçant les appels à `DGEQRF` et `DORMQR` par du code spécialisé, tirant parti du fait que la matrice est tridiagonale.

Bibliographie

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.

[2] Ernst Hairer and Gerhard Wanner. Introduction à l'Analyse Numérique. Accessible sur `http://www.unige.ch/~hairer`, juin 2005.

[3] Julien Jacques. Analyse en composantes principales. `http://math.univ-lille1.fr/~jacques/Download/Cours/Cours-ACP.pdf`, 2013.

[4] Cristian Preda. Statistiques Exploratoires. Support du cours de IS 4, Polytech Lille, 2013.

[5] Gilbert W. Stewart. *Afternotes goes to Graduate School*. SIAM, 1998.

[6] Raluca Tanase and Remus Radu. The Mathematics of Web Search. `http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus`.

[7] Lloyd Nicholas Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

Table des figures

Index