METHODES NUMERIQUES

Manfred GILLI

Département d'économétrie
Université de Genève

Version : 25 mars 2006

Table des matières

1 Arithmétique en précision finie **1**
- 1.1 Représentation en virgule flottante . . . 2
  - 1.1.1 Normalisation . . . 2
  - 1.1.2 Underflow et Overflow . . . 4
- 1.2 Précision, erreurs d'arrondi . . . 5
  - 1.2.1 Précision machine . . . 6
  - 1.2.2 Digits significatifs . . . 8
- 1.3 Mesures de l'erreur . . . 8
  - 1.3.1 Erreur relative de $\mathit{float}(x)$ . . . 9

2 Instabilité numérique et condition d'un problème **11**
- 2.1 Condition d'un problème . . . 11
- 2.2 Condition d'une matrice . . . 12
- 2.3 Remarques . . . 13

3 Complexité des algorithmes **17**
- 3.1 Taille du problème . . . 17
- 3.2 Critères de comparaison . . . 17
- 3.3 La fonction $\mathcal{O}(\cdot)$ . . . 19
- 3.4 Opérations élémentaires (flops) . . . 20
- 3.5 Classification des algorithmes . . . 20

4 Resolution numérique de systèmes linéaires **23**

5 Systèmes triangulaires **27**
- 5.1 Forward substitution . . . 27
- 5.2 Back substitution . . . 28
- 5.3 Propriétés des matrices triangulaires unitaires . . . 28

6 Factorisation LU **31**
- 6.1 Formalisation de l'élimination de Gauss . . . 32
- 6.2 Matrices de permutation . . . 38
- 6.3 Pivotage partiel . . . 41
- 6.4 Considérations pratiques . . . 43
- 6.5 Elimination de Gauss-Jordan . . . 45

7 Matrices particulières **47**
7.1 Factorisation LDM' . . . 47
7.2 Factorisation LDL' . . . 50
7.3 Matrices symétriques définies positives . . . 51
7.4 Factorisation de Cholesky . . . 52

8 Matrices creuses **55**
8.1 Matrices par bande . . . 55
8.2 Matrices creuses irrégulières . . . 56
8.2.1 Représentation de matrices creuses . . . 56
8.2.2 Conversion entre représentation pleine et creuse . . . 58
8.2.3 Initialisation de matrices creuses . . . 59
8.2.4 Principales fonctions pour matrices creuses . . . 59
8.3 Propriétés structurelles des matrices creuses . . . 60
8.3.1 Exemples de matrices creuses . . . 63

9 Méthodes itératives stationnaires **67**
9.1 Méthodes de Jacobi et de Gauss-Seidel . . . 68
9.2 Interprétation géométrique . . . 69
9.3 Convergence de Jacobi et Gauss-Seidel . . . 71
9.4 Méthodes de relaxation (SOR) . . . 73
9.5 Structure des algorithmes itératifs . . . 74
9.6 Méthodes itératives par bloc . . . 75

10 Méthodes itératives non stationnaires **79**

11 Equations non-linéaires **81**
11.1 Equation à une variable . . . 82
11.1.1 Recoupement par intervalles (bracketing) . . . 86
11.1.2 Itérations de point fixe . . . 88
11.1.3 Bisection . . . 89
11.1.4 Méthode de Newton . . . 91
11.2 Systèmes d'équations . . . 92
11.2.1 Méthodes itératives . . . 92
11.2.2 Formalisation de la méthode du point fixe . . . 92
11.2.3 Méthode de Newton . . . 95
11.2.4 Quasi-Newton . . . 99
11.2.5 Newton amorti (*damped*) . . . 100
11.2.6 Solution par minimisation . . . 101
11.3 Tableau synoptique des méthodes . . . 101

12 Décompositions orthogonales **105**
12.1 La factorisation QR . . . 105
12.1.1 Matrices de Givens . . . 106
12.1.2 Réflexion de Householder . . . 106
12.1.3 Algorithme de Householder - Businger - Golub . . . 108

12.2 Décomposition singulière SVD 108
12.2.1 Approximation d'une matrice par une somme de matrices de rang un 109
12.2.2 Calcul numérique du rang d'une matrice 109
12.2.3 Interprétation des valeurs singulières 109
12.2.4 Complexité 110
12.2.5 Calcul du pseudoinverse 111

13 Moindres carrés **113**
13.1 Moindres carrés linéaires 114
13.1.1 Méthode des équations normales 114
13.1.2 Solution à partir de la factorisation QR 117
13.1.3 Décomposition à valeurs singulières SVD 118
13.1.4 Comparaison des méthodes 119
13.2 Moindres carrés non-linéaires 120
13.2.1 Méthode de Gauss-Newton 122
13.2.2 Méthode de Levenberg-Marquardt 123

Chapitre 1

Arithmétique en précision finie

Sources d'approximation dans le calcul numérique

Les deux sources d'erreur qui interviennent systématiquement dans le calcul numérique sont :

- Les *erreurs de troncature* ou de discrétisation qui proviennent de simplifications du modèle mathématique comme par exemple le remplacement d'une dérivée par une différence finie, le développement en série de Taylor limité, etc.
- Les *erreurs d'arrondi* qui proviennent du fait qu'il n'est pas possible de représenter (tous) les réels exactement dans un ordinateur.

Dans ce chapitre on expliquera d'abord quelle est l'approximation utilisée pour représenter les nombres réels dans un ordinateur et ensuite on discutera les conséquences de cette représentation inexacte.

Rappelons que dans un ordinateur le support pour toute information est un mot constitué par une séquence de bits (en général 32), un bit prenant la valeur 0 ou 1.

2^{31}	2^{30}							2^3	2^2	2^1	2^0
0	0	0	0	$\cdots$	$\cdots$	0	0	1	0	1	0

Cette séquence de bits est alors interprétée comme la représentation en base 2 d'un nombre entier.

Ainsi les nombres entiers peuvent être représentés exactement dans un ordinateur. Si tous les calculs peuvent se faire en nombres entiers, on parle *d'arithmétique en nombres entiers*. Ces calculs sont alors exacts.

Une division $\frac{1}{3}$ nécessite un nombre infini de digits pour représenter le résultat exactement. Dans la pratique, on recourra à *l'arithmétique en virgule flottante* pour représenter une approximation du résultat. Ainsi apparaît le problème des erreurs d'arrondis.

Remarque : Il existe une catégorie de logiciels (e.g. Maple) qui permettent de faire des calculs symboliques en traitant des expressions algébriques. Ces mêmes logiciels

effectuent des calculs, impliquant des nombres réels, avec une précision quelconque, limitée uniquement par la performance de l'ordinateur. Ceci ne permet évidemment pas de contourner les problèmes dans la plupart des calculs numériques.

1.1 Représentation en virgule flottante

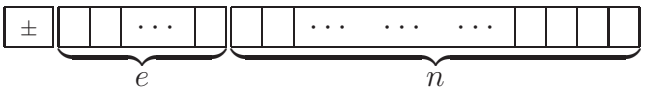
Dans un ordinateur, une variable réelle $x \neq 0$ est représentée de la façon suivante :

$$x = \pm n \times b^e$$

où n est la mantisse (ou partie fractionnelle), b la base et e est l'exposant. Dans un ordinateur, la base est toujours égale à 2.

Exemple 1.1 $12.153 = .75956 \times 2^4$ ou en base 10 on a $.12153 \times 10^2$

Pratiquement on partitionne le mot en deux parties, l'une contenant e et l'autre contenant n. (Le premier bit à gauche indique le signe).



Quelle que soit la taille du mot on ne disposera donc que d'un nombre limité de bits pour représenter les entiers n et e.

Soit t le nombre de bits disponibles pour coder la mantisse. En base 2, on peut alors coder les entiers allant de 0 à $\sum_{i=0}^{t-1} 2^i = 2^t - 1$.

Exemple 1.2 $t = 3$ $\boxed{0}\boxed{0}\boxed{0} = 0$ $\boxed{1}\boxed{1}\boxed{1} = 2^2 + 2^1 + 2^0 = 2^3 - 1 = 7$

1.1.1 Normalisation

Afin de maximiser le nombre de digits significatifs, on *normalise* la mantisse, c'est-à-dire on élimine les digits nuls à gauche.

Exemple 1.3 Soit $x = 0.00123456$. En base 10 et avec 5 digits on peut représenter ce nombre comme

$$x = \boxed{0}\boxed{0}\boxed{1}\boxed{2}\boxed{3} \times 10^{-5} \quad \text{ou} \quad \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5} \times 10^{-7}$$

La deuxième représentation exploite au maximum le nombre de digits disponibles.

Ainsi, pour un mot en base 2, on fait varier la mantisse n dans l'intervalle

$$\underbrace{\boxed{1}\boxed{0}\boxed{\cdots}\boxed{0}}_{2^{t-1}} \leq n \leq \underbrace{\boxed{1}\boxed{1}\boxed{\cdots}\boxed{1}}_{2^t-1}$$

En rajoutant 1 à droite (ce qui change le signe $\leq$ en $<$) et en multipliant la mantisse par 2^{-t}, la valeur normalisée de la mantisse n variera dans l'intervalle

$$\tfrac{1}{2} \leq n \times 2^{-t} < 1\,.$$

L'ensemble des nombres en virgule flottante F que l'on peut ainsi représenter constitue un sous-ensemble de $\mathbb{R}$. Si l'on représente la mantisse avec un mot de t bits, les éléments $f \in F$ sont définis par

$$f = \pm \boxed{1}\boxed{d_2}\boxed{\cdots}\boxed{d_t} \times 2^{-t} \times 2^e = \pm\, n \times 2^{e-t}$$

avec $d_i = 0$ ou 1 pour $i = 2, \ldots, t$ et d_1 vaut toujours 1 a cause de la normalisation.

Si l'exposant peut prendre les valeurs entières de l'intervalle $[L, U]$ on a que

$$m \leq |f| \leq M \qquad \text{avec} \quad m = 2^{L-1} \quad \text{et} \quad M = 2^U(1 - 2^{-t})\,.$$

Exemple 1.4 Construction de l'ensemble F pour $t = 3$ et $e \in [-1, 2]$.

$f = n \times 2^{e-t}$

n	2^{e-t}			
	$e=-1$	$e=0$	$e=1$	$e=2$
	1/16	1/8	1/4	1/2
$\boxed{1}\boxed{0}\boxed{0}$ $=2^2=4$	1/4	1/2	1	2
$\boxed{1}\boxed{0}\boxed{1}$ $=2^2+2^0=5$	5/16	5/8	5/4	5/2
$\boxed{1}\boxed{1}\boxed{0}$ $=2^2+2^1=6$	3/8	3/4	3/2	3
$\boxed{1}\boxed{1}\boxed{1}$ $=2^2+2^1+2^0=7$	7/16	7/8	7/4	7/2

En reportant ces nombres sur une droite on observe que les nombres en virgule flottante ne sont pas également espacés.

Pour décrire l'ensemble des nombres réels qui trouvent une représentation dans F on définit l'ensemble

$$G = \{x \in \mathbb{R} \text{ tel que } m \leq |x| \leq M\} \cup \{0\}$$

et l'opérateur $\mathit{float} : G \to F$ qui fait correspondre à un élément de G un élément de F comme

$$\mathit{float}\,(x) = \begin{cases} \text{plus proche } f \in F \text{ à } x \text{ qui satisfait } |f| \leq |x| \text{ (chopping)} \\ \text{plus proche } f \in F \text{ à } x \text{ (perfect rounding).} \end{cases}$$

Exemple 1.5 Soit l'ensemble F défini pour $t = 3$ et $e \in [-1, 2]$ dans l'exemple 1.4, alors avec chopping on a $\mathit{float}\,(3.4) = 3$ et $\mathit{float}\,(0.94) = 0.875$ et avec perfect rounding le résultat est $\mathit{float}\,(3.4) = 3.5$ et $\mathit{float}\,(0.94) = 1$.

1.1.2 Underflow et Overflow

Soit op l'une des 4 opérations arithmétiques $+ - \times \div$ et a, b deux nombres réels ; alors, si $|a \text{ op } b| \notin G$, on est dans une situation d'erreur (*overflow* si $|a \text{ op } b| > M$ et *underflow* si $|a \text{ op } b| < m$).

Matlab utilise des mots en double précision de 64 bits avec $t = 52$ le nombre de bits de la mantisse et $e \in [-1023, 1024]$ les valeurs possibles pour l'exposant. On a alors $m \approx 1.11 \times 10^{-308}$ et $M \approx 1.67 \times 10^{308}$. La fonction Matlab `realmin` et `realmax` produit m respectivement M.

Les 11 bits réservés à l'exposant permettent de représenter des entiers (positifs) de 0 à $2^{11} - 1 = 2047$. En soustrayant à cet entier la constante 1023, appelée *offset*, et qui est définie comme $o = 2^{s-1} - 1$ avec s le nombre de bits réservés pour l'exposant on peut représenter les entiers dans l'intervalle

$$(0 - o) \leq e \leq (2^s - 1 - o)$$

ce qui correspond à $0 - 1023 = -1023$ et $2047 - 1023 = 1024$ comme indiqué plus haut.

Dans le standard IEEE[1] les situations d'overflow et underflow ne provoquent pas l'arrêt des calculs. Un *overflow* produit le symbole `Inf` qui se propage dans la suite des calculs et un *underflow* peut produire le résultat zéro.

Ainsi une situation d'overflow est fatale pour la poursuite des calculs. Dans la pratique on peut cependant souvent transformer une situation d'overflow en une situation d'underflow sans conséquences pour la précision des calculs.

Exemple 1.6 Évaluons avec Matlab l'expression

$$c = \sqrt{a^2 + b^2}$$

avec $a = 10^{200}$ et $b = 1$. Le résultat attendu est $c = 10^{200}$ mais avec Matlab a^2 produit un overflow d'où $c =$ `Inf`. On peut éviter ce problème en procédant à une normalisation

$$c = s\sqrt{\left(\frac{a}{s}\right)^2 + \left(\frac{b}{s}\right)^2}$$

[1] Institute of Electrical and Electronics Engineers. Organisation professionnelle avec plus de 100 000 membres qui chapeaute 35 sociétés techniques dont la Computer Society.

avec $s = \max\{|a|, |b|\} = 10^{200}$, d'où

$$c = 10^{200}\sqrt{1^2 + \left(\frac{b}{10^{200}}\right)^2} = 10^{200}$$

car $(\frac{1}{10^{200}})^2$ produit un underflow avec la valeur zéro. Ceci ne gène cependant pas la précision de calculs étant donnée que la quantité 10^{-400} est insignifiante lorsque l'on additionne à 1.

1.2 Précision, erreurs d'arrondi

Avant de préciser la définition de précision d'un système en virgule flottante, illustrons avec un exemple les problèmes qui peuvent surgir lorsque les calculs se font en précision finie.

Exemple 1.7 Résolution numérique d'une équation du deuxième degré $ax^2+bx+c = 0$. Les solutions analytiques sont :

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

Algorithme 1 Transcription de la solution analytique.

1: $\Delta = \sqrt{b^2 - 4ac}$
2: $x_1 = (-b - \Delta)/(2a)$
3: $x_2 = (-b + \Delta)/(2a)$

Pour $a = 1$, $c = 2$ et une arithmétique à 5 digits significatifs l'algorithme 1 est appliqué ci-après pour des valeurs différentes de b.

b	Δ	*float* (Δ)	*float* (x_2)	x_2	erreur relative %
5.2123	4.378135	4.3781	−0.41708	−0.4170822	0.004
121.23	121.197	121.20	−0.01500	−0.0164998	9.1
1212.3	1212.2967	1212.3	0	−0.001649757	Catastrophe!!

Attention : *float* $(x_2) = (-b+$ *float* $(\Delta))/(2a)$

Ci-après on propose un autre algorithme qui exploite le théorème de Viète et qui évite le calcul de la différence de deux nombres qui est à l'origine de la perte de précision si ces nombres sont très proches.

Algorithme 2 Exploite le théorème de Viète $x_1 x_2 = c/a$.

1: $\Delta = \sqrt{b^2 - 4ac}$
2: **if** $b < 0$ **then**
3: $\quad x_1 = (-b + \Delta)/(2a)$
4: **else**
5: $\quad x_1 = (-b - \Delta)/(2a)$
6: **end if**
7: $x_2 = c/(a\, x_1)$

b	*float* (Δ)	*float* (x_1)	*float* (x_2)	x_2
1212.3	1212.3	-1212.3	-0.0016498	-0.001649757

Pour caractériser la précision d'un système de représentation des nombres réels en virgule flottante on présente souvent les mesures suivantes :

– Précision machine (*eps*)
– Digits significatifs
– Erreur d'arrondi (**u**)
– Erreur relative (ϵ)

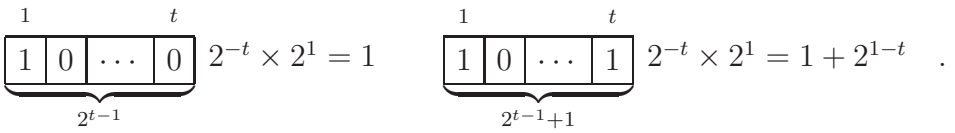
Bien que ces nombres diffèrent légèrement entre eux et dépendent du schéma d'arrondi (chopping ou perfect rounding), tous donnent une mesure qui caractérise la granularité du système en virgule flottante.

1.2.1 Précision machine

La *précision machine* d'une arithmétique en virgule flottante est définie comme le plus petit nombre positif *eps* tel que

$$\mathit{float}\,(1 + \mathit{eps}) > 1\ .$$

Regardons quelle est la valeur de *eps* pour une machine avec des mots dont la mantisse comporte t bits et lorsqu'on arrondit selon la technique dite chopping. Représentons le nombre 1 et le plus proche nombre suivant :

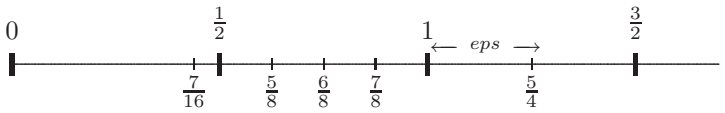


.

La distance qui sépare le nombre 1 du plus proche nombre suivant est égale à 2^{1-t}. Ainsi pour une mantisse comportant t bits on a

$$\mathit{eps} = 2^{1-t}\ .$$

Vérifions ce résultat sur notre mantisse avec $t = 3$ introduite avant :



On voit aussi, que dans une situation où on utilise perfect rounding, l'on obtient $eps = \frac{1}{2}2^{1-t} = 2^{-t}$.

Avec Matlab, qui utilise $t = 52$ bits pour la mantisse et perfect rounding, la précision machine est alors $eps = 2^{-52} \approx 2.22 \times 10^{-16}$.

Exemple 1.8 Montrons, que dans certains cas, l'approximation que fournit le résultat d'un calcul en virgule flottante peut être surprenant. Considérons l'expression

$$\sum_{k=1}^{\infty} \frac{1}{k} = \infty.$$

En calculant cette somme avec un ordinateur on obtient évidemment un résultat qui est fini. Ce qui peut cependant surprendre est que cette somme sera certainement inférieure à 100. Le problème ne vient pas d'un *underflow* de $1/k$ ou d'un *overflow* de la somme partielle $\sum_{k=1}^{n} 1/k$ mais du fait que pour n qui vérifie

$$\frac{1/n}{\sum_{k=1}^{n-1} \frac{1}{k}} < eps$$

la somme reste constante.

Pour le montrer considérons qu'il existe n pour lequel

$$\begin{aligned} float\left(\textstyle\sum_{k=1}^{n-1} \frac{1}{k} + \quad \frac{1}{n} \quad\right) &= \textstyle\sum_{k=1}^{n-1} \frac{1}{k} \\ float\left(\quad 1 \quad + \frac{1/n}{\sum_{k=1}^{n-1} \frac{1}{k}}\right) &= \quad 1\,. \end{aligned}$$

On peut facilement expérimenter que $\sum_{k=1}^{n-1} \frac{1}{k} < 100$ pour des valeurs de n qui peuvent être envisagées pour le calcul pratique et comme $eps \approx 2 \times 10^{-16}$ on établit que

$$\frac{1/n}{100} = 2 \times 10^{-16} \quad \rightarrow \quad n = \frac{10^{14}}{2}$$

d'où l'on trouve que n est d'ordre 10^{14}.

Considérant un ordinateur dont la performance est de 1 Gflops (10^9 flops) et qu'il faut trois opérations élémentaires par étape, la somme cessera de croître après $(3 \times 10^{14})/(2 \times 10^9) = 1.5 \times 10^5$ secondes, c'est-à-dire après quelques 42 heures de calcul sans avoir dépassé la valeur de 100.

1.2.2 Digits significatifs

La précision machine définit le nombre de digits significatifs d'un nombre réel dans sa représentation en virgule flottante. Dans un mot de 32 bits[2], on réserve en général 23 bits pour la mantisse ce qui donne $eps = 2^{-22} \approx 2.38 \times 10^{-7}$ et $1+eps = \underbrace{1.0000002}38$ ce qui donne 8 digits significatifs. Dans ce cas, il est inutile de lire ou imprimer des réels de plus que 8 digits.

Pour les illustrations, on choisira souvent la base 10 pour la définition de l'ensemble F. On aura alors

$$f = \pm\, d_1 d_2 \ldots d_t \times 10^{-t} \times 10^e$$

avec $d_1 \neq 0$ et $0 \leq d_i \leq 9$, $i = 2, \ldots t$. On utilise ensuite la représentation en point fixe dans laquelle seuls les premiers t digits sont significatifs. Voici à titre d'exemple, pour une base de 10 et $t = 3$, les digits significatifs : $\underbrace{2.37}$, $\underbrace{139}\,00$, $0.00\,\underbrace{293}\,7$.

1.3 Mesures de l'erreur

On peut envisager plusieurs façons pour mesurer l'erreur e entre une valeur approchée $\hat{x}$ et une valeur exacte x.

Erreur absolue

Elle est définie comme

$$|\hat{x} - x|.$$

Dans une situation avec $\hat{x} = 3$ et $x = 2$ l'erreur absolue vaut un, ce qui dans ce cas ne peut être considéré comme "petit". Par contre la même erreur absolue avec $\hat{x} = 10^9 + 1$ et $x = 10^9$ peut certainement être considérée comme relativement "petite" par rapport à x.

Erreur relative

La remarque précédente nous conduit à la définition de l'erreur relative

$$\frac{|\hat{x} - x|}{|x|}$$

qui est défini si $x \neq 0$ et pas définie si $x = 0$ (dans ce cas l'erreur absolue ferait bien l'affaire). Pour les exemples précédents l'erreur relative est respectivement 0.5 et 10^{-9} ce qui indique une "petite" erreur pour le deuxième cas.

[2]Ceci correspond à une précision simple, qui est en général le défaut pour la longueur d'un mot.

Combinaison entre erreur absolue et erreur relative

L'utilisation de l'erreur relative pose des problèmes lorsque x prends des valeurs qui sont proches de zero. Pour pallier à cet inconvénient on utilise souvent dans la pratique la mesure suivante :

$$\frac{|\hat{x} - x|}{|x| + 1} .$$

Cette mesure a les caractéristiques de l'erreur relative si $|x| \gg 1$, et les caractéristiques de l'erreur absolue si $|x| \ll 1$.

1.3.1 Erreur relative de *float*(x)

Considérons maintenant l'incrément de la mantisse dû au dernier bit à l'extrême droite. Cette valeur est appelée *roundoff error*, que l'on note

$$\mathbf{u} = 2^{-t} .$$

L'*erreur relative* ε due à l'arrondissement est

$$\frac{|\ \mathit{float}\ (x) - x|}{|x|} = \varepsilon\ .$$

Un résultat fondamental de l'arithmétique en virgule flottante est alors que quelque soit $x \in G$, l'erreur relative commise par l'approximation $\mathit{float}\ (x)$ est bornée comme

$$|\varepsilon| \leq \mathbf{u}$$

lorsqu'on utilise perfect rounding et $|\varepsilon| \leq 2\,\mathbf{u}$ pour chopping.

Démonstration 1.1 Soit une mantisse avec t bits. La plus grande erreur relative se produit lorsqu'on représente 1/2. On considère $x = 2^k = 2^{t-1} \times 2^{-t} \times 2^{k+1}$ et le nombre en virgule flottante qui le précède $(2^t - 1) \times 2^{-t} \times 2^k$. On a alors

$$\mathit{float}\ (x) - x = \underbrace{\boxed{1\ \ 1\ \ \cdots\ \ 1}}^{1 \qquad t}_{2^t - 1}\ 2^{-t} \times 2^k \quad - \quad \underbrace{\boxed{1\ \ 0\ \ \cdots\ \ 0}}^{1 \qquad t}_{2^{t-1}}\ 2^{-t} \times 2^{k+1} \quad = \quad 2^{-t+k}$$

et

$$\frac{|\ \mathit{float}\ (x) - x|}{|x|} = 2^{-t} .$$

□

La relation qui définit l'erreur relative peut aussi s'écrire

$$\mathit{float}\ (x) = x(1 + \varepsilon) \qquad \text{avec} \qquad |\varepsilon| \leq \mathbf{u}\,.$$

Cette relation est à la base de toute étude des erreurs d'arrondi.

Considérons maintenant les erreurs relatives associées aux opérations suivantes :

$$\begin{aligned} \mathit{float}\,(a \times b) &= (a \times b)(1+\varepsilon_1) \\ \mathit{float}\,(a \div b) &= (a \div b)(1+\varepsilon_2) \\ \mathit{float}\,(a \pm b) &= (a \pm b)(1+\varepsilon_3) \end{aligned}$$

On peut alors montrer que

$$|\varepsilon_1| \leq \mathbf{u} \quad \text{et} \quad |\varepsilon_2| \leq \mathbf{u}$$

alors que ε_3 n'est pas garanti d'être petit. C'est le cas notamment, lorsque l'addition de deux nombres très proches en valeur absolue (différence) donne un résultat très petit. Ceci est connu sous le nom de *catastrophic cancellation*.

Exemple 1.9 Soit l'opération $x = .123456 - .123465 = -.000009 = -9 \times 10^{-6}$. Si l'on choisit une représentation décimale avec $t = 5$ et "perfect rounding" on a

$$\mathit{float}\,(x) = \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5} \times 10^{-5} - \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{6} \times 10^{-5} = -\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1} \times 10^{-5}$$

$$\varepsilon_3 = \frac{|\mathit{float}\,(x) - x|}{|x|} = \frac{|-10 \times 10^{-6} + 9 \times 10^{-6}|}{|-9 \times 10^{-6}|} = \frac{1}{9} = .1111$$

L'*erreur absolue* est

$$|\mathit{float}\,(x) - x| = 10^{-6}$$

et vérifie

$$|\mathit{float}\,(x) - x| \leq \mathbf{u} = 10^{-5}$$

mais l'erreur relative ε_3 est très grande.

Un fait important est que l'arithmétique en virgule flottante n'est pas toujours associative.

Exemple 1.10 Pour le montrer, considérons une base de 10, $t = 3$ et l'expression $10^{-3} + 1 - 1$. On vérifie alors aisément que

$$\mathit{float}\,(\,\mathit{float}\,(10^{-3}+1) - 1) = 0 \qquad \text{et} \qquad \mathit{float}\,(10^{-3} + \mathit{float}\,(1-1)) = 10^{-3} \quad .$$

En effet pour l'addition les exposants des deux chiffres doivent être identiques ce qui implique l'abandon de la normalisation pour le chiffre le plus petit

$$\mathit{float}\,(10^{-3}+1) = \boxed{0}\boxed{0}\boxed{0} \times 10^{-3} \times 10^{1} + \boxed{1}\boxed{0}\boxed{0} \times 10^{-3} \times 10^{1} = 1\,.$$

C'est une conséquence du catastrophic cancellation.

Exemple 1.11 Calcul de e^{-a} pour $a > 0$ (Golub and Van Loan, 1989, p. 62). Golden mean (Press *et al.*, 1986, p. 18), récurrence instable, etc.

Chapitre 2

Instabilité numérique et condition d'un problème

Dans la section précédente, on a montré que l'on peut seulement représenter un sous-ensemble fini des réels dans un ordinateur, ce qui conduit à des erreurs d'arrondi. Comme on l'a vu, ces derniers peuvent, dans certains cas, provoquer l'instabilité de la solution ou même la destruction totale des résultats du calcul.

Si la précision des résultats n'est pas acceptable, il est important de distinguer les deux situations suivantes :

- Les erreurs d'arrondi sont considérablement grossies par la méthode de calcul. Dans ce cas, on parle d'une méthode ou d'un *algorithme numériquement instable*.
- Une petite perturbation des données peut engendrer un grand changement de la solution. Dans ce cas, on parle de *problème mal conditionné*.

Les exemples à la fin de la section précédente illustrent des cas d'algorithmes numériquement instables et proposent également les modifications à apporter afin que les erreurs d'arrondi ne jouent plus le même rôle destructeur pour la précision des calculs.

2.1 Condition d'un problème

Pour évaluer la sensibilité d'un problème à une perturbation des données, on définit *la condition du problème*. Soit un problème qui consiste à évaluer $f(x)$ et une perturbation des données $x + \delta x$. On considère alors le rapport

$$\frac{\frac{|f(x+\delta x)-f(x)|}{|f(x)|}}{|\frac{\delta x}{x}|} = \frac{|f(x+\delta x)-f(x)|}{|\delta x|} \times \frac{|x|}{|f(x)|}$$

et pour $\delta x \to 0$ on a

$$\text{cond}(f(x)) \approx \frac{|x\, f'(x)|}{|f(x)|}$$

la condition du problème $f(x)$, ce qui correspond à la valeur absolue de l'élasticité.

Dans le cadre d'un calcul numérique, il s'agit de l'élasticité de la solution de $f(x)$ par rapport aux données. Si l'élasticité est grande alors le problème est dit *mal conditionné*. Pour des problèmes pratiques il est le plus souvent très difficile d'évaluer cette élasticité. On peut distinguer 3 situations :

– $f'(x)$ très grand et x, $f(x)$ normales
– x très grand et $f(x)$, $f'(x)$ normales
– $f(x)$ très petit et x, $f'(x)$ normales

Considérons les deux matrices A et C et le vecteur b

$$A = \begin{bmatrix} -\frac{1}{2} & 1 \\ -\frac{1}{2}+d & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 2 & 4 \\ 1 & 2+e \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Si d est très petit, le système linéaire $Ax = b$ est un problème mal conditionné. Si par contre on considère une méthode qui chercherait la solution de x en résolvant le système transformé $CAx = Cb$, avec $d = \frac{3}{2}$ et e très petit, on a a faire à une méthode numériquement instable. La figure 2.1 illustre ces situations.

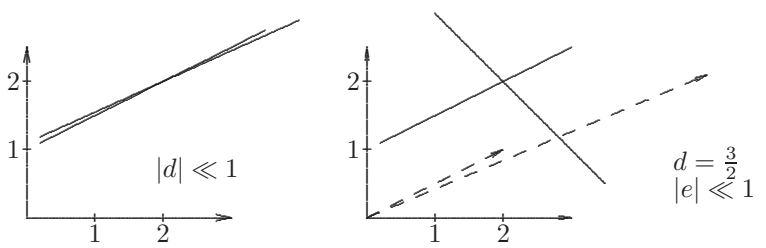


FIG. 2.1 – Illustration d'un problème mal conditionné et d'une méthode numériquement instable.

2.2 Condition d'une matrice

Nous discuterons ici comment évaluer la sensibilité d'un problème à une perturbation des données lorsqu'il s'agit du problème de la résolution d'un système linéaire $Ax = b$.

Exemple 2.1 Considérons un système linéaire $Ax = b$, avec

$$A = \begin{bmatrix} .780 & .563 \\ .913 & .659 \end{bmatrix} \qquad b = \begin{bmatrix} .217 \\ .254 \end{bmatrix}$$

où l'on vérifie facilement que la solution exacte est $x = [1 \ -1]'$. On rappelle que, dans la pratique, on ne connaît presque jamais la solution exacte. Avec Matlab, on obtient :

$$x = A\backslash b = \begin{bmatrix} .99999999991008 \\ -.99999999987542 \end{bmatrix}.$$

Ceci est un problème très mal conditionné. En effet considérons la matrice

$$E = \begin{bmatrix} .001 & .001 \\ -.002 & -.001 \end{bmatrix}$$

et résolvons le système perturbé $(A+E)x_E = b$. On obtient alors

$$x_E = (A+E)\backslash b = \begin{bmatrix} -5.0000 \\ 7.3085 \end{bmatrix} \quad .$$

Définition de la condition de A Soit un système linéaire $Ax = b$ et la perturbation E. On peut alors écrire

$$(A-E)(x+d) = b$$

d'où l'on tire la perturbation d engendrée pour x

$$d = (A-E)^{-1}Ex\,.$$

En appliquant $\|M\,v\| \leq \|M\| \cdot \|v\|$ pour une norme matricielle subordonnée à une norme vectorielle, on écrit

$$\|d\| \leq \|(A-E)^{-1}\|\,\|E\| \cdot \|x\|$$

et, en divisant par $\|x\|$ et en multipliant par $\frac{\|A\|}{\|E\|}$, on obtient

$$\frac{\frac{\|d\|}{\|x\|}}{\frac{\|E\|}{\|A\|}} \leq \|(A-E)^{-1}\| \cdot \|A\|$$

ce qui rappelle l'expression de l'élasticité. Pour $E \to 0$ on a

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

avec $\kappa(A)$ la *condition de la matrice A*.

Pour le moment, on ne discutera pas comment obtenir numériquement ces normes. La commande `cond` de Matlab donne la condition d'une matrice.

Pour notre exemple, on a $\text{cond}(A) = 2.2 \times 10^6$. Si $\text{cond}(A) > 1/\text{sqrt}(eps)$, il faut se méfier pour la suite des calculs. Pour $eps = 2.2 \times 10^{-16}$, on a $1/\text{sqrt}(eps) = 6.7 \times 10^8$. (On perd la moitié des digits significatifs.)

2.3 Remarques

La condition d'une matrice constitue une borne supérieure pour la mesure des problèmes que l'on peut rencontrer lors de la résolution d'un système linéaire. A titre d'exemple soit le système linéaire $Ax = b$ avec

$$A = \begin{bmatrix} 10^{-10} & 0 \\ 0 & 10^{10} \end{bmatrix}$$

et b quelconque. La matrice A vérifie $\texttt{cond}(A) = 10^{20}$ mais la solution $x_i = b_i/A_{ii}$ ne pose aucun problème numérique.

Soit un système linéaire $Ax = b$ et la solution calculée x_c. On appelle alors *erreur résiduelle* la quantité $r = Ax_c - b$. On imagine souvent que cette erreur résiduelle est un indicateur pour la précision des calculs. Ceci n'est le cas que si le problème est bien conditionné. Illustrons ce fait avec l'exemple précédent. Soit deux candidats x_c pour la solution et l'erreur résiduelle associée :

$$x_c = \begin{bmatrix} .999 \\ -1.001 \end{bmatrix} \quad r = \begin{bmatrix} -.0013 \\ -.0016 \end{bmatrix} \quad \text{et} \quad x_c = \begin{bmatrix} .341 \\ -.087 \end{bmatrix} \quad r = \begin{bmatrix} 10^{-6} \\ 0 \end{bmatrix}$$

On constate que le candidat nettement moins bon produit une erreur résiduelle nettement inférieure. Donc, pour un problème mal conditionné, l'erreur résiduelle n'est pas un indicateur utilisable.

Montrons avec deux exemples qu'une matrice peut être bien conditionnée pour la résolution du système linéaire associé et mal conditionnée pour la recherche des valeurs propres et vice versa.

Exemple 2.2 Soit $A = \texttt{triu(ones(20))}$. A est mal conditionnée pour le calcul des valeurs propres. Avec la fonction `eig` on obtient

$$\lambda_1, \ldots, \lambda_{20} = 1$$

et en posant $A_{20,1} = .001$ on obtient

$$\lambda_1 = 2.77, \ldots, \lambda_{20} = .57 - .08i \quad .$$

A est bien conditionnée pour la résolution du système linéaire, car $\text{cond}(A) = 26.03$.

Exemple 2.3 Soit la matrice

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1+\delta \end{bmatrix}$$

A est bien conditionnée pour le calcul des valeurs propres. Pour $\delta = 0$ on obtient les valeurs propres

$$\lambda_1 = 0 \text{ et } \lambda_2 = 2 \quad .$$

Pour $\delta = .001$ on obtient les valeurs propres

$$\lambda_1 = .0005 \text{ et } \lambda_2 = 2.0005 \quad .$$

A est mal conditionnée pour la résolution du système linéaire. Pour $\delta = 0$ A est singulière, avec $\text{cond}(A) = \infty$. Pour $\delta = .001$, on a $\text{cond}(A) = 4002$.

Pour conclure, on peut rappeler que si l'on veut obtenir des résultats numériquement précis avec un ordinateur, il faut :

– un problème bien conditionné,
– un algorithme qui est numériquement stable lorsque l'on exécute avec une précision arithmétique finie,
– un logiciel qui soit une bonne transcription de l'algorithme.

Un algorithme numériquement stable ne pourra pas résoudre un problème mal conditionné avec plus de précision que celle contenue dans les données. Cependant, un algorithme numériquement instable produira de mauvaises solutions pour des problèmes bien conditionnés.

Chapitre 3

Complexité des algorithmes

Ce chapitre aborde de manière succincte la problématique de la comparaison des algorithmes. Beaucoup de problèmes sont de nature à pouvoir être résolus par une variété d'algorithmes différents. Il se pose alors le problème du choix du meilleur algorithme. La notion de complexité d'un algorithme est utilisée pour comparer les performances des algorithmes. Une comparaison approfondie est souvent une tâche ardue et nous n'introduirons que des notions très générales.

3.1 Taille du problème

La notion de base en complexité des algorithmes est la définition de la taille du problème. Il s'agit naturellement de la quantité de données qui doivent être traitées. Pratiquement, la taille d'un problème est mesurée par la *longueur du plus petit codage* nécessaire à la représentation des données du problème (nombre d'éléments qu'il faut traiter). Par la suite, on tentera d'exprimer le temps nécessaire au déroulement de l'algorithme comme une fonction de la taille du problème.

Exemple 3.1 Soit le problème qui consiste à résoudre un système linéaire $Ax = b$. La taille du problème est alors caractérisée par l'ordre n de la matrice A si la matrice est dense. Dans le cas d'une matrice rectangulaire, la taille est définie par le nombre de lignes n et le nombre de colonnes m.

S'il s'agit d'analyser un graphe, les grandeurs caractérisant la taille du problème sont en général le nombre de sommets n et le nombre d'arcs m.

3.2 Critères de comparaison

Une comparaison détaillée de la performance de deux algorithmes est souvent très difficile. Beaucoup de critères peuvent être utilisés. Le principal critère est le temps

d'exécution de l'algorithme. Afin d'obtenir des résultats qui soient indépendants de la performance d'un ordinateur particulier (ou compilateur particulier), on mesure le temps en comptant le *nombre d'opérations élémentaires* exécutées.

On appelle *complexité de l'algorithme* le nombre d'opérations nécessaires pour résoudre un problème de taille n. Cette complexité est donc proportionnelle au temps d'exécution effectif.

Exemple 3.2 Soit un algorithme qui recherche de façon séquentielle un mot particulier dans une liste de n mots. La complexité est alors donnée par

$$C(n) \leq k_1\, n + k_2$$

avec $k_1 \geq 0$ et $k_2 \geq 0$, deux constantes indépendantes de n qui caractérisent une installation particulière.

On retient deux mesures pour compter le nombre d'opérations élémentaires pour caractériser la performance d'un algorithme, le pire des cas et le cas moyen.

Nombre d'opérations dans le pire des cas Il s'agit d'une borne supérieure du nombre d'opérations nécessaires pour résoudre un problème de taille n dans sa configuration la plus défavorable pour l'algorithme étudié.

Exemple 3.3 Pour l'algorithme séquentiel de l'exemple précédent, c'est la situation ou le mot recherché se trouve en dernière position dans la liste. On a $C(n) = k_1\, n + k_2$.

Nombre moyen d'opérations Souvent des algorithmes qui sont très mauvais pour le pire des cas se comportent très bien en moyenne[1]. La détermination du nombre moyen d'opérations nécessite souvent un analyse mathématique sophistiquée et, la plupart du temps, on recourt simplement à l'expérimentation.

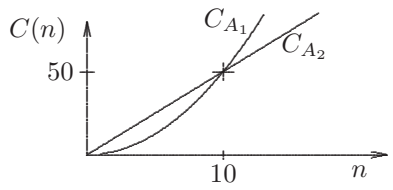
Autres critères D'autres critères sont la *place mémoire nécessaire* à la résolution du problème (space complexity) et la *simplicité de la mise en oeuvre* d'un algorithme.

Pour des ordinateurs à architecture parallèle, la quantité et la fréquence des communications entre processeurs sont des facteurs extrêmement importants. Le temps nécessaire pour communiquer des données est souvent de plusieurs ordres de grandeur plus élevé que le temps nécessaire pour effectuer des opérations élémentaires dans un processeur.

[1] L'algorithme du simplexe pour les problèmes de programmation linéaire constitue un exemple frappant étant donné que dans le pire des cas il nécessite 2^m itérations, avec m le nombre de contraintes, et que l'on observe que dans la pratique ce nombre n'excède en général pas $3m/2$.

3.3 La fonction $\mathcal{O}(\cdot)$

Soient deux algorithmes A_1 et A_2 de complexité $C_{A_1}(n) = \frac{1}{2}n^2$ et $C_{A_2}(n) = 5n$, alors l'algorithme A_2 est plus rapide que A_1 pour $n > 10$. En fait quels que soient les coefficients de n^2 et n, il existe toujours un n à partir duquel A_2 est plus rapide que A_1.



Définition 3.1 Une fonction $g(n)$ est dite $O(f(n))$ s'il existent des constantes c_o et n_o telles que $g(n)$ est inférieure à $c_o f(n)$ pour tout $n > n_o$.

Ainsi l'algorithme A_1 est d'ordre $O(n^2)$ et l'algorithme A_2 est d'ordre $O(n)$.

La notion d'ordre est donc proportionnelle au temps d'exécution sans être encombré de caractéristiques particulières d'une machine tout en restant indépendante d'un input particulier.

Exemple 3.4 Un algorithme qui nécessite $\frac{n^3}{3} + n^2 + \frac{2}{3}n$ opérations élémentaires a une complexité d'ordre $O(n^3)$.

Exemple 3.5 Soit l'algorithme récursif `maxmin` qui recherche l'élément maximum et l'élément minimum d'un vecteur de 2^n éléments.

```
function [a,b]=maxmin(s)
if length(s)==2
     if s(1) > s(2), a = s(1); b=s(2); else, a = s(2); b=s(1); end
  else
     k=length(s);
     [a1,b1]=maxmin(s(1:k/2));
     [a2,b2]=maxmin(s((k/2)+1:k));
     if a1 > a2, a=a1; else, a=a2; end
     if b1 < b2, b=b1; else, b=b2; end
end
```

On désire établir sa complexité en comptant le nombre de comparaisons effectuées. Le nombre de comparaisons en fonction de n est donné par la récurrence suivante :

$$C_{2^n} = \begin{cases} 1 & \text{pour} \quad n = 1 \\ 2\,C_{2^{n-1}} + 2 & \text{pour} \quad n > 1 \end{cases} .$$

Résolvons alors cette récurrence :

$$\begin{aligned}
C_{2^1} &= 1 \\
C_{2^2} &= 2\,C_{2^1} + 2 = 2 + 2 \\
C_{2^3} &= 2\,C_{2^2} + 2 = 2^2 + 2^2 + 2 \\
C_{2^4} &= 2\,C_{2^3} + 2 = 2^3 + 2^3 + 2^2 + 2 \\
&\vdots \\
C_{2^n} &= 2\,C_{2^{n-1}} + 2 = 2^{n-1} + 2^{n-1} + 2^{n-2} + \cdots + 2^1 \\
&= 2^{n-1} + \sum_{k=1}^{n-1} 2^k = 2^{n-1} + 2^n - 2 = 2^n(1 + \frac{1}{2}) - 2 \\
&= \frac{3}{2}2^n - 2 \quad .
\end{aligned}$$

En posant $N = 2^n$ le nombre de comparaisons devient $\frac{3}{2}N - 2$ et la complexité de l'algorithme est $O(N)$.

3.4 Opérations élémentaires (flops)

Pour les algorithmes résolvant des problèmes du domaine de l'algèbre linéaire, on utilise souvent le *flops* (floating point operations) comme mesure de dénombrement des opérations élémentaires. Chacune des quatre opérations élémentaires – addition, soustraction, multiplication et division – compte pour un flop[2]. Soient les vecteurs x, $y \in \mathbb{R}^n$, $z \in \mathbb{R}^m$ et les matrices $A \in \mathbb{R}^{m\times r}$ et $B \in \mathbb{R}^{r\times n}$ on a alors :

Opération	flops	
$x + y$	n	
$x'x$	$2\,n$	$(n > 1)$
xz'	$n\,m$	
AB	$2\,m\,r\,n$	$(r > 1)$

3.5 Classification des algorithmes

La notion de complexité est très importante, car elle permet une classification des algorithmes et des problèmes. On distingue notamment deux classes, suivant que le nombre d'opérations élémentaires est une fonction :

– *polynômiale* de la taille du problème,

– *non-polynômiale* de la taille du problème.

Seuls les algorithmes appartenant à la première classe peuvent être qualifiés d'efficaces. Pour les problèmes de la deuxième classe, on ne peut garantir d'obtenir

[2]Dans certains ouvrages un flop est défini comme $z_i = \alpha x_i + y_i$.

la solution dans un intervalle de temps raisonnable. Ceci est illustré dans les tableaux 3.1 et 3.2.

Dans le tableau 3.1 on donne le nombre d'opérations élémentaires pour résoudre des problèmes de taille variable en fonction de différentes complexités couramment rencontrées. On trouve aussi l'information sur la relation entre opérations élémentaires et temps de calcul qui correspondent à une puissance de calcul de 10^6 opérations élémentaires par seconde (1 Mflops). On voit notamment l'accroissement extraordinaire du nombre d'opérations élémentaires (et du temps de calcul) lorsque la complexité du problème devient non-polynômiale.

TAB. 3.1 – Flops en fonction de la complexité.

Complexité	n (Taille du problème)			
	2	8	128	1024
$\log_2 n$	1	3	7	10
n	2	2^3	2^7	2^{10}
$n \log_2 n$	2	3×2^3	7×2^7	10×2^{10}
n^2	2^2	2^6	2^{14}	2^{20}
n^3	2^3	2^9	2^{21}	2^{30}
2^n	2^2	2^8	2^{128}	2^{1024}
$n!$	2	5×2^{13}	5×2^{714}	7×2^{8766}

1 Mflops $\approx 2^{26}$ flops/min, 2^{32} flops/h, 2^{36} flops/jour, 2^{45} flops/année, 2^{52} flops/siecle.

Le tableau 3.2 montre qu'un accroissement de la performance des ordinateurs a un effet négligeable sur le gain de la taille des problèmes qui peuvent être envisagés pour une solution. En effet les accroissements de performance que l'on peut s'attendre sont d'ordre polynomial et ne modifient pas l'ordre des temps de calcul.

TAB. 3.2 – Effet d'un accroissement de vitesse sur la taille du problème.

Complexité	Taille maximale	Accroissement de vitesse		
		8	128	1024
n	N_1	$8 \times N_1$	$128 \times N_1$	$1024 \times N_1$
n^2	N_2	$2.8 \times N_2$	$11.3 \times N_2$	$32 \times N_2$
2^n	N_3	$N_3 + 3$	$N_3 + 7$	$N_3 + 10$
8^n	N_4	$N_4 + 1$	$N_4 + 2.3$	$N_4 + 3.3$

A titre d'exemple, la performance de quelques ordinateurs et la date de leur mise sur le marché. On remarque qu'en 1985 le rapport de performance entre un ordinateur personnel et un ordinateur de haute performance (Cray) était de mille. Aussi voit-on que la performance des ordinateurs personnels a été multipliée par mille approximativement en quelque 15 ans.

Machine	Mflops	Année
8086/87 à 8 MHz	0.03	1985
Cray X–MP	33	1985
Pentium 133 MHz	12	1995
Pentium II 233 MHz	28	1997
Pentium III 500 MHz	81	1999
Pentium IV 2.8 GHz	1000	2003
Earth Simulator	30 TFlops	2003

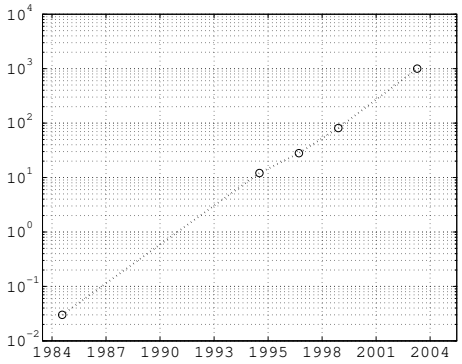


FIG. 3.1 – Evolution de la performance des PC.

Chapitre 4

Resolution numérique de systèmes linéaires

Le problème consiste à trouver la solution du système d'équations

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

En notation matricielle ce système s'écrit

$$Ax = b$$

et on rappelle qu'il existe une solution unique si et seulement si $\det(A) \neq 0$.

Rappel Soit $Ax = b$ avec $x \in \mathbb{R}^n$, $r = \text{rang}(A)$ et $B = [A\ b]$ avec $r' = \text{rang}(B)$. Alors solution de

$$Ax = b \quad \begin{cases} \text{est possible si } r = r' \\ \text{est unique si } r = r' = n \\ \text{comporte une infinité de solutions si } r = r' < n \end{cases} .$$

La solution d'un système linéaire se note

$$x = A^{-1}b$$

ce qui suggère que l'on a besoin de l'inverse de la matrice A pour calculer x. Ceci constitue une méthode numérique *très inefficace* (~ 3 fois plus de calculs que nécessaires et 2 fois plus d'espace mémoire pour moins de précision).

En général on n'a *quasiment jamais* besoin de l'inverse dans les problèmes d'algèbre linéaire. Il n'est pas non plus indiqué de calculer A^{-1} pour résoudre des systèmes

$$x^{(i)} = A^{-1}b^{(i)} \quad i = 1, \ldots, k$$

même si k est très grand.

En règle générale, si l'on n'a pas besoin d'analyser les éléments de A^{-1} il est très vraisemblable qu'on puisse et *doive* éviter le calcul de A^{-1} dans le problème d'algèbre linéaire considéré.

Le choix d'une méthode particulière pour la résolution d'un système linéaire dépendra également du type et de la structure de la matrice A intervenant dans le problème. La matrice A peut être caractérisée par une structure :

- creuse,
- diagonale,
- avec concentration des éléments non-nuls autour de la diagonale ($a_{ij} = 0$ si $|i - j| > b$, b est appelée la largeur de la bande),
- matrices par blocks,
- tridiagonale (cas particulier d'une matrice par bande avec $b = 1$),
- triangulaire,

ou par une quantification particulière, comme par exemple les matrices :

- symétriques ($A = A'$),
- définies positives ($x'Ax > 0, \quad \forall x \neq 0$),
- semi définies positives ($x'Ax \geq 0, \quad \forall x \neq 0$),
- matrices de Töplitz ($T \in \mathcal{R}^{n \times n}$ est une matrice de Töplitz s'il existe des scalaires $r_{-n+1}, \ldots, r_{-1}, r_0, r_1, \ldots, r_{n-1}$, tels que $T_{ij} = r_{j-i}$ quelque soit i et j),
- matrices Hessenberg (une matrice de Hessenberg est une matrice triangulaire plus une diagonale immédiatement adjacente non nulle),
- Hankel, Vandermonde, ...
- etc.

Le déterminant de A joue un rôle central dans les considérations théoriques, mais n'est *d'aucune utilité* pour la résolution numérique (sauf rares exceptions). La règle de Cramer

$$x_i = \frac{\det(A_i)}{\det(A)}$$

est *incroyablement inefficace* pour résoudre un système linéaire si l'on calcule le déterminant selon le schéma de l'expansion des mineurs.

Souvent on entend dire que l'on ne peut résoudre un système donné car le déterminant est nul. En utilisant des méthodes efficaces de calcul on découvre que le système ne peut être résolu *avant* d'avoir la valeur du déterminant.

La valeur du déterminant ne contient *aucune* information sur les problèmes numériques que l'on peut rencontrer en résolvant $Ax = b$.

Exemple 4.1 Soit deux matrices A et B d'ordre 40 et vérifiant la structure ci-après :

$$A = \begin{bmatrix} .1 & & & & \\ & .1 & & & \\ & & \ddots & & \\ & & & .1 & \\ & & & & .1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & -1 & \cdots & \cdots & -1 \\ & 1 & -1 & \cdots & -1 \\ & & \ddots & \ddots & \vdots \\ & & & 1 & -1 \\ & & & & 1 \end{bmatrix}$$

On a $\det(A) = 10^{-40}$ et $\det(B) = 1$. La matrice A ne pose évidemment aucun problème, alors que la matrice B est très mal conditionnée ($\text{cond}(B) \simeq 10^{13}$).

Multiplier les m équations du système $Ax = b$ par 100 modifie la valeur du déterminant de 100^m. Ceci change la valeur du déterminant sans modifier les calculs numériques.

Dans les chapitres qui suivent on discutera des techniques qui consistent à transformer le problème original en un problème ayant une forme particulière, pour laquelle la solution dévient triviale ou très simple. Par exemple, on transforme le système $Ax = b$ en $Ux = c$ avec U triangulaire (ou diagonale) et ainsi x est facile à calculer. Dans d'autres situations on peut manipuler $Ax = b$ de la sorte à obtenir le système $Rx = c$ avec R orthogonale, ce qui se résoud facilement par rapport à x.

Avertissement

Les algorithmes présentés dans les chapitres qui suivent ne sont pas spécifiques à un language ou une machine précise. Ils sont écrits dans un pseudo-language de programmation que l'on pourra traduire dans n'importe quel language de programmation formel. Ce pseudo-language est très proche du language Matlab sans toutefois respecter exactement la syntaxe de Matlab (p. ex. la produit AB est codé `A*B` dans Matlab, ou encore $A \neq B$ est codé `A~=B`).

Chapitre 5

Systèmes triangulaires

Beaucoup de méthodes de résolution transforment le système original en un système triangulaire qui possède la même solution. De ce fait on est souvent amené à manipuler des systèmes triangulaires dans les algorithmes. Ainsi on discutera d'abord la résolution des systèmes triangulaires.

Considérons le système triangulaire inférieur suivant :

$$\begin{bmatrix} \ell_{11} & 0 \\ \ell_{21} & \ell_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} .$$

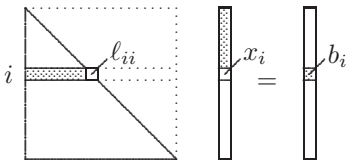
Si les éléments ℓ_{11}, $\ell_{22} \neq 0$, alors les inconnues peuvent être déterminés de façon séquentielle, c'est-à-dire

$$\begin{aligned} x_1 &= b_1/\ell_{11} \\ x_2 &= (b_2 - \ell_{21}x_1)/\ell_{22} . \end{aligned}$$

5.1 Forward substitution

Pour un système $Lx = b$, avec L triangulaire inférieure, la solution de la i-ème équation s'écrit :

$$x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij} x_j \right) / \ \ell_{ii}$$



Cette procédure s'appelle "*forward substitution*". On remarque que b_i n'est utilisé que pour le calcul de x_i (voir aussi la figure), ainsi on peut remplacer b_i par x_i. L'algorithme 3 résoud le système triangulaire inférieur $Lx = b$ d'ordre n par "*forward substitution*" et remplace b avec la solution x.

L'algorithme 3 nécessite $n^2 + 3n - 4$ flops. Sa complexité est donc d'ordre $O(n^2)$.

Algorithme 3 `fsub`

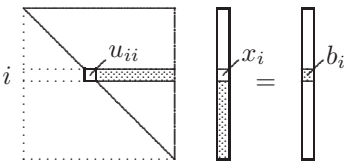
1: $b_1 = b_1/L_{1,1}$
2: **for** $i = 2 : n$ **do**
3: $\quad b_i = (b_i - L_{i,1:i-1}\, b_{1:i-1})/L_{ii}$
4: **end for**

Démonstration 5.1 1 flop pour la première instruction. Pour $i = 2$ on a 5 flops (1 flop pour le produit scalaire si les vecteurs sont de longueur 1, 2 flops pour calculer les indices et 2 flops pour la soustraction et la division). Pour $i > 2$ on a $\sum_{i=3}^{n}(2(i-1)+4) = n^2+3n-10$ flops ($2(i-1)$ flops pour le produit scalaire si les vecteurs sont de longueur > 1). Le total est alors $n^2 + 3n - 4$ flops pour $n \geq 3$.

5.2 Back substitution

On peut résoudre un système triangulaire supérieur $Ux = b$ de façon analogue à celle présentée avant, soit

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / \; u_{ii}$$



et que l'on appelle "*back-substitution*´´. On peut à nouveau remplacer b_i par x_i. L'algorithme 4 résoud le système triangulaire supérieur $Ux = b$ d'ordre n par "*back substitution*´´ et remplace b avec la solution x.

Algorithme 4 `bsub`

1: $b_n = b_n/U_{n,n}$
2: **for** $i = n-1 : -1 : 1$ **do**
3: $\quad b_i = (b_i - U_{i,i+1:n}\, b_{i+1:n})/U_{ii}$
4: **end for**

La complexité de l'algorithme 4 est d'ordre $O(n^2)$.

5.3 Propriétés des matrices triangulaires unitaires

Définition 5.1 On appelle une matrice triangulaire avec une diagonale unitaire une matrice triangulaire unitaire.

On rappelle ici quelques propriétés du produit et de l'inverse des matrices triangulaires.

– L'inverse d'une matrice triangulaire inférieure (supérieure) est triangulaire inférieur (supérieur).

– Le produit de deux matrices triangulaires inférieures (supérieures) est triangulaire inférieur (supérieure).
– L'inverse d'une matrice triangulaire inférieure (supérieure) unitaire est triangulaire inférieure (supérieure) unitaire.
– Le produit de deux matrices triangulaires inférieures (supérieures) unitaires est triangulaire inférieure (supérieure) unitaire.

Chapitre 6

Factorisation LU

C'est la méthode de prédilection pour la résolution de systèmes linéaires comportant une matrice A dense, sans structure et sans quantification particulière.

On vient de voir qu'il est très simple de résoudre des systèmes triangulaires. La transformation du système original en un système triangulaire se fait en choisissant des combinaisons linéaires appropriées des équations du système.

Exemple 6.1 Soit le système

$$\begin{array}{rcrcl} 3x_1 & + & 5x_2 & = & 9 \\ 6x_1 & + & 7x_2 & = & 4 \end{array}$$

si l'on multiplie la première équation par 2 et que l'on la soustrait de la deuxième on obtient

$$\begin{array}{rcrcl} 3x_1 & + & 5x_2 & = & 9 \\ & - & 3x_2 & = & -14 \end{array} .$$

Cette procédure s'appelle *élimination de Gauss*. Dans la suite on donnera à ce procédé une formulation matricielle, notamment en termes de factorisation d'une matrice. On présentera un algorithme qui factorise une matrice donnée A en une matrice triangulaire inférieure L et une matrice triangulaire supérieure U, tel que $A = LU$.

Exemple 6.2 Pour la matrice A correspondant à l'exemple précédent la factorisation $A = LU$ est :

$$\underbrace{\begin{bmatrix} 3 & 5 \\ 6 & 7 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 3 & 5 \\ 0 & -3 \end{bmatrix}}_{U} .$$

La solution du système original $Ax = b$ est alors obtenue en résolvant deux systèmes triangulaires successivement. On remplace A par sa factorisation

$$Ax = L \underbrace{Ux}_{y} = Ly = b$$

et l'on cherche la solution y à partir du système triangulaire inférieur $Ly = b$, puis la solution de x à partir du système triangulaire supérieur $Ux = y$.

6.1 Formalisation de l'élimination de Gauss

Il s'agit de formaliser une procédure qui transforme une matrice en une matrice triangulaire supérieure en éliminant, colonne après colonne, les éléments non nuls en dessous de la diagonale comme illustré dans l'exemple 6.3.

Exemple 6.3 Soit la matrice

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix}$$

que l'on veut transformer en une matrice triangulaire supérieure. On obtient cette forme en deux étapes. Lors de la première étape on soustrait 2 fois la première ligne de la deuxième ligne et 3 fois la première ligne de la troisième ligne pour obtenir la matrice

$$\begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix}.$$

Lors de la deuxième étape on transforme cette nouvelle matrice en soustrayant 2 fois la deuxième ligne de la troisième ligne pour obtenir la forme triangulaire recherchée.

$$\begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix}.$$

Matrice d'élimination M

Formulons la transformation en matrice triangulaire comme une transformation non-singulière d'un système linéaire. Considérons d'abord le vecteur à deux éléments $x = [x_1\ x_2]$, alors si $x_1 \neq 0$, on peut définir la transformation

$$\begin{bmatrix} 1 & 0 \\ -x_2/x_1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}.$$

De manière générale, étant donné un vecteur x avec $x_k \neq 0$, on peut annuler tout les éléments x_i, $i > k$, avec la transformation

$$M_k\, x = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -\tau_{k+1}^{(k)} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\tau_n^{(k)} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

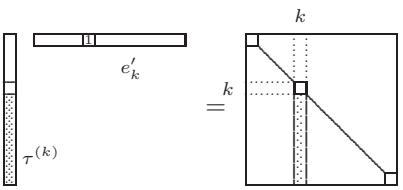
où $\tau_i^{(k)} = x_i/x_k$, $i = k+1, \ldots, n$. Le diviseur x_k est appelé le *pivot*, la matrice M_k est appelé la *matrice de transformation de Gauss* et les $\tau_i^{(k)}$ constituent les *multiplicateurs de Gauss*.

Propriétés des matrices M_k

Propriété 6.1 Les matrices M_k sont triangulaires inférieures avec diagonale unitaire et de ce fait elles sont non-singulières.

Propriété 6.2 Les matrices M_k peuvent s'écrire comme

$$M_k = I - \tau^{(k)} e_k'$$



avec $\tau^{(k)} = [0, \ldots, 0, \tau_{k+1}^{(k)}, \ldots, \tau_n^{(k)}]'$ et e_k la $k$ème colonne de la matrice identité.

Propriété 6.3 L'inverse de la matrice M_k s'écrit

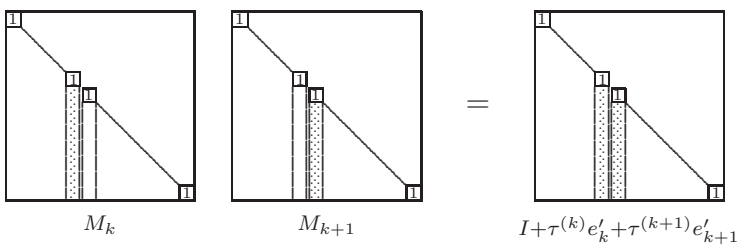
$$M_k^{-1} = I + \tau^{(k)} e_k'$$

qui peut être déduit directement de M_k en inversant simplement le signe des multiplicateurs de Gauss. On vérifie $(I - \tau^{(k)} e_k')(I + \tau^{(k)} e_k') = I - \tau^{(k)} e_k' + \tau^{(k)} e_k' - \tau^{(k)} e_k' \tau^{(k)} e_k' = I$ étant donné que $e_k' \tau^{(k)} = 0$.

Propriété 6.4 Soient deux matrices d'élimination M_k et M_i avec $i > k$, alors

$$M_k M_i = I - \tau^{(k)} e_k' - \tau^{(i)} e_i' + \tau^{(k)} e_k' \tau^{(i)} e_i' = I - \tau^{(k)} e_k' - \tau^{(i)} e_i'$$

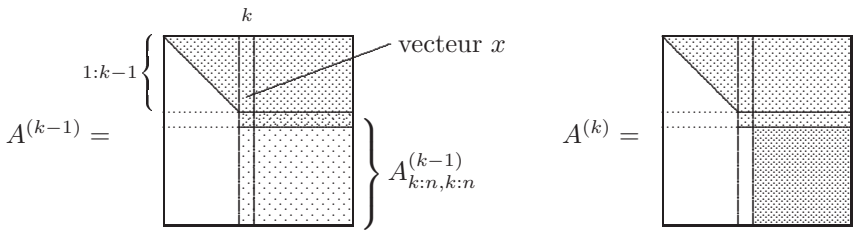
étant donné que $e_k' \tau^{(i)} = 0$. Ainsi le produit peut être considéré comme "l'union" des deux matrices.



La même propriété est vérifiée pour le produit $M_k^{-1} M_i^{-1}$.

Revenons à la matrice A à factoriser

Considérons maintenant la matrice $A^{(k-1)}$ dont les premières $k-1$ lignes ont déjà la forme triangulaire supérieure. Il s'agit alors de transformer la colonne k de la matrice, que nous désignerons ici comme le vecteur x, de la sorte que $x_i = 0$ pour $i = k+1, \ldots, n$.



Cette transformation s'opère en faisant le produit $M_k A^{(k-1)}$ et seulement les colonnes $j > k$ sont affectées par la transformation.

Le produit de M_k par la matrice $A^{(k-1)}$ s'avère être particulièrement simple. Seulement la matrice $A^{(k-1)}_{k:n,k:n}$ intervient dans cette transformation. Comme le résultat de la transformation de Gauss est connu pour la colonne $A^{(k)}_{k+1:n,k}$ (zéros), seuls les éléments de $A^{(k)}_{k+1:n,k+1:n}$ doivent être calculés.

En fait la transformation $M_k A^{(k-1)}$ peut être formulée comme une transformation M_1 appliquée à la sous-matrice $A^{(k-1)}_{k:n,k:n}$. L'algorithme 5 applique cette transformation M_1 à une matrice C.

Soit une matrice $C \in \mathbb{R}^{m\times m}$, l'algorithme 5 remplace $C_{2:m,1}$ par le vecteur des multiplicateurs $\tau^{(1)}$, et la sous-matrice $C_{2:m,2:m}$ par le produit $M_1 C$. Les éléments de $C_{1,1:m}$ restent inchangés.

Algorithme 5 (`tgm1`)

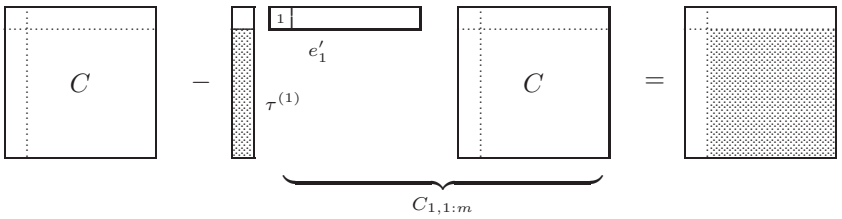
1: `function` $C = \texttt{tgm1}(C)$
2: $m = \texttt{size}(C, 1)$
3: **if** $C_{11} = 0$ **then** Pivot est nul, arrêt
4: $C_{2:m,1} = C_{2:m,1}/C_{11}$
5: $C_{2:m,2:m} = C_{2:m,2:m} - C_{2:m,1}\, C_{1,2:m}$

L'algorithme 5 nécessite $2(m-1)^2+m-1 = (2\,m-1)(m-1)$ opérations élémentaires. Les multiplicateurs conservés seront utilisés dans une étape ultérieure.

Détaillons l'algorithme 5. La transformation s'écrit

$$M_1 C = (I - \tau^{(1)} e_1')\, C = C - \tau^{(1)} e_1'\, C$$

ce qui correspond à



et ce qui explique l'instruction 5 de l'algorithme 5.

Exemple 6.4 Soit la matrice C à laquelle on applique la transformation M_1C.

$$C = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 5 & 6 \\ 6 & 7 & 8 \end{bmatrix}, \quad \text{alors} \quad M_1C = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 3 \\ 4 & 5 & 6 \\ 6 & 7 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 3 \\ 0 & 3 & 0 \\ 0 & 4 & -1 \end{bmatrix}$$

$$C_{2:3,1} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad C_{2:3,2:3} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} - \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 4 & -1 \end{bmatrix}$$

En appliquant $n-1$ fois dans l'ordre $M_1, \ldots, M_{n-1}$ la transformation de Gauss à une matrice A d'ordre n

$$M_{n-1} \cdots M_2 M_1 A = U$$

on obtient une matrice U qui est triangulaire supérieure. Ce procédé est appelé *élimination de Gauss*.

Exemple 6.5 Soit la matrice A de l'exemple du début de la section précédente

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -7 & 0 & 1 \end{bmatrix}, \quad \underbrace{M_1A}_{A^{(1)}} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix},$$

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix}, \quad \underbrace{M_2M_1A}_{A^{(2)}} = M_2A^{(1)} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix}.$$

Résumons l'étape k de l'élimination de Gauss :

- On a une matrice $A^{(k-1)} = M_{k-1} \cdots M_1 A$ qui a une forme triangulaire supérieure des colonnes 1 à $k-1$.
- Les multiplicateurs dans la matrice M_k sont calculés à partir du vecteur $A^{(k-1)}_{k+1:n,k}$. Pour qu'ils soient définis il faut que $A^{(k-1)}_{k,k} \neq 0$.
- L'élément $A^{(k-1)}_{k,k}$ qui intervient dans le calcul des multiplicateurs de Gauss $\tau^{(k)}_i = A^{(k-1)}_{i,k} / A^{(k-1)}_{k,k}$ est appelé le *pivot*. Le pivot doit être différent de zéro et sa grandeur relative est d'importance pour la précision des calculs.

Comment identifier la matrice L

A partir de l'élimination de Gauss

$$M_{n-1}\cdots M_1 A = U$$

qui conduit à une matrice triangulaire supérieure U on déduit, en écrivant

$$\underbrace{(M_{n-1}\cdots M_1)^{-1}(M_{n-1}\cdots M_1)}_{I}\ A = \underbrace{(M_{n-1}\cdots M_1)^{-1}}_{L}\ U$$

que la matrice L dans la factorisation $A = LU$ est constitué par le produit

$$L = (M_{n-1}\cdots M_1)^{-1} = M_1^{-1}\cdots M_{n-1}^{-1}\,.$$

A partir des propriétés 6.3 et 6.4 on vérifie aisément que

$$L = (I+\tau^{(1)}e_1')\cdots(I+\tau^{(n-1)}e_{n-1}') = I + \sum_{k=1}^{n-1}\tau^{(k)}e_k'$$

et que les éléments de la matrice L correspondent aux multiplicateurs de Gauss qui ont été conservés par l'algorithme 5 (instruction 4).

Existence de la factorisation LU

Il apparaît dans l'algorithme 5 que dans le cas d'un pivot nul, la factorisation n'existe pas toujours. Le théorème qui suit, montre qu'un pivot nul s'identifie avec une sous-matrice diagonale singulière.

Théorème 6.1 Existence : Une matrice A d'ordre n admet une factorisation LU si elle vérifie $\det(A_{1:k,1:k}) \neq 0$ pour $k = 1,\ldots,n-1$. Unicité : Si la factorisation existe et A est non-singulière, alors elle est unique et $\det(A) = \prod_{i=1}^{n} u_{ii}$.

Exemple 6.6 Matrice A non-singulière avec pivot nul.

$$\begin{bmatrix} 1 & 3 & 2 \\ 3 & 9 & 5 \\ 2 & 5 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

En développant le produit de la matrice triangulaire inférieure avec la matrice triangulaire supérieure on déduit que :

$$\begin{aligned} a_{11} = 1\cdot u_{11} = 1 &\ \Rightarrow\ u_{11} = 1 \\ a_{12} = 1\cdot u_{12} = 3 &\ \Rightarrow\ u_{12} = 3 \\ a_{21} = \ell_{21}u_{11} = 3 &\ \Rightarrow\ \ell_{21} = 3 \\ a_{22} = \ell_{21}u_{12} + 1\cdot u_{22} = 9 &\ \Rightarrow\ u_{22} = 0 \\ a_{31} = \ell_{31}u_{11} = 2 &\ \Rightarrow\ \ell_{31} = 2 \end{aligned}$$

Cependant le produit de la troisième ligne avec la deuxième colonne

$$a_{32} = 5 \neq \ell_{31}u_{12} + \ell_{32}u_{22} = 6$$

conduit à une contradiction. On a $\det(A) \neq 0$, mais $\det(A_{1:2,1:2}) = 0$.

Finalement pour une matrice A d'ordre n et vérifiant les conditions du théorème 6.1 la factorisation en une matrice triangulaire inférieure L et une matrice triangulaire supérieure U est obtenu avec l'algorithme 6. Cet algorithme remplace A avec U et L à l'exception des éléments de la diagonale de L (les éléments diagonaux de L valent 1).

Algorithme 6 (`eg`) Elimination de Gauss.

```
1: function  A = eg(A)
2: for k = 1 : n − 1 do
3:     A_{k:n,k:n} = tgm1(A_{k:n,k:n})
4: end for
```

L'algorithme 6 comporte $\sum_{k=1}^{n-1}\big(2(n{+}1{-}k){-}1\big)(n{-}k) = \frac{2}{3}n^3{-}\frac{1}{2}n^2{-}\frac{1}{6}n{+}1$ opérations élémentaires. Il s'agit de la formulation classique de l'élimination de Gauss.

On va montrer par la suite que cet algorithme n'est pas numériquement stable.

Choix du pivot

Lors de la présentation de l'élimination de Gauss on a défini le pivot. Il s'agit de l'élément a_{kk} de la matrice A au début de l'étape k. Pour pouvoir calculer les multiplicateurs de Gauss, le pivot doit être différent de zéro. Étant donné que les calculs se font avec une arithmétique en précision finie, la grandeur relative du pivot influence la précision des résultats.

Exemple 6.7 Soit le système

$$\begin{array}{rcrcrl} \boxed{0.0001} \times x_1 & + & 1 \times x_2 & = & 1 & (1) \\ 1.0000 \times x_1 & + & 1 \times x_2 & = & 2 & (2) \end{array}$$

pour lequel la solution exacte est $x_1 = 1.0001\ldots$ et $x_2 = 0.9998\ldots$. Appliquons l'élimination de Gauss avec une arithmétique en base 10 et $t = 3$.

On a $a_{11} = 0.0001$ le pivot et $\tau_1 = 1/.0001 = 10000$ le multiplicateur de Gauss. L'équation (2) devient alors

$$\begin{array}{rcrcrl} 1 \times x_1 & + & 1 \times x_2 & = & 2 & (2) \\ -1 \times x_1 & + & -10000 \times x_2 & = & -10000 & \text{eq. (1)} \times (-\tau_1) \\ \hline & & -9999 \times x_2 & = & -9998 & \end{array}$$

Avec 3 digits significatifs on obtient alors $x_2 = 1$. On remplace x_2 dans (1) et on obtient $x_1 = 0$. (Catastrophe numérique ! !).

Permutons les équations :

$$\begin{array}{rcrcrl} \boxed{1.0000} \times x_1 & + & 1 \times x_2 & = & 2 & (2) \\ 0.0001 \times x_1 & + & 1 \times x_2 & = & 1 & (1) \end{array}$$

On a $a_{11} = 1$ le pivot et $\tau_1 = .0001/1 = .0001$ le multiplicateur de Gauss. L'équation (1) devient maintenant

$$\begin{array}{rcrcrr} 0.0001 \times x_1 & + & 1.0000 \times x_2 & = & 1.0000 & (1) \\ -0.0001 \times x_1 & - & 0.0001 \times x_2 & = & -0.0002 & \text{eq. } (2) \times (-\tau_1) \\ \hline & & 0.9999 \times x_2 & = & 0.9998 & \end{array}$$

Avec 3 digits significatifs on obtient alors $x_2 = 1$. On remplace x_2 dans (2) et on obtient $x_1 + 1 = 2$ d'où $x_1 = 1$. (Excellent pour une précision de 3 digits ! !).

Les petits pivots sont à l'origine de coefficients relativement grands dans la matrice triangularisée. La factorisation de la matrice A de l'exemple précédent donne

$$A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 10000 & 1 \end{bmatrix} \begin{bmatrix} .0001 & 1 \\ 0 & -9999 \end{bmatrix} = LU \quad .$$

Lors de la résolution du système triangulaire on effectue alors des soustractions entre nombres relativement grands ce qui peut conduire à une perte de précision fatale. En appliquant la permutation $P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ on obtient

$$PA = \begin{bmatrix} 1 & 1 \\ .0001 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ .0001 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & .9999 \end{bmatrix} = LU$$

ce qui modifie l'amplitude des éléments dans la matrice U.

On peut rencontrer des pivots très petits sans que le problème soit mal conditionné (e.g. pour $A = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix}$ on a $\text{cond}(A) = 1$). Ainsi, l'élimination de Gauss, telle que présenté jusqu'ici, est un algorithme numériquement instable. Il est donc absolument nécessaire d'introduire dans l'algorithme un mécanisme de permutation des lignes et/ou des colonnes afin d'éviter des petits pivots.

6.2 Matrices de permutation

La permutation des lignes d'une matrice peut se formaliser en écriture matricielle avec une matrice de permutation qui n'est rien d'autre qu'une matrice d'identité avec des lignes réordonnées. La matrice suivante est un exemple de matrice de permutation :

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Le codage d'une matrice de permutation d'ordre n se fait à l'aide d'un vecteur p de longueur n. On aura $p(k)$ égal à l'indice de l'unique colonne non nulle de la ligne k. La matrice de l'exemple ci-dessus sera alors codé $p = [4\ 1\ 3\ 2]$.

Soit P une matrice de permutation, alors

– le produit PA permute les lignes de A,
– le produit AP permute les colonnes de A,
– P est orthogonale, $P^{-1} = P'$,
– le produit de deux matrices de permutation engendre une matrice de permutation.

Dans ce qui suit on sera plus particulièrement intéressé à la permutation de deux lignes (ou colonnes) d'une matrice. La matrice de permutation particulière qui correspond à une opération de ce type est la *matrice d'échange*. Il s'agit de la matrice d'identité dans laquelle on a échangé deux lignes, par exemple

$$E = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad .$$

Le produit EA échange la ligne 1 avec la ligne 4 dans A. Le produit AE échange la colonne 1 avec la colonne 4 dans A.

Pour la suite nous sommes particulièrement intéressés à échanger les lignes dans l'ordre, c'est-à-dire on échange d'abord la première ligne avec une autre ligne, puis la deuxième ligne avec une autre ligne, puis la troisième ligne, etc.

A titre d'illustration considérons un vecteur $x \in \mathbb{R}^4$ pour lequel on échange d'abord la ligne 1 avec la ligne 3, puis la ligne 2 avec la ligne 3 et finalement la ligne 3 avec la ligne 4. Le produit des matrices qui correspond à cet échange est :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_1 \\ x_4 \\ x_2 \end{bmatrix}$$

ce que peut être formalisé comme

$$E_3 E_2 E_1\, x$$

ou les matrices E_k sont définies à l'aide du vecteur $e(k)$, $k = 1, 2, 3$ comme :

E_k est la matrice d'identité ou la ligne k est échangé avec la ligne $e(k)$.

Pour notre exemple $e = [3\ 3\ 4]$. La matrice de permutation P qui correspond à ces trois échanges

$$\underbrace{E_3 E_2 E_1}_{P} x = Px$$

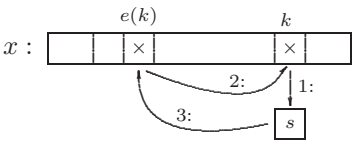
est définie par $p = [3\ 1\ 4\ 2]'$.

La permutation d'un vecteur $x \in \mathbb{R}^n$ qui correspond aux r échanges successifs $E_r \cdots E_1\, x$ et définis par le vecteur $e \in \mathbb{R}^r$ peut être obtenu par la procédure suivante qui remplace x :

Algorithme 7 (`perm`)

1: `function` $x = \texttt{perm}(e, x)$
2: **for** $k = 1 : r$ **do**
3: $\quad x(k) \leftrightarrow x(e(k))$
4: **end for**

Le symbole $\leftrightarrow$ signifie "échanger le contenu" :



1: $s = x(k)$
2: $x(k) = x(e(k))$
3: $x(e(k)) = s$

Le vecteur p qui définit la matrice de permutation P qui correspond aux échanges de lignes successifs définis par un vecteur e peut être obtenu avec la même procédure dans laquelle le vecteur x est remplacé par le vecteur $1 : n$, soit $p = \texttt{perm}(e, 1 : n)$.

Exemple 6.8 Soit le vecteur $x = [19\ 20\ 13\ 55]$ et le vecteur $e = [3\ 3\ 4]$. En appliquant la procédure `perm` à x et à un vecteur $[1\ 2 \cdots n]$ on obtient

		État du vecteur x			
k	$e(k)$	19	20	13	55
1	3	**13**	20	**19**	55
2	3	13	**19**	**20**	55
3	4	13	19	**55**	**20**

		État du vecteur $1 : n$			
k	$e(k)$	1	2	3	4
1	3	**3**	2	**1**	4
2	3	3	**1**	**2**	4
3	4	3	1	**4**	**2**

$$Px = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 19 \\ 20 \\ 13 \\ 55 \end{bmatrix} = \begin{bmatrix} 13 \\ 19 \\ 55 \\ 20 \end{bmatrix}.$$

Si pour une permutation donnée $y = E_r \cdots E_1\, x$ on veut revenir au vecteur original on doit procéder aux échanges des lignes dans l'ordre inverse :

$$x = (E_r \cdots E_1)^{-1} y\,.$$

Etant donné que les matrices E_k, $k = 1, \ldots, r$ sont orthogonales on vérifie $(E_r \cdots E_1)^{-1} = E_1' \cdots E_r'$. La procédure qui remplace x par le produit $(E_r \cdots E_1)^{-1}x$ sera :

for $k = r : -1 : 1$ **do**
  $x(k) \leftrightarrow x(e(k))$
end for

On peut facilement vérifier cette procédure avec l'exemple introduit plus haut.

6.3 Pivotage partiel

Montrons comment échanger les lignes lors de la factorisation LU de la sorte à ce qu'aucun multiplicateur ne devienne supérieur à 1 en valeur absolue.

Exemple 6.9 Soit la matrice A que l'on désire factoriser :

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}.$$

Pour obtenir les multiplicateurs les plus petits possibles a_{11} doit être l'élément le plus grand de la première colonne. Pour échanger la première ligne avec la troisième ligne on utilise la la matrice d'échange

$$E_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \text{et on forme le produit} \quad E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{bmatrix}.$$

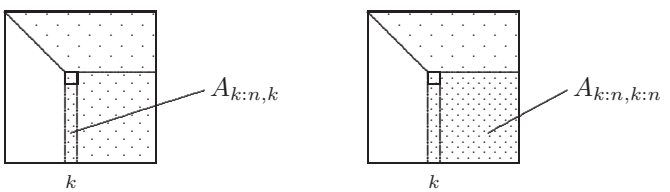
On calcule alors M_1 (tous les multiplicateurs sont inférieurs à 1 en valeur absolue)

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{3} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{bmatrix} \quad \Rightarrow \quad M_1 E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{bmatrix}.$$

Pour obtenir à nouveau les plus petits multiplicateurs on échange la ligne 2 avec la ligne 3. On a

$$E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{et} \quad M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{4} & 1 \end{bmatrix} \Rightarrow M_2 E_2 M_1 E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{bmatrix}.$$

La stratégie présentée dans cet exemple est appelée *pivotage partiel*. A chaque étape on recherche le maximum dans la colonne $A_{k:n,k}$. Une alternative est le pivotage complet, ou l'on recherche le maximum dans la matrice $A_{k:n,k:n}$.



La stratégie du pivotage partiel peut se résumer par les étapes suivantes :

for $k = 1 : n-1$ **do**
  Déterminer E_k tel que si z est la colonne k de $E_k A$, on a $|z_k| = \|z_{k:n}\|_\infty$
  Échanger les lignes
  Appliquer la transformation de Gauss
end for

L'algorithme transforme A en une matrice triangulaire supérieure U en effectuant les produits :

$$M_{n-1}E_{n-1}\cdots M_1E_1A = U\,.$$

Le pivotage partiel a comme conséquence qu'aucun multiplicateur n'est supérieur à un en valeur absolue.

L'algorithme 8 réalise l'élimination de Gauss avec pivotage partiel. Tous les multiplicateurs sont ≤ 1 en valeur absolue. $A_{1:k,k}$ est remplacé par $U_{1:k,k}$, $k = 1 : n$ et $A_{k+1:n,k}$ par une permutation de $-M_{k\ k+1:n,k}$, $k = 1 : n-1$. Le vecteur $e_{1:n-1}$ définit les matrices d'échange E_k. E_k échange la ligne k avec la ligne e_k, $k = 1 : n-1$.

Algorithme 8 (`egpp`) Elimination de Gauss avec pivotage partiel.

1: **for** $k = 1 : n-1$ **do**
2: Chercher p, $k \leq p \leq n$ tel que $|A_{pk}| = \|A_{k:n,k}\|_\infty$
3: $A_{k,1:n} \leftrightarrow A_{p,1:n}$
4: $e_k = p$
5: $A_{k:n,k:n} = \mathrm{tgm1}(A_{k:n,k:n})$
6: **end for**

L'algorithme 8 remplace la partie triangulaire supérieure de A avec U. La partie triangulaire inférieure contient les éléments de L mais dans le désordre étant donné les échanges de lignes effectués ($LU \neq A$).

Si l'élimination de Gauss avec pivotage partiel a été appliquée suivant l'algorithme 8 pour obtenir la forme triangulaire supérieure

$$\begin{aligned} M_{n-1}E_{n-1}M_{n-2}E_{n-2}\cdots M_1E_1A &= U \\ M_{n-1}M_{n-2}\cdots M_1 \underbrace{E_{n-1}E_{n-2}\cdots E_1}_{P} A &= U \end{aligned}$$

on a

$$PA = LU$$

avec $P = E_{n-1}\cdots E_1$ et L une matrice triangulaire inférieure unitaire avec $|\ell_{ij}| \leq 1$. La $k^{\text{ème}}$ colonne de L correspond au $k^{\text{ème}}$ vecteur de la transformation de Gauss qui a été permuté, c'est-à-dire si $M_k = I - \tau^{(k)}e_k'$ alors $L_{k+1:n,k} = g_{k+1:n}$ avec $g = E_{n-1}\cdots E_{k+1}\tau^{(k)}$.

Exemple 6.10 Si l'on applique l'algorithme 8 à la matrice A de l'exemple précédant on obtient

$$\begin{bmatrix} 6 & 18 & -12 \\ 1/2 & 8 & 16 \\ 1/3 & -1/4 & 6 \end{bmatrix} \quad et \quad e = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \quad \Rightarrow \quad P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad .$$

Avec Matlab la commande `[L,U] = lu(A)` retourne une matrice triangulaire inférieure L permutée. On a $LU = A$. La commande `[L,U,P] = lu(A)` retourne une matrice triangulaire L mais $LU = PA$.

$$L = \begin{bmatrix} 1 & & \\ 1/2 & 1 & \\ 1/3 & -1/4 & 1 \end{bmatrix}, \; U = \begin{bmatrix} 6 & 18 & -12 \\ & 8 & 16 \\ & & 6 \end{bmatrix} \quad et$$

$$PA = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix} = \begin{bmatrix} 6 & -18 & -12 \\ 3 & 17 & 10 \\ 2 & 4 & -2 \end{bmatrix}.$$

6.4 Considérations pratiques

Montrons comment résoudre des systèmes linéaires à partir d'une factorisation de matrice. Soit la collection de systèmes linéaires $AX = B$ avec $A \in \mathbb{R}^{n\times n}$, non-singulière et $B \in \mathbb{R}^{n\times q}$. En posant $X = [x_1 \dots x_q]$ et $B = [b_1 \dots b_q]$ les q solutions s'obtiennent avec la procédure :

Calculer $PA = LU$
for $k = 1 : q$ **do**
  Résoudre $Ly = Pb_k$
  Résoudre $Ux_k = y$
end for

Remarquons que l'on considère le système linéaire permuté $PAX = PB$ et la factorisation $PA = LU$. Cette procédure est programmée dans l'algorithme 9.

Algorithme 9 (`rsl`) Résolution d'un système d'équations linéaires $AX = B$

```
function B = rsl(A,B)
[n,q] = size(B)
[A,e] = egpp(A)
for k = 1 : q do
   B(1:n,k) = fsub1(A,perm(e,B(1:n,k)))
   B(1:n,k) = bsub(A,B(1:n,k))
end for
```

Les fonctions `egpp` et `bsub` correspondent aux algorithmes 8 et 4. La fonction `fsub1` correspond à une variante de l'algorithme 3 dans laquelle la matrice L est une matrice triangulaire unitaire.

On remarque que la matrice A n'est factorisée qu'une fois. Si $B = I_n$ alors la solution correspond à A^{-1}.

Considérons encore le système linéaire $A^k x = b$ avec $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ et k entier positif. Un possibilité serait de calculer $C = A^k$ puis de résoudre $Cx = b$. Ceci nécessite $2(k-1)n^3 + \frac{2}{3}n^3 - \frac{1}{2}n^2 + 2n^2 = (2k - \frac{1}{2})n^3 + \frac{3}{2}n^2$ flops. Pour montrer que l'on peut procéder plus efficacement considérons la solution du système linéaire $A^3 x = b$. On peut alors écrire

$$\begin{aligned} A\underbrace{AAx}_{b_1} &= b \\ A\underbrace{Ax}_{b_2} &= b_1 \\ Ax &= b_2 \end{aligned}$$

ce qui se formalise avec l'algorithme suivant :

Calculer $PA = LU$
for $i = 1 : k$ **do**
  Remplacer b avec la solution de $Ly = Pb$
  Remplacer b avec la solution de $Ux = b$
end for

Dans ce cas le nombre de flops est $\frac{2}{3}n^3 + k2n^2$.

Finalement montrons avec un exemple comment éviter le calcul de l'inverse lorsqu'il s'agit d'évaluer une expression du type $s = q'A^{-1}r$. On remarque que $A^{-1}r$ est la solution du système linéaire $Ax = r$. On résout alors ce système comme montré avant, puis on évalue $s = q'x$. Cet exemple montre qu'il faut toujours raisonner en termes d'équations linéaires à résoudre et non en termes d'inverse d'un matrice.

Revenons à la remarque faite à la page 23 qui disait qu'il ne fallait jamais recourir à l'inverse pour résoudre un système linéaire.

En ne considérant que les termes d'ordre le plus élevé dans le comptage du nombre d'opérations élémentaires, rappelons que la factorisation LU nécessite $\frac{2}{3}n^3$ opérations élémentaires et la solution d'un système triangulaire n^2 opérations élémentaires. Ainsi le nombre d'opérations élémentaires pour la résolution d'un système linéaire avec LU est d'ordre $\frac{2}{3}n^3$.

L'inverse A^{-1} s'obtient en résolvant n systèmes linéaires d'où elle nécessite $\frac{2}{3}n^3 + n2n^2 = \frac{8}{3}n^3$ opérations élémentaires et le produit $A^{-1}b$ nécessite n^2 opérations supplémentaires. Donc cette approche est quelque trois fois plus coûteuse qu'une solution avec LU. On vérifie aussi que la situation où l'on a $Ax = B$ avec un grand nombre de colonnes dans B ne modifie pas les conclusions qui précèdent.

Illustrons encore que la factorisation produit des résultats plus précis. Considérons le système $3x = 12$ composé d'une seule équation et la solution $x = \frac{12}{3} = 4$ obtenue par une transformation qui est équivalente à ce que produit la factorisation. Si l'on procède à une inversion explicite en utilisant une arithmétique à trois digits on a $x = 3^{-1} \times 12 = .333 \times 12 = 3.99$. On voit que l'inversion explicite nécessite une opération de plus pour aboutir à un résultat moins précis.

6.5 Elimination de Gauss-Jordan

L'élimination de Gauss visait la transformation du système original en un système triangulaire afin qu'il puisse être résolu plus facilement.

L'élimination de Gauss-Jordan est une variante qui transforme le système original en un système diagonal en annulant également les éléments qui se trouvent en dessus de la diagonale. La matrice de transformation pour un vecteur x donné est de la forme

$$\begin{bmatrix} 1 & \cdots & 0 & -\tau_1^{(k)} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & -\tau_{k-1}^{(k)} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & -\tau_{k+1}^{(k)} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & -\tau_n^{(k)} & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{k-1} \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

où $\tau_i^{(k)} = x_i/x_k$, $i = 1, \ldots, n$.

La transformation de Gauss-Jordan est environ 50% plus coûteuse que l'élimination de Gauss. Cependant lorsqu'on applique la transformation aussi au vecteur b la solution du système diagonal nécessite que n divisions. La méthode reste cependant globalement plus coûteuse. Un autre désavantage est qu'elle ne garantit pas la stabilité numérique, même lorsqu'on pivote.

Un système diagonal peut cependant être facilement résolu en parallèle ce qui actuellement confère à cette méthode un certain regain d'intérêt.

Chapitre 7

Matrices particulières

Lorsqu'on effectue des calculs numériques il est très important de tenir compte de la structure du problème afin de choisir l'algorithme le plus adapté pour sa resolution. On va notamment montrer que dans le cas de matrices symétriques, définies positives ou matrices par bande, il existe des variantes de la factorisation LU qui, pour de telles matrices, sont plus efficaces.

Dans ce chapitre on présentera d'abord une formalisation différente de la factorisation LU sous la forme d'un produit de trois matrices LDM'. Cette factorisation n'a pas d'intérêt pratique mais elle sera utile pour obtenir la factorisation LDL' d'une matrice symétrique. Finalement on discutera la situation très importante d'une matrice définie positive pour laquelle il existe une factorisation de la forme GG' appelée factorisation de Cholesky.

7.1 Factorisation LDM'

Montrons d'abord qu'il est possible de factoriser la matrice A en un produit de trois matrices

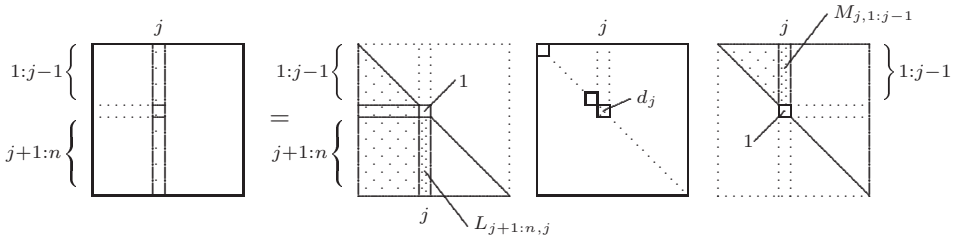
$$A = LDM'$$

avec D une matrice diagonale et L et M deux matrices triangulaires inférieures unitaires.

Théorème 7.1 Si toutes les sous-matrices diagonales de A sont non-singulières, alors il existe deux matrices triangulaires inférieures unitaires L et M et une matrice diagonale D tel que $A = LDM'$. Cette factorisation est unique.

Démonstration 1 A partir du théorème 6.1 on sait que A admet la factorisation $A = LU$. Posons $D = \text{diag}(d_1, \ldots, d_n)$ avec $d_i = u_{ii}$, $i = 1, \ldots, n$, alors D est non-singulière et $M' = D^{-1}U$ est triangulaire supérieure unitaire. Ainsi $A = LU = LD(D^{-1}U) = LDM'$. L'unicité découle de l'unicité de la factorisation LU.

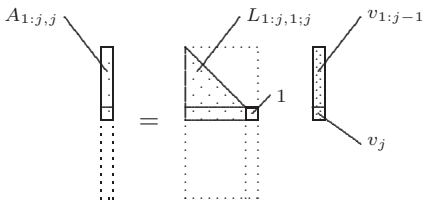
La factorisation LDM' peut donc être obtenu à partir de la factorisation LU. Il existe cependant un algorithme qui permet d'obtenir L, D et M directement. Supposons que l'on connaisse les $j-1$ premières colonnes de la factorisation $A = LDM'$:



Développons la $j^{\text{ème}}$ colonne de $A = LDM'$

$$A_{1:n,j} = Lv \quad \text{avec } v = DM'e_j \ .$$

Afin d'identifier les inconnues $L_{j+1:n,j}$, d_j et $M_{j,1:j-1}$ on résoud $A_{1:n,j} = Lv$ en deux étapes en considérant d'abord les j premières équations



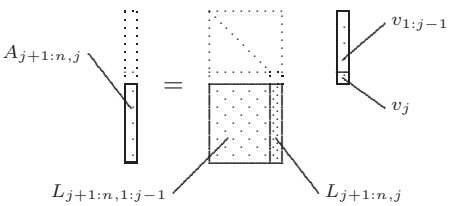
Les éléments $v_{1:j}$ sont la solution du système triangulaire

$$L_{1:j,1:j}\ v_{1:j} = A_{1:j,j} \ .$$

Étant donné que les d_i, pour $i = 1, \ldots, j-1$ sont connus et $m_{jj} = 1$ on peut calculer

$$\begin{aligned} m_{j,i} &= v_i/d_i \qquad i = 1, \ldots, j-1 \\ d_j &= v_j \ . \end{aligned}$$

Dans une deuxième étape on considère les $n-j$ equations restantes



qui permettent d'identifier le vecteur $L_{j+1:n,j}$. Ce vecteur est défini par le système

$$L_{j+1:n,1:j}\ v_{1:j} = A_{j+1:n,j}$$

que l'on réécrit sous la forme

$$L_{j+1:n,j}\ v_j = A_{j+1:n,j} - L_{j+1:n,1:j-1}\ v_{1:j-1}$$

d'où l'on obtient directement $L_{j+1:n,j}$ en divisant le membre de droite par v_j. La factorisation se résume alors par l'algorithme suivant :[1]

$d_1 = A_{11}$
$L_{2:n,1} = A_{2:n,1}/A_{11}$
for $j = 2 : n-1$ **do**
  `solve` $L_{1:j,1:j}\ v_{1:j} = A_{1:j,j}$ (Système triangulaire)
  $M_{j,1:j-1} = v_{1:j-1}./d_{1:j-1}$
  $d_j = v_j$
  $L_{j+1:n,j} = (A_{j+1:n,j} - L_{j+1:n,1:j-1}\ v_{1:j-1})/v_j$
end for
`solve` $L_{1:n,1:n}\ v_{1:n} = A_{1:n,n}$ (Système triangulaire)
$M_{n,1:n-1} = v_{1:n-1}./d_{1:n-1}$
$d_n = v_n$

Comme pour la factorisation LU, il est possible de formuler l'algorithme de sorte que A soit remplacé par L, D et M.

Soit $A \in \mathbb{R}^{n\times n}$ et admettant une factorisation LU, alors l'algorithme 10 produit la factorisation $A = LDM'$ et remplace A par L, D et M. On a $a_{ij} = l_{ij}$ pour $i > j$, $a_{ij} = d_i$ pour $i = j$ et $a_{ij} = m_{ji}$ pour $i < j$.

Algorithme 10 (`ldm`) Factorisation $A = LDM'$.

1: $A_{2:n,1} = A_{2:n,1}/A_{11}$
2: **for** $j = 2 : n-1$ **do**
3: $v_{1:j} = \texttt{fsub1}(A_{1:j,1:j}, A_{1:j,j})$
4: $A_{1:j-1,j} = v_{1:j-1}./\texttt{diag}(A_{1:j-1,1:j-1})$
5: $A_{jj} = v_j$
6: $A_{j+1:n,j} = (A_{j+1:n,j} - A_{j+1:n,1:j-1}\ v_{1:j-1})/v_j$
7: **end for**
8: $v_{1:n} = \texttt{fsub1}(A_{1:n,1:n}, A_{1:n,n})$
9: $A_{1:n-1,n} = v_{1:n-1}./\texttt{diag}(A_{1:n-1,1:n-1})$
10: $A_{n,n} = v_n$

Les instructions 1 et 6 remplacent A par L, en 4 et 9 on remplace A par M et en 5 et 10 c'est v qui remplace A. En 3 et 8 on résout un système triangulaire unitaire.

L'algorithme 10 nécessite $\frac{2}{3}n^3 + 2n^2 + \frac{10}{3}n - 9$ flops. Il est de même complexité que l'algorithme de factorisation LU.

Exemple 7.1 Factorisation d'une matrice A en LDM'. L'algorithme remplace A avec L, D et M'.

$$A = \begin{bmatrix} 3 & 3 & 6 \\ 9 & 15 & 42 \\ 12 & 54 & 194 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 4 & 7 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow A = \begin{bmatrix} 3 & 1 & 2 \\ 3 & 6 & 4 \\ 4 & 7 & 2 \end{bmatrix}$$

[1] Cette version sépare les étapes pour $j = 1$ et $j = n$ de l'étape générique. Voir les fonctions `ldm1`, `ldm2`, `ldm3` et `ldm4`.

On pourra vérifier que l'algorithme `eg` conduit à la même matrice L et que $U = DM'$.

7.2 Factorisation LDL'

Lorsque la matrice A est symétrique la factorisation LDM' comporte des opérations rédondantes.

Théorème 7.2 Si $A = LDM'$ est la factorisation d'une matrice non-singulière et symétrique A, alors $L = M$.

Démonstration 2 La matrice $M^{-1}A(M')^{-1} = M^{-1}LD$ est à la fois symétrique et triangulaire inférieure, donc diagonale. Comme D n'est pas singulière $M^{-1}L$ est aussi diagonale. Mais $M^{-1}L$ est aussi une matrice triangulaire unitaire et donc $M^{-1}L = I$.

Ainsi il sera possible d'économiser la moitié des opérations dans l'algorithme 10 si la matrice A est symétrique. En effet, à l'étape j on connaît déjà $M_{j,1:j-1}$ étant donné que $M = L$. Précédemment le vecteur v était défini comme $v = DM'e_j$ et comme $M = L$ le vecteur v devient

$$v = \begin{bmatrix} d_1\ L_{j,1} \\ \vdots \\ d_{j-1}\ L_{j,j-1} \\ d_j \end{bmatrix}.$$

Les premières $j-1$ composantes du vecteur v peuvent donc être obtenues par des simples produits terme à terme et v_j est obtenue à partir de l'équation $L_{1:j,1:j}\ v = A_{1:j,j}$, c'est-à-dire

$$v_j = A_{jj} - L_{j,1:j-1}\ v_{1:j-1}$$

ce qui donne l'algorithme :

for $j = 1 : n$ **do**
  for $i = 1 : j-1$ **do**
    $v_i = L_{j,i}\ d_i$
  end for
  $v_j = A_{jj} - L_{j,1:j-1}\ v_{1:j-1}$
  $d_j = v_j$
  $L_{j+1:n,j} = (A_{j+1:n,j} - L_{j+1:n,1:j-1}\ v_{1:j-1})/v_j$
end for

A nouveau il est possible de remplacer la matrice A avec le résultat. Soit $A \in \mathbb{R}^{n\times n}$, symétrique et admettant la factorisation LU alors l'algorithme 11 produit la factorisation $A = LDL'$ et remplace A. On a $a_{ij} = l_{ij}$ pour $i > j$ et $a_{ij} = d_i$ pour $i = j$.

En 6 on remplace A avec D et en 7 A est remplace par L. L'algorithme 11 nécessite $\frac{n^3}{3}$ flops, c'est-à-dire la moitié de ce que nécessite la factorisation LU.

Algorithme 11 (`ldl`) Factorisation $A = LDL$.

1: **for** $j = 1 : n$ **do**
2: **for** $i = 1 : j - 1$ **do**
3: $v_i = A_{j,i}\, A_{i,i}$
4: **end for**
5: $A_{jj} = A_{jj} - A_{j,1:j-1}\, v_{1:j-1}$
6: $A_{j+1:n,j} = (A_{j+1:n,j} - A_{j+1:n,1:j-1}\, v_{1:j-1})/v_j$
7: **end for**

Exemple 7.2 Factorisation d'une matrice symétrique A en LDL'. L'algorithme remplace A par L et D.

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 20 & 45 & 80 \\ 30 & 80 & 171 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow A = \begin{bmatrix} 10 & 20 & 30 \\ 2 & 5 & 80 \\ 3 & 4 & 1 \end{bmatrix}$$

7.3 Matrices symétriques définies positives

Les systèmes linéaires $Ax = b$ avec une matrice A définie positive constituent une classe, parmi les plus importantes, des systèmes linéaires particuliers. Rappelons qu'une matrice $A \in \mathbb{R}^{n\times n}$ est *définie positive* si quelque soit $x \in \mathbb{R}^n$, $x \neq 0$ on a $x'Ax > 0$.

Considérons le cas symétrique avec $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix}$. Si A est définie positive on a $x'Ax = a_{11}x_1^2 + 2a_{12}x_1x_2 + a_{22}x_2^2 > 0$ ce qui implique pour

$$\begin{aligned} x &= [1 \;\; 0]' \Rightarrow a_{11} > 0 \\ x &= [0 \;\; 1]' \Rightarrow a_{22} > 0 \\ x &= [1 \;\; 1]' \Rightarrow a_{11} + 2a_{12} + a_{22} > 0 \\ x &= [1 \;\; \text{-}1]' \Rightarrow a_{11} - 2a_{12} + a_{22} > 0 \end{aligned}$$

Les deux derniers résultats impliquent que $|a_{12}| \leq (a_{11} + a_{22})/2$ et que le plus grand élément de A se trouve sur la diagonale et qu'il est positifs. Il en résulte qu'une matrice symétrique et définie positive a des éléments diagonaux qui sont relativement grands ce qui rendra le pivotage inutile.

Propriété 7.1 Soit $A \in \mathbb{R}^{n\times n}$ et définie positive, alors A est non-singulière. Démonstration : Car sinon il existerait $x \neq 0$ tel que $x'Ax = 0$ et donc $Ax = 0$ ce qui n'est pas possible pour A non-singulière.

Propriété 7.2 Si $A \in \mathbb{R}^{n\times n}$ et définie positive et si $X \in \mathbb{R}^{n\times k}$ est de rang k alors $B = X'AX \in \mathbb{R}^{k\times k}$ est aussi définie positive.

Démonstration 3 Montrons que le contraire n'est pas possible. Soit un vecteur $z \neq 0$ et $z \in \mathbb{R}^k$ qui satisfait $0 \geq z'Bz = (Xz)'A(Xz)$ alors $Xz = 0$. Mais, le fait que les colonnes de X sont linéairement indépendantes implique $z = 0$.

Corollaire 1 Si A est définie positive, alors toutes les sous-matrices diagonales sont définies positives. En particulier tous les éléments diagonaux sont positifs.

Démonstration 4 Soit $v \in \mathbb{R}^k$ un vecteur entier avec $1 \leq v_1 < \ldots < v_k \leq n$, alors $X = I_n(:, v)$ est une matrice de rang k composée des colonnes $v_1, \ldots, v_k$ de la matrice d'identité. A partir de la propriété 7.2 il s'ensuit que $A_{v,v} = X'AX$ est définie positive.

Corollaire 2 Si A est définie positive, alors la factorisation $A = LDM'$ existe et tous les éléments de D sont positifs.

Démonstration 5 A partir du premier corollaire il découle que les sous-matrices $A_{1:k,1:k}$ sont non-singulières pour $k = 1, \ldots, n$. A partir du théorème 7.1 il découle que la factorisation $A = LDM'$ existe. Si l'on applique la propriété 7.2 avec $X = (L^{-1})'$ alors $B = L^{-1}LDM'(L^{-1})' = DM'(L^{-1})' = L^{-1}A(L^{-1})'$ est définie positive. Comme $M'(L^{-1})'$ est triangulaire supérieure unitaire, B et D ont la même diagonale, qui doit être positive.

Une situation pratique courante ou l'on rencontre des matrices définies positives est donnée lorsque $A = X'X$ avec toutes les colonnes de X linéairement indépendantes. (c.f. propriété 7.2 avec $A = I_n$).

7.4 Factorisation de Cholesky

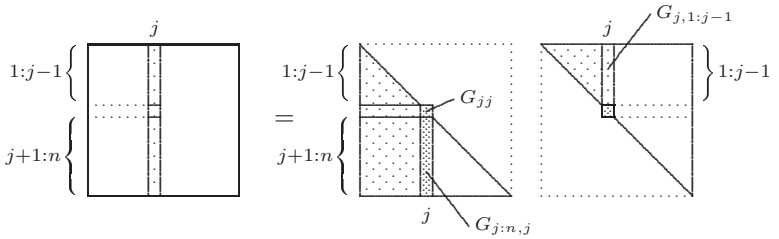
Nous savons que pour des matrices symétriques il existe la factorisation $A = LDL'$. Montrons qu'il existe encore une autre factorisation.

Théorème 7.3 (Factorisation de Cholesky) Si $A \in \mathbb{R}^{n \times n}$ est symétrique et définie positive, alors il existe une matrice triangulaire inférieure unique $G \in \mathbb{R}^{n \times n}$ avec les éléments diagonaux positifs et telle que $A = GG'$.

Démonstration 6 A partir du théorème 7.2 on sait que la factorisation $A = LDL'$ existe. Comme les d_k sont positifs, la matrice $G = L \operatorname{diag}(\sqrt{d_1}, \ldots, \sqrt{d_n})$ est une matrice réelle triangulaire inférieure avec diagonale positive. On a $A = GG'$. L'unicité découle de l'unicité de la factorisation LDL'.

La matrice G est appelle le *triangle de Cholesky*. La démonstration du théorème fournit une méthode pour obtenir G. Il existe cependant une méthode plus efficace pour obtenir G.

Considérons l'étape de la factorisation $A = GG'$ ou les premières $j - 1$ colonnes de G sont connus :



La $j^{\text{ème}}$ colonne du produit s'exprime alors sous la forme d'une combinaison des j colonnes de la matrice $G_{1:n,1:j}$, soit :

$$A_{1:n,j} = \sum_{k=1}^{j} G_{jk}\, G_{1:n,k}$$

d'ou l'on peut expliciter

$$G_{jj}\, G_{1:n,j} = A_{1:n,j} - \sum_{k=1}^{j-1} G_{jk}\, G_{1:n,k} \equiv v$$

et comme $G_{1:n,1:j-1}$ est connu, on peut calculer v. Considérons alors l'expression $G_{jj}\, G_{1:n,j} = v$. Les composantes du vecteur $G_{1:j-1,j}$ sont nulles car G est triangulaire inférieure. La $j^{\text{ème}}$ composante s'écrit $G_{jj}\, G_{jj} = v_j$ d'ou l'on tire $G_{jj} = \sqrt{v_j}$. La solution du vecteur $G_{j:n,j}$ s'écrit alors :

$$G_{j:n,j} = v_{j:n} / \sqrt{v_j}$$

ce qui conduit à formuler l'algorithme :

for $j = 1 : n$ **do**
  $v_{j:n} = A_{j:n,j}$
  for $k = 1 : j-1$ **do**
    $v_{j:n} = v_{j:n} - G_{jk}\, G_{j:n,k}$
  end for
  $G_{j:n,j} = v_{j:n} / \sqrt{v_j}$
end for

Il est possible de formuler l'algorithme de sorte à remplacer le triangle inférieur de la matrice A par la matrice G. Soit $A \in \mathbb{R}^{n\times n}$, symétrique et définie positive, alors l'algorithme 12 calcule une matrice triangulaire inférieure $G \in \mathbb{R}^{n\times n}$ telle que $A = GG'$. Pour $i \geq j$, l'algorithme remplace A_{ij} par G_{ij}.

Algorithme 12 (`cholesky`) Factorisation de Cholesky $A = GG'$.

1: **for** $j = 1 : n$ **do**
2: **if** $j > 1$ **then**
3: $A_{j:n,j} = A_{j:n,j} - A_{j:n,1:j-1}\, A'_{j,1:j-1}$
4: **end if**
5: $A_{j:n,j} = A_{j:n,j} \,/\, \sqrt{A_{jj}}$
6: **end for**

L'algorithme nécessite $\frac{n^3}{3} + n^2 + \frac{8}{3}n - 2$ flops. Ceci correspond à la moitié de ce que nécessite la factorisation LU. Il n'est pas nécessaire de pivoter.

Détail pour le calcul des flops

Instruction	Opération	Nombre d'opérations
		$j = 1$
5 :		$n + 1$
		$j > 1$
3 :	Produit	$2(n - j + 1)(j - 1)$
3 :	Addition	$n - j + 1$
3 :	Indices	2
3 :	Total	$2(n - j + 1)(j - 1) + n - j + 1 + 2 = (n - j + 1)\big(2(j - 1) + 1\big) + 2$
5 :		$(n - j + 1) + 1$
3 + 5 :		$(n - j + 1)\big(2(j - 1) + 2\big) + 3$

La somme des opérations élémentaires est $n+1+\sum_{j=2}^{n}\Big((n-j+1)\big(2(j-1)+2\big)+3\Big)$. Avec Maple on peut évaluer cette expression à l'aide des commandes :

```
> fl:= n+1 + sum('(n-j+1)*(2*(j-1)+2)+3','j'=2..n);
> simplify(fl);
```

Remarques

Avec Matlab on obtient la factorisation de Cholesky avec la commande `R=chol(A)` ou R est une matrice triangulaire supérieure d'ou $R'R = A$.

La factorisation de Cholesky s'utilise efficacement pour vérifier numériquement si une matrice est définie positive. Pour ce faire il suffit de vérifier qu'a chaque étape de l'algorithme a_{jj} est positif. Dans le cas contraire la matrice n'est pas définie positive.

Avec Matlab on peut appeler la factorisation de Cholesky avec un deuxième argument `[R,p] = chol(A)` qui indique l'élément diagonal qui est nul. Si p est nul la matrice est définie positive sinon seul la sous-matrice formé des $p - 1$ premieres lignes et colonnes est définie positive.

La décomposition de Cholesky est aussi utilisée pour la génération de variables aléatoires multidimensionelles x vérifiant une variance et covariance V donnée. On procède comme :

$$x = Ge$$

où G est la factorisation de Cholesky de la matrice V et $e \sim N(0, I)$. La matrice de variance et covariance de x vérifie alors

$$E(xx') = E(G \underbrace{ee'}_{I} G') = GG' = V\,.$$

La factorisation de Cholesky est aussi fréquemment utilisée pour la solution des équations normales.

Chapitre 8

Matrices creuses

8.1 Matrices par bande

Une matrice dont tout élément a_{ij} pour $|i-j| > b$ est nul est appelée une matrice par bande avec b la largeur de bande. Pour de telles matrices les techniques de factorisation vues avant peuvent être modifiés de façon à ne pas faire intervenir les éléments nuls.

Un cas particulier est constitué par une matrice tridiagonale pour laquelle on a $b = 1$. Dans la situation où il n'est pas nécessaire de pivoter pour assurer la stabilité numérique, la factorisation $LU = A$ s'écrit :

$$\begin{bmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & l_2 & 1 & & \\ & & l_3 & 1 & \\ & & & l_4 & 1 \end{bmatrix} \begin{bmatrix} u_1 & r_1 & & & \\ & u_2 & r_2 & & \\ & & u_3 & r_3 & \\ & & & u_4 & r_4 \\ & & & & u_5 \end{bmatrix} = \begin{bmatrix} d_1 & q_1 & & & \\ p_1 & d_2 & q_2 & & \\ & p_2 & d_3 & q_3 & \\ & & p_3 & d_4 & q_4 \\ & & & p_4 & d_5 \end{bmatrix} \tag{8.1}$$

L'algorithme 13 effectue une factorisation $A = LU$ d'une matrice tridiagonale A avec p_i, $i = 1, \ldots, n-1$ la diagonale inférieure, d_i, $i = 1, \ldots, n$ la diagonale et q_i, $i = 1, \ldots, n-1$ la diagonale supérieure. On vérifie que $r_i = q_i$, $i = 1, \ldots, n-1$.

Algorithme 13 (`lu3diag`) Factorisation LU d'une matrice tridiagonale A.

1: $u_1 = d_1$
2: **for** $i = 2 : n$ **do**
3: $\quad l_{i-1} = p_{i-1}/u_{i-1}$
4: $\quad u_i = d_i - l_{i-1} q_{i-1}$
5: **end for**

L'algorithme 13 nécessite $4\,n - 4$ opérations élémentaires.

8.2 Matrices creuses irrégulières

Définition d'une matrice creuse (J. H. Wilkinson) : Toute matrice avec suffisamment de zéros pour qu'il y ait un "gain" à les considérer. Le gain consiste ici à éviter des opérations redondantes avec des éléments nuls et surtout d'éviter à devoir les conserver en mémoire.

Dans la pratique on rencontre souvent des systèmes creux de grande taille. Sans traitement particulier ces problèmes resteraient impraticables.

Exemple 8.1 La discrétisation du Laplacien à 5 points avec une grille de 64×64 noeuds conduit à une matrice d'ordre 4096×4096 avec 20224 éléments non nuls. Le tableau qui suit indique la mémoire requise pour conserver la matrice et le temps nécessaire pour effectuer un produit et la solution d'un système linéaire lorsqu'on recourt à la technique traditionnelle et lorsqu'on exploite la structure creuse.

	Plein	Creux
Mémoire	128 Mbytes	0.25 Mbytes
Dx	30 sec	0.2 sec
$Dx = b$	> 12 h	10.0 sec

Lorsqu'on exploite efficacement la structure creuse d'une matrice comportant nnz éléments non nuls la complexité d'algorithmes faisant intervenir des produits et des resolutions de systèmes linéaires est $O(nnz)$ et ne fait pas intervenir l'ordre n des matrices.

8.2.1 Représentation de matrices creuses

Dans la suite nous mettrons l'accent sur l'approche réalisée dans Matlab pour le traitement de structures creuses. Pour l'utilisateur de Matlab le maniement de structures creuses est presque transparent.

On peut envisager de nombreuses façons pour représenter une matrice creuse et il n'existe pas de schéma qui soit le meilleur dans l'absolu. Le choix optimal dépend du type d'opérations effectuées sur les éléments de la matrice.

Matlab stocke les matrices qu'elles soient pleines ou creuses colonne par colonne. Une matrice creuse est représentée par la concatenation des éléments non nuls des vecteurs colonnes. Soit la matrice

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix}$$

L'information est stockée colonne par colonne

$$\begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix}$$

La commande `nnz` renseigne sur la nombre d'éléments non nuls et `nonzeros` empile les éléments non-nuls colonne après colonne.

```
>> nnz(A)
ans =
     4
>> nonzeros(A)
ans =
     2
     4
     1
     3
```

Avec la commande `find` on obtient le vecteur d'indices des éléments non nuls dans le vecteur des colonnes empilées.

```
>> find(A)
ans =
     2
     3
     4
     8
```

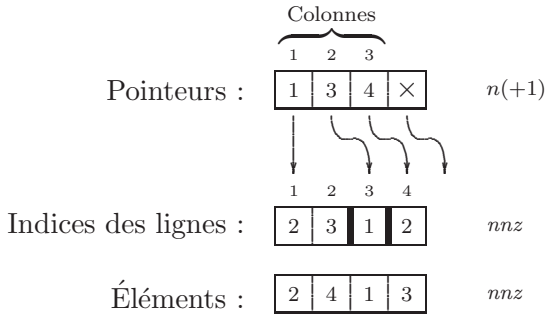
Pour obtenir l'indice de la ligne et de la colonne des éléments non nuls et leur valeur on exécute la commande

```
>> [i,j,e] = find(A)
i =
     2
     3
     1
     2
j =
     1
     1
     2
     3
```

```
e =
     2
     4
     1
     3
```

Dans Matlab une matrice creuse est représentée par la liste des éléments non nuls dans chaque colonne. Cette représentation nécessite un vecteur de pointeurs avec autant d'éléments que la matrice a de colonnes, une liste qui contient les indices de lignes correspondant aux nnz éléments non nuls et un vecteur de même longueur pour conserver la valeur des éléments.

Ci-après cette représentation pour la matrice creuse A de l'exemple précédent où $n = 3$ et $nnz = 4$.



Cette représentation nécessite $n(+1)$ mots entiers pour le vecteur de pointeurs, nnz mots entiers pour les listes des indices de lignes et nnz mots réels pour les éléments. Comme un entier nécessite 4 bytes et un réel 8 bytes, le total des bytes nécessaires à cette représentation est $12\,nnz + 4\,n$ bytes.

8.2.2 Conversion entre représentation pleine et creuse

Contrairement a ce qui est le cas pour la conversion de matrices réelles en matrices complexes, la conversion en matrice creuse ne se fait pas automatiquement. L'utilisateur doit donner une commande explicite. Cependant une fois initié, la représentation creuse se propage, c'est-à-dire que le résultat d'une opération avec une matrice creuse produit une matrice creuse. (Ceci n'est pas le cas pour une addition ou soustraction.) Les fonction `full` et `sparse` effectuent la conversion.

Conversion de la matrice A défini avant :

```
>> B = sparse(A)
B =
   (2,1)         2
   (3,1)         4
   (1,2)         1
   (2,3)         3
```

```
>> C = full(B)
C =
     0     1     0
     2     0     3
     4     0     0
>> whos
  Name       Size                     Bytes  Class

  A          3x3                         72  double array
  B          3x3                         64  double array (sparse)
  C          3x3                         72  double array
```

8.2.3 Initialisation de matrices creuses

Il est possible et indiqué de créer directement une matrice creuse (sans recourir à la conversion d'une matrice pleine) en donnant une liste d'éléments et les indices correspondants. La commande

```
S = sparse(i,j,s,m,n,nzmax);
```

crée une matrice creuse avec $S_{i(k),j(k)} = s(k)$. Les éléments dans les vecteurs d'indices i et j peuvent être données dans un ordre quelconque. Remarque importante : Si une même paire d'indices existe plus d'une fois les éléments correspondants sont additionnés. Si s ou l'un des arguments i, j est un scalaire ils s'ajustent automatiquement.

```
S = sparse(i,j,s,m,n);        % nzmax = length(s)
S = sparse(i,j,s);            % m = max(i)  n = max(j)
S = sparse(m,n);              % S = sparse([],[],[],n,m)
S = sparse([2 3 1 2],[1 1 2 3],[2 4 1 3])  % matrice A
```

8.2.4 Principales fonctions pour matrices creuses

```
help sparfun
speye(n,m)
sprandn(n,m,d)
spdiags
spalloc(n,m,nzmax)
issparse(S)
spfun                % C = spfun('exp',B)
spy(S)
[p,q,r,s] = dmperm(S)
condest(S)
eigs
sprank
```

8.3 Propriétés structurelles des matrices creuses

Pour des matrices creuses on peut mettre en évidence des propriétés intéressantes qui dépendent uniquement de la structure qualitative de la matrice, c'est-à-dire de l'information indiquant quels sont les éléments non nuls. Lors de la résolution de systèmes linéaires ou non-linéaires les propriétés structurelles des matrices définissant ces systèmes sont très importantes.

Pour étudier de telles propriétés il convient alors d'associer à une matrice sa matrice d'incidence. La matrice d'incidence M d'une matrice A est définie comme

$$m_{ij} = \begin{cases} 1 \text{ si } a_{ij} \neq 0 \\ 0 \text{ sinon} \end{cases} .$$

Dans la suite les propriétés structurelles d'une matrice seront étudiés en analysant sa matrice d'incidence[1].

Rang structurel d'une matrice

Rappelons que le déterminant d'une matrice $M \in \mathbb{R}^{n\times n}$ peut s'exprimer comme

$$\det M = \sum_{p\in P} \text{sign}(p)\, m_{1p_1}\, m_{2p_2} \cdots m_{np_n} \tag{8.2}$$

où P est l'ensemble des $n!$ permutations des éléments de l'ensemble $\{1, 2, \ldots, n\}$ et $\text{sign}(p)$ est une fonction qui prend des valeurs $+1$ et -1.

De la relation (8.2) on déduit qu'une condition nécessaire pour la non-singularité de la matrice M est l'existence d'au moins un produit non nul dans la somme de l'expression définissant le déterminant. Ceci correspond à l'existence de n éléments non nuls, chaque ligne et chaque colonne de M contenant exactement un de ces éléments.

Une formulation équivalente de cette condition consiste à dire qu'il doit exister une permutation des colonnes de la matrice M de sorte que la diagonale de la matrice ne comporte pas d'éléments nuls. On parle alors d'existence d'une *normalisation*.

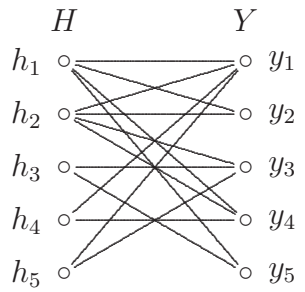
Si la matrice correspond à la matrice jacobienne d'un système d'équations, l'existence d'une normalisation implique qu'il est possible d'écrire les équations de sorte que chaque équation explicite une variable endogène différente.

Soit la matrice d'incidence d'une matrice creuse pour laquelle on donne également

[1] La fonction `spones` produit la matrice d'incidence et la fonction `spy` produit le graphique de la matrice d'incidence.

le graphe biparti associé.

$$M = \begin{array}{c|ccccc} & y_1 & y_2 & y_3 & y_4 & y_5 \\ \hline h_1 & 1 & 1 & . & 1 & 1 \\ h_2 & 1 & 1 & 1 & 1 & . \\ h_3 & . & . & 1 & . & 1 \\ h_4 & 1 & . & . & 1 & . \\ h_5 & 1 & . & 1 & . & . \end{array}$$



On vérifie facilement que les éléments de l'ensemble $\{m_{51}, m_{22}, m_{33}, m_{44}, m_{15}\}$ définissent une normalisation et que de ce fait, la matrice M ci-dessus vérifie la condition nécessaire pour la non-singularité.

Dans le graphe biparti associé à la matrice une normalisation correspond à un ensemble d'arêtes non adjacentes qui relient tous les sommets de l'ensemble H avec tous les sommets de l'ensemble Y. Un ensemble d'arêtes est appelé un *couplage* et on le note W. Si le couplage couvre tous les sommets du graphe on parle de de *couplage parfait*.

La fonction `dmperm`[2] de Matlab permet la recherche d'un couplage W de cardinalité maximale. La commande

```
p = dmperm(M)
```

produit un vecteur dont les éléments sont définis comme

$$p_i = \begin{cases} j & \text{si } m_{ji} \in W \\ 0 & \text{sinon} \end{cases} .$$

Si $p_i \neq 0, \forall i, i = 1, \ldots, n$ la permutation `M(p, :)` produit une matrice avec diagonale non nulle.

On appelle *rang structurel* la cardinalité du couplage maximal, ce qui correspond au nombre d'éléments non nuls du vecteur p. La fonction Matlab `sprank` donne ce nombre[3].

Matrices structurellement singulières

Le rang structurel d'une matrice M vérifie toujours

$$\texttt{sprank(M)} \geq \texttt{rank(M)}.$$

Soit une matrice carrée d'ordre n de rang structurel k avec $k < n$. Dans une telle situation on peut vouloir savoir ou introduire des éléments nouveaux dans la matrice afin d'augmenter son rang structurel. Afin d'illustrer le problème considérons

[2] Cette fonction réalise l'algorithme de Dulmage and Mendelsohn (1963).

[3] La fonction `sprank` calcule ce nombre comme : `r = sum(dmperm(M) > 0)`.

un système de 8 équations et la matrice d'incidence de la matrice jacobienne $\frac{\partial h}{\partial y'}$ associée :

$$
\begin{aligned}
&h_1 : f_1(y_1, y_2, y_3, y_4, y_5) = 0\\
&h_2 : y_6 = f_2(y_3)\\
&h_3 : y_3 = f_3(y_7)\\
&h_4 : f_4(y_1, y_2, y_4, y_6, y_7, y_8) = 0\\
&h_5 : f_5(y_5, y_6) = 0\\
&h_6 : y_6 = f_6(y_7)\\
&h_7 : y_7 = f_7(y_3)\\
&h_8 : f_8(y_3, y_5) = 0
\end{aligned}
\qquad M =
$$

	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
h_1	1	1	1	1	1	.	.	.
h_2	.	.	1	.	.	1	.	.
h_3	.	.	1	.	.	.	1	.
h_4	1	1	.	1	.	1	1	1
h_5	.	.	.	.	1	1	.	.
h_6	.	.	.	.	.	1	1	.
h_7	.	.	1	.	.	.	1	.
h_8	.	.	1	.	1	.	.	.

Avec la fonction `dmperm` on vérifie que le rang structurel de la matrice jacobienne est 6. En effet en exécutant `p = dmperm(M)` on obtient

$$p = \begin{bmatrix} 1 & 4 & 2 & 0 & 5 & 6 & 3 & 0 \end{bmatrix}.$$

On peut montrer (Gilli, 1995; Gilli and Garbely, 1996) que les lignes et les colonnes d'une telle matrice peuvent être réordonnées de sorte à faire apparaître une sous-matrice nulle de dimension $v \times w$ avec $v + w = 2\,n - k$. Pour augmenter le rang structurel de la matrice, de nouveaux éléments non nuls devront alors être introduits exclusivement dans cette matrice nulle mise en évidence.

On peut obtenir les vecteurs de permutation p et q qui mettent en évidence la matrice nulle avec la fonction `dmperm` de la façon suivante :

```
[p,q,r,s] = dmperm(M)
```

Pour notre exemple on obtient les résultats :

$$
\begin{aligned}
p &= \begin{bmatrix} 4 & 1 & 7 & 8 & 5 & 6 & 3 & 2 \end{bmatrix} \qquad q = \begin{bmatrix} 8 & 4 & 2 & 1 & 5 & 6 & 7 & 3 \end{bmatrix}\\
r &= \begin{bmatrix} 1 & 3 & 9 \end{bmatrix} \qquad s = \begin{bmatrix} 1 & 5 & 9 \end{bmatrix}.
\end{aligned}
$$

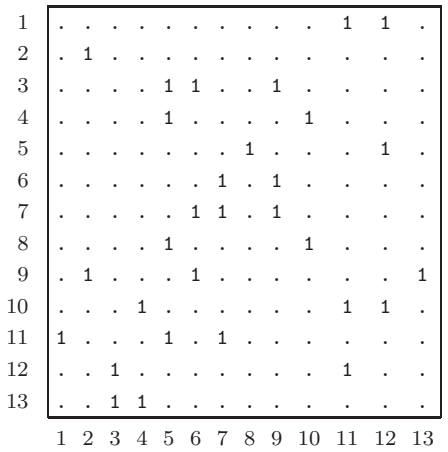
Les vecteurs r et s définissent les lignes et les colonnes de la décomposition. La matrice jacobienne réordonnée est

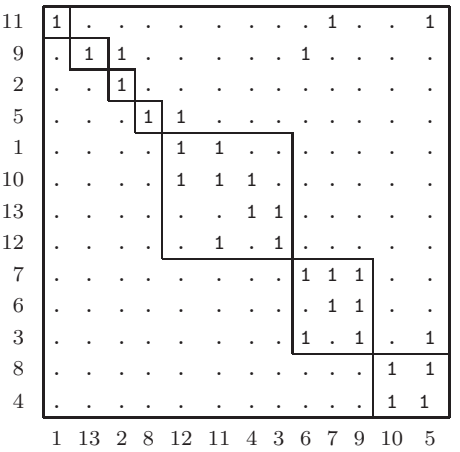
	y_8	y_4	y_2	y_1	y_5	y_6	y_7	y_3
h_4	1	1	1	1	.	1	1	.
h_1	.	1	1	1	1	.	.	1
h_7	.	.	.	.	.	.	1	1
h_8	.	.	.	.	1	.	.	1
h_5	.	.	.	.	1	1	.	.
h_6	.	.	.	.	.	1	1	.
h_3	.	.	.	.	.	.	1	1
h_2	.	.	.	.	.	1	.	1

Décomposition triangulaire par blocs d'une matrice

Les lignes et les colonnes d'une matrice carrée et creuse qui admet un couplage parfait peuvent être permutées de sorte à faire apparaître une structure triangulaire par blocs avec des matrices diagonales qui sont carrées et indécomposables. Les situations extrêmes d'une telle décomposition sont une matrice triangulaire, ce qui correspond à un système d'équations récursives et une matrice indécomposable, ce qui correspond à un système d'équations interdépendant. Dans les situations intermédiaires on peut mettre en évidence une succession de sous-systèmes interdépendants.

Ci-après une matrice décomposable M, les commandes Matlab nécessaires pour la mise en évidence de la décomposition, ainsi que la matrice réordonnée.





La commande Matlab

```
[p,q,r,s] = dmperm(M)
```

produit les résultats suivants

$$p = \begin{bmatrix} 11\ 9\ 2\ 5\ 1\ 10\ 13\ 12\ 7\ 6\ 3\ 8\ 4 \end{bmatrix}$$
$$q = \begin{bmatrix} 1\ 13\ 2\ 8\ 12\ 11\ 4\ 3\ 6\ 7\ 9\ 10\ 5 \end{bmatrix}$$
$$r = \begin{bmatrix} 1\ 2\ 3\ 4\ 5\ 9\ 12\ 14 \end{bmatrix} \qquad s = \begin{bmatrix} 1\ 2\ 3\ 4\ 5\ 9\ 12\ 14 \end{bmatrix}$$

et la permutation de la matrice M à droite s'obtient avec la commande `M(p,q)`.

L'intérêt d'une telle décomposition apparaît lors de la résolution d'un système d'équation étant donné que l'on pourra résoudre de manière recursive les sous-systèmes interdépendants correspondant aux blocs diagonaux.

8.3.1 Exemples de matrices creuses

Les exemples qui suivent illustrent les gains d'efficacité considérables obtenus en considérant la structure creuse de matrices.

```
close all, clear
n = 400; d = .0043;
randn('seed',270785650);

figure(1)
A = sprandn(n,n,d) + speye(n);   subplot(221),spy(A)
B = sprandn(n,100,0.5*d);        subplot(222),spy(B)
```

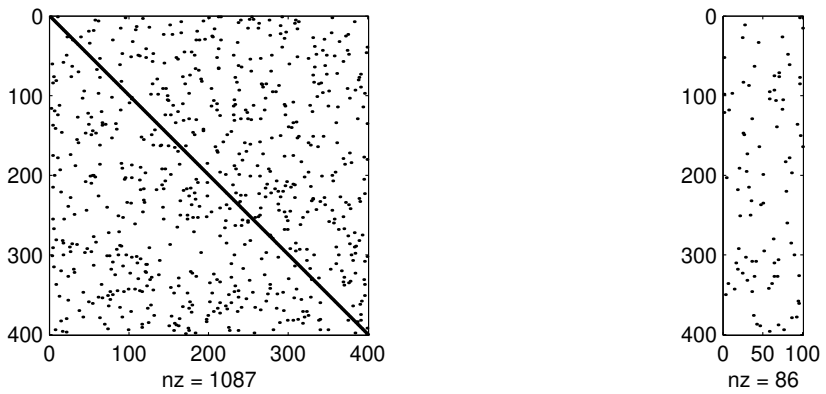


FIG. 8.1 – Système linéaire creux.

```
AF = full(A);
BF = full(B);
whos

  Name        Size              Bytes  Class
  A         400x400             14660  sparse array
  AF        400x400           1280000  double array
  B         400x100              1436  sparse array
  BF        400x100            320000  double array
  d           1x1                   8  double array
  n           1x1                   8  double array

Grand total is 201176 elements using 1616112 bytes
```

L'exemple suivant illustre le produit de matrices creuses et la résolution de systèmes linéaires creux.

```
flops(0), tic, Y = AF*BF;
fprintf('\n* Full: %5.2f sec %7i flops',toc,flops);

flops(0), tic, X = A*B;
fprintf('\n* Sparse: %5.2f sec %7i flops\n',toc,flops);

flops(0), tic, Y = AF\BF;
```

```
fprintf('\n\n␣\\␣␣␣Full:␣%5.2f␣sec␣␣␣␣␣%7i␣flops',toc,flops);

flops(0), tic, X = A\B;
fprintf('\n␣\\␣Sparse:␣%5.2f␣sec␣␣␣␣␣%7i␣flops\n',toc,flops);

 *   Full:  0.38 sec    32000018 flops
 * Sparse:  0.00 sec         496 flops

 \   Full:  0.83 sec     9100260 flops
 \ Sparse:  0.11 sec      231664 flops
```

Décomposition d'une système d'équations linéaires en systèmes indécomposables.

```
subplot(223), mgspypar(A)
```

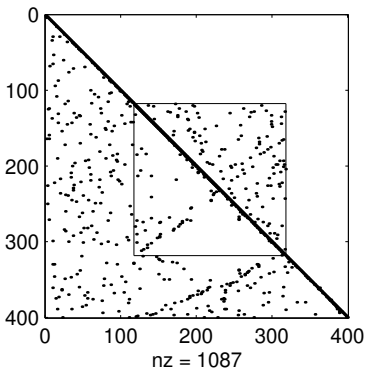


FIG. 8.2 – Système linéaire décomposé par blocs.

```
function mgspypar(A)
% Display the blocktriangular pattern of a matrix
n = size(A,1);
[p,q,r,s] = dmperm(A);  % A(p,q) block-order bottom up
spy(A(p(n:-1:1),q(n:-1:1))); hold on
ncfc = length(r) - 1;
if ncfc > 1
     for k = ncfc:-1:1
         nf = r(k+1)-1;
         nd = r(k);
         if nf > nd
             i = n - nd + 1.5;
             j = n - nf + 0.5;
             plot([i j],[j j],'b-','LineWidth',.05);
             plot([i j],[i i],'b-');
             plot([i i],[j i],'b-');
             plot([j j],[j i],'b-');
         end
```

```
        end
    end
```

Chapitre 9

Méthodes itératives stationnaires

Les méthodes de résolution directes procèdent d'abord à une transformation du système original, puis à la résolution du système transformé. La transformation consiste à factoriser la matrice, puis la résolution s'applique à des systèmes triangulaires. De ce fait, une méthode directe conduit, abstraction faite des erreurs d'arrondi, à la solution exacte, après un nombre d'opérations élémentaires connu et fini.

Les méthodes itératives génèrent une séquence d'approximations $\{x^k\}$, $k = 0, 1, \ldots$ de la solution et de ce fait fournissent une réponse approximative en un nombre infini d'opérations. Cependant dans la pratique l'approximation est souvent satisfaisante au bout d'un nombre d'itérations relativement petit.

Étant donné que la complexité des méthodes directes est $O(n^3)$, la taille des problèmes que l'on peut résoudre avec la technologie actuelle est de l'ordre de quelques milliers. Souvent les grands systèmes d'equations sont creux, c'est-à-dire le nombre d'éléments non nuls ne constitue qu'une très petite fraction de n^2. Dans une telle situation une méthode directe effectue un grand nombre d'opérations redondantes du fait qu'elles impliquent des coefficients dont la valeur est nulle. En adaptant les algorithmes à la structure creuse d'un problème on peut réduire la complexité en évitant notamment ces opérations redondantes. Ainsi il existe des méthodes directes dites *creuses* qui adaptant les méthodes de factorisation à la structure creuse de la matrice A. La mise en oeuvre de telles méthodes n'est cependant pas triviale.

Les méthodes itératives procèdent simplement à des produits matrice-vecteur et il devient alors facile de programmer ce produit de sorte à éviter des opérations redondantes impliquant des éléments nuls de la matrice. Les méthodes itératives peuvent dans certaines circonstances constituer une alternative aux méthodes directes creuses.

Le fait qu'elles soient aussi simples à mettre en oeuvre, tout en exploitant la structure creuse du problème, a beaucoup contribué à leur popularité (surtout parmi les économistes). Leur efficacité depend cependant de la vitesse avec laquelle la séquence

$\{x^k\}$ converge vers la solution.

9.1 Méthodes de Jacobi et de Gauss-Seidel

Ces méthodes remontent à Gauss,[1] Liouville (1837) et Jacobi (1845).

Soit une matrice A dont tous les éléments diagonaux sont non-nuls. Pour une matrice creuse on doit pouvoir mettre en évidence une diagonale non nulle par des permutation des lignes et des colonnes[2]. On appelle *normalisation des équations* la mise en évidence d'une telle diagonale et son existence constitue une condition nécessaire pour la non-singularité de la matrice. Considérons alors le système linéaire $Ax = b$ d'ordre 3 ci-après :

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

Explicitons l'élément diagonal de chaque ligne et réécrivons le système de la façon suivante :

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned}$$

Les x_i dans le membre droit des équations ne sont pas connus, mais supposons que $x^{(k)}$ est une approximation pour $x = A^{-1}b$, alors on peut imaginer d'obtenir une nouvelle approximation en calculant

$$\begin{aligned} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{aligned}$$

Ceci définit *l'itération de Jacobi* qui est formalisée avec l'algorithme 14.

L'itération de Jacobi est particulière dans le sens qu'on n'utilise pas les résultats les plus récents et on procède à n résolutions unidimensionnelles indépendantes. Ainsi lorsqu'on calcule, par exemple, $x_2^{(k+1)}$, on utilise $x_1^{(k)}$ et non $x_1^{(k+1)}$ qui est déjà connu.

Si l'on modifie l'itération de Jacobi de sorte à tenir compte des résultats les plus récents on obtient *l'itération de Gauss-Seidel* :

for $j = 1 : n$ **do**

[1] En 1823 Gauss écrivait : "Fast jeden Abend mache ich eine neue Auflage des Tableau, wo immer leicht nachzuhelfen ist. Bei der Einförmigkeit des Messungsgeschäftes gibt dies immer eine angenehme Unterhaltung ; man sieht daran auch immer gleich, ob etwas Zweifelhaftes eingeschlichen ist, was noch wünschenswert bleibt usw. Ich empfehle Ihnen diesen Modus zur Nachahmung. Schwerlich werden Sie je wieder *direct elimieren*, wenigstens nicht, wenn Sie mehr als zwei Unbekannte haben. Das indirecte Verfahren läßt sich halb im Schlafe ausführen oder man kann während desselben an andere Dinge denken."

[2] Ceci a été formalisé avec la relation (8.2).

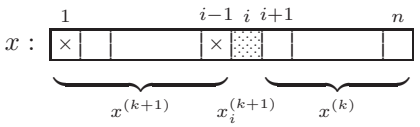
Algorithme 14 Algorithme de Jacobi

1: Donner un point de départ $x^{(0)} \in \mathbb{R}^n$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: **for** $i = 1 : n$ **do**
4: $x_i^{(k+1)} = \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right) / a_{ii}$
5: **end for**
6: **end for**

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

end for

Remarquons qu'il est dans ce cas possible de remplacer au fur et à mesure l'élément $x_i^{(k)}$ par l'élément $x_i^{(k+1)}$ dans le vecteur x.



Les éléments 1 à $i-1$ du vecteur x contiennent déjà les valeurs de l'itération $k+1$ et les éléments $i+1$ à n encore les valeurs de l'itération précédente. On obtient alors l'algorithme 15.

Algorithme 15 Algorithme de Gauss-Seidel

1: Donner un point de départ $x^{(0)} \in \mathbb{R}^n$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: **for** $i = 1 : n$ **do**
4: $x_i = \left(b_i - \sum_{j \neq i} a_{ij} x_j\right) / a_{ii}$
5: **end for**
6: **end for**

9.2 Interprétation géométrique

La présentation géométrique du fonctionnement des algorithmes de Jacobi et Gauss-Seidel permet de montrer pourquoi la normalisation et l'ordre des équations (pour Gauss-Seidel seulement) influencent la convergence de ces méthodes.

Pour cette illustration nous résolvons le système donné par les deux équations suivantes :

$$\begin{aligned} \text{eq1}: \quad & 2x_1 - 1x_2 = 0 \\ \text{eq2}: \quad & -5x_1 + 9x_2 = 0 \end{aligned}$$

Ce système d'équations admet deux normalisations, soit l'équation 1 explique x_1 et l'équation 2 explique x_2, soit l'équation 1 explique x_2 et l'équation 2 explique x_1. Pour la résolution on peut considérer les équations soit dans l'ordre équation 1 puis

équation 2, soit dans l'ordre inverse. Ci-après les quatre systèmes équivalents qui en résultent :

$$\mathrm{N_1O_1}: \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{5}{9} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathrm{N_2O_1}: \quad \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ \frac{9}{5} & 0 \end{bmatrix} \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}$$

$$\mathrm{N_1O_2}: \quad \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 & \frac{5}{9} \\ \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} \qquad \mathrm{N_2O_2}: \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & \frac{9}{5} \\ 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Considérons les itérations de Jacobi appliquées aux systèmes d'équations $\mathrm{N_1O_1}$ et $\mathrm{N_2O_1}$ en choisissant $x^{(0)} = [6 \;\; 5]'$ comme point de départ :

$$\begin{array}{lrcl} \mathrm{N_1O_1}: & x_1^{(1)} & = & \frac{1}{2}x_2^{(0)} = \frac{5}{2} \\ & x_2^{(1)} & = & \frac{5}{9}x_1^{(0)} = \frac{10}{3} \\ & x_1^{(2)} & = & \frac{1}{2}x_2^{(1)} = \frac{5}{3} \\ & x_2^{(2)} & = & \frac{5}{9}x_1^{(1)} = \frac{25}{18} \\ & x_1^{(3)} & = & \frac{1}{2}x_2^{(2)} = \frac{25}{36} \\ & x_2^{(3)} & = & \frac{5}{9}x_1^{(2)} = \frac{25}{27} \end{array} \qquad \begin{array}{ll} \mathrm{N_2O_1}: & x_2^{(1)} = 2x_1^{(0)} = 12 \\ & x_1^{(1)} = \frac{9}{5}x_2^{(0)} = 9 \\ & x_2^{(2)} = 2x_1^{(1)} = 18 \\ & x_1^{(2)} = \frac{9}{5}x_2^{(1)} = \frac{108}{5} \end{array}$$

A chaque itération l'algorithme de Jacobi opère n résolutions unidimensionnelles indépendantes. Chaque équation est résolue par rapport à la variable spécifiée par la normalisation, les autres variables restent fixées à la valeur donnée au début de l'itération.

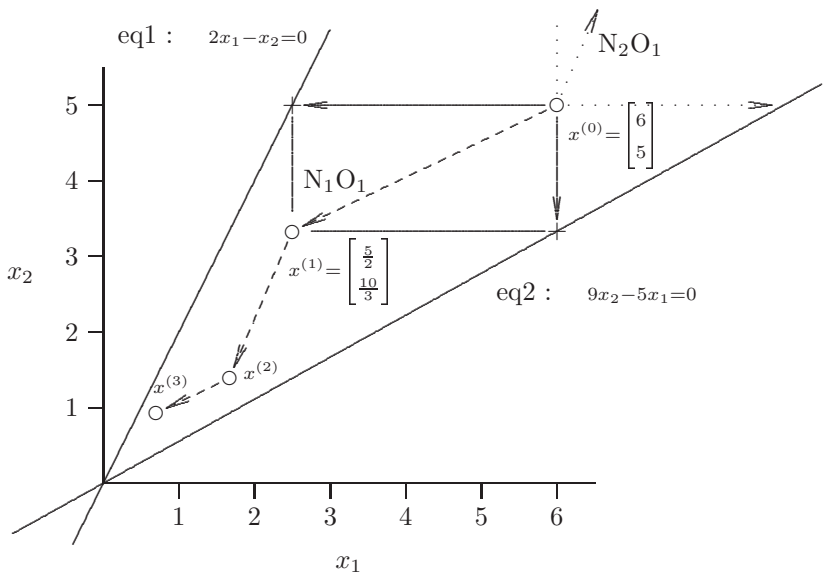


FIG. 9.1 – Algorithme de Jacobi.

La figure 9.1 montre comment l'algorithme converge pour la normalisation 1 et diverge pour la normalisation 2. Étant donné que les composantes de la solution

sont calculées indépendamment il apparaît clairement que l'ordre des équations est sans incidence sur le fonctionnement de l'algorithme de Jacobi.

Lors de l'itération de Gauss-Seidel la résolution unidimensionnelle ne se fait pas de manière indépendante mais on tient successivement compte de la mise à jour des composantes du vecteur de solution. L'algorithme de Gauss-Seidel est donc sensible à la normalisation et à l'ordre des équations comme le montre son application aux quatre systèmes spécifiés plus haut :

$$
\begin{array}{lrcl@{\qquad}lrcl}
\mathrm{N_1O_1}: & x_1^{(1)} & = & \frac{1}{2}x_2^{(0)} = 2.5 & \mathrm{N_1O_2}: & x_2^{(1)} & = & \frac{5}{9}x_1^{(0)} = 3.33 \\
& x_2^{(1)} & = & \frac{5}{9}x_1^{(1)} = 1.38 & & x_1^{(1)} & = & \frac{1}{2}x_2^{(1)} = 1.66 \\
& x_1^{(2)} & = & \frac{1}{2}x_2^{(1)} = 0.69 & & x_2^{(2)} & = & \frac{5}{9}x_1^{(1)} = 0.92 \\
& x_2^{(2)} & = & \frac{5}{9}x_1^{(2)} = 0.38 & & x_1^{(2)} & = & \frac{1}{2}x_2^{(2)} = 0.40 \\
\mathrm{N_2O_1}: & x_2^{(1)} & = & 2x_1^{(0)} = 12 & \mathrm{N_2O_2}: & x_1^{(1)} & = & \frac{9}{5}x_2^{(0)} = 9 \\
& x_1^{(1)} & = & \frac{9}{5}x_2^{(1)} = 21.6 & & x_2^{(1)} & = & 2x_1^{(1)} = 18
\end{array}
$$

Dans la figure 9.2 on peut suivre le cheminement de la solution en fonction de la normalisation et de l'ordre des équations.

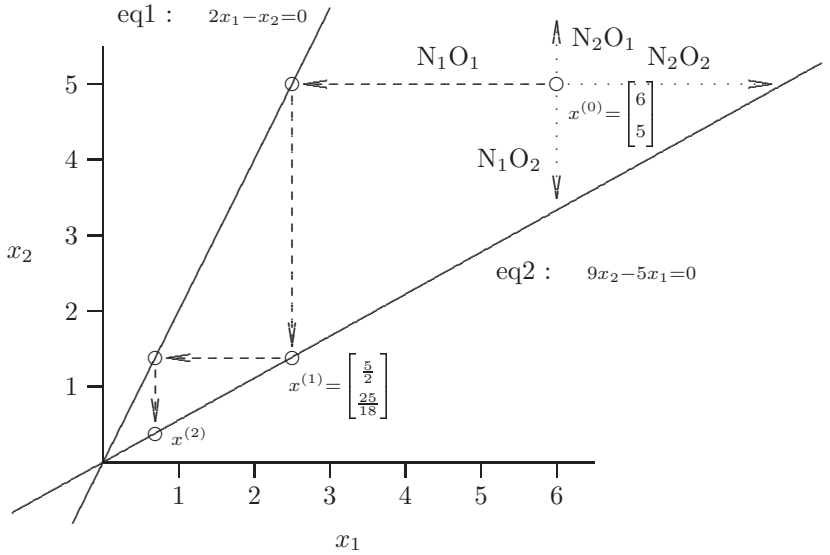


FIG. 9.2 – Algorithme de Gauss-Seidel.

9.3 Convergence de Jacobi et Gauss-Seidel

Afin de pouvoir étudier la convergence des méthodes itératives il convient de formaliser ces algorithmes de manière différente. Considérons la matrice A et sa décomposition $A = L + D + U$:

$$\begin{bmatrix} 0 & \cdots & \cdots & 0 \\ a_{21} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{bmatrix} + \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix} + \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & a_{n-1,n} \\ 0 & \cdots & \cdots & 0 \end{bmatrix}.$$

L'itération générique pour la méthode de Jacobi s'écrit alors

$$Dx^{(k+1)} = -(L+U)x^{(k)} + b$$

et pour la méthode de Gauss-Seidel on a

$$(D+L)x^{(k+1)} = -Ux^{(k)} + b.$$

Ainsi les méthodes itératives peuvent se généraliser sous la forme

$$Mx^{(k+1)} = Nx^{(k)} + b \tag{9.1}$$

avec $A = M - N$ une décomposition de la matrice A. Pour la méthode de Jacobi on a $M = D$ et $N = -(L+U)$ et pour la méthode de Gauss-Seidel $M = D + L$ et $N = -U$.

Pour qu'il ait un intérêt pratique, le système linéaire (9.1), c'est-à-dire

$$Mx^{(k+1)} = c$$

doit être facile à résoudre. Ceci est le cas dans nos exemples, étant donné que M est diagonale pour la méthode de Jacobi et triangulaire pour la méthode de Gauss-Seidel.

Montrons que la convergence de (9.1) vers $x = A^{-1}b$ dépend des valeurs propres de $M^{-1}N$.

Définition 9.1 Soit une matrice $B \in \mathbb{R}^{n\times n}$, alors son *rayon spectral* ρ est défini comme

$$\rho(B) = \max\{\ |\lambda| \text{ tel que } \lambda \text{ est une valeur propre de } B\}.$$

Théorème 9.1 Soit $b \in \mathbb{R}^n$ et $A = M - N \in \mathbb{R}^{n\times n}$ non-singulière. Si M est non-singulière et si $\rho(M^{-1}N) < 1$, alors les itérations $x^{(k)}$ définies par $Mx^{(k+1)} = Nx^{(k)} + b$ convergent vers $x = A^{-1}b$, quelque soit la valeur initiale $x^{(o)}$.

Démonstration 7 Soit $e^{(k)} = x^{(k)} - x$ l'erreur à l'itération k. Comme $Ax = b$ et $Mx = Nx + b$ et en remplacent x par $x^{(k)} - e^{(k)}$ on déduit que l'erreur pour $x^{(k+1)}$

est donné par $e^{(k+1)} = M^{-1}Ne^{(k)} = (M^{-1}N)^k e^{(o)}$. L'on sait que $(M^{-1}N)^k \to 0$ si et seulement si $\rho(M^{-1}N) < 1$.

$$\begin{aligned} Ax &= b \\ Mx &= Nx + b = N(x^{(k)} - e^{(k)}) + b = \underbrace{Nx^{(k)} + b}_{Mx^{(k+1)}} - Ne^{(k)} \\ M(\underbrace{x^{(k+1)} - x}_{e^{(k+1)}}) &= Ne^{(k)} \\ e^{(k+1)} &= M^{-1}Ne^{(k)} = (M^{-1}N)^k e^{(o)} \end{aligned}$$

Ainsi la convergence de (9.1) dépendra de $\rho(M^{-1}N)$.

Conditions suffisantes pour la convergence

Une condition suffisante mais pas nécessaire qui garantit la convergence pour l'itération de Jacobi est la dominance diagonale. Une matrice $A \in \mathbb{R}^{n\times n}$ est dite à *diagonale dominante stricte* si

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \quad i = 1, \ldots, n.$$

Dans ce cas on vérifie

$$\rho(M^{-1}N) \;\leq \| \; D^{-1}(L+U) \; \|_\infty \; = \max_{1\leq i\leq n} \sum_{\substack{j=1 \\ j\neq i}}^{n} \left| \frac{a_{ij}}{a_{ii}} \right| < 1 \,.$$

Pour la méthode de Gauss-Seidel on peut monter que l'itération converge toujours si A est symétrique et définie positive (condition suffisante mais pas nécessaire).

9.4 Méthodes de relaxation (SOR)

Lorsque le rayon spectral de $M^{-1}N$ est proche de l'unité, la méthode de Gauss-Seidel converge très lentement. On peut y remédier en modifiant l'iteration de Gauss-Seidel de la façon suivante :

$$x_i^{(k+1)} = \omega \, x_{GS,i}^{(k+1)} + (1-\omega) x_i^{(k)}$$

avec $x_{GS,i}^{(k+1)}$ la valeur de $x_i^{(k+1)}$ calculé avec un pas de Gauss-Seidel et ω le paramètre de relaxation. Ceci définit la méthode de la *sur-relaxation successive* (SOR). On voit qu'il s'agit d'une combinaison linéaire d'un pas de Gauss-Seidel et le pas précédent de SOR. On a l'algorithme :

Algorithme 16 Sur-relaxation successive (SOR)

1: Donner un point de départ $x^{(0)} \in \mathbb{R}^n$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: **for** $i = 1 : n$ **do**
4: $x_i^{(k+1)} = \omega\, x_{GS,i}^{(k+1)} + (1-\omega) x_i^{(k)}$
5: **end for**
6: **end for**

Choix du paramètre de relaxation ω

Afin d'étudier le choix des valeurs de ω formalisons l'itération matriciellement. On peut écrire

$$M_\omega\, x^{(k+1)} = N_\omega\, x^{(k)} + \omega\, b$$

avec $M_\omega = D + \omega L$ et $N_\omega = (1-\omega)D - \omega U$. La valeur du paramètre de relaxation ω qui minimise le rayon spectral n'est connue que pour quelques problèmes particuliers. En général on définit ω par tâtonnement ou par balayage.

Montrons encore que le paramètre de relaxation ω doit vérifier $0 < \omega < 2$. En effet on a

$$\begin{aligned}
\det(M_\omega^{-1} N_\omega) &= \det(D + \omega L)^{-1} \times \det\Big((1-\omega)D - \omega U\Big) \\
&= \det\Big((1-\omega)D\Big) / \det(D) \\
&= (1-\omega)^n < 1
\end{aligned}$$

et comme d'autre part $\det(M_\omega^{-1} N_\omega) = \Pi_{i=1}^n \lambda_i$ on voit que la condition $\rho(M_\omega^{-1} N_\omega) < 1$ implique $0 < \omega < 2$.

Rappelons que si $\rho > 1$ évidemment aucune des méthodes ne converge.

9.5 Structure des algorithmes itératifs

Les méthodes itératives nécessitent un nombre infini d'itérations pour converger vers la solution. Ainsi, pour rendre ces méthodes opérationnelles, il convient d'introduire un critère d'arrêt pour le procédé itératif. On arrêtera les itérations lorsque les changements dans le vecteur de solution, c'est-à-dire l'erreur devient suffisamment petite.

Comme il a été suggéré à la section 1.3 on choisira une combinaison entre erreur absolue et erreur relative ce qui conduira à stopper les itérations lorsque la condition suivante est satisfaite

$$\frac{|x_i^{(k+1)} - x_i^{(k)}|}{|x_i^{(k)}| + 1} < \varepsilon \qquad i = 1, 2, \ldots, n$$

avec ε une tolérance donnée.

Structure des algorithmes itératifs

La mise en oeuvre d'algorithmes itératifs peut alors se schématiser avec les instructions suivantes :

Initialiser $x^{(1)}$, $x^{(0)}$, ε et le nombre maximum d'itérations
while ~`converged`$(x^{(0)}, x^{(1)}, \varepsilon)$ **do**
  $x^{(0)} = x^{(1)}$ (Conserver l'itération précédente dans $x^{(0)}$)
  Évaluer $x^{(1)}$ avec Jacobi, Gauss-Seidel ou SOR
  Test sur le nombre d'itérations
end while

La fonction Matlab `converged` se présente comme :

```
function  res = converged(y0,y1,tol)
res = all(abs(y1-y0)./(abs(y0)+1) < tol);
```

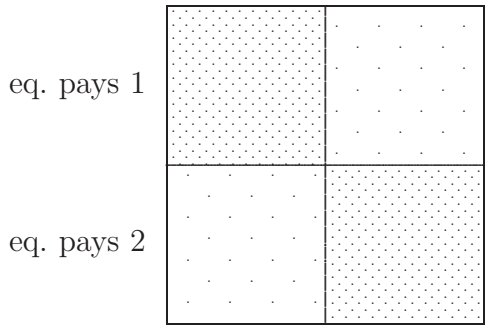
L'étape qui évalue x nécessite $n\,2\,p$ opérations élémentaires pour Jacobi et Gauss-Seidel et $n\,(2\,p+3)$ opérations élémentaires pour SOR, ou p est le nombre moyen d'éléments non nuls dans une ligne de la matrice A d'ordre n. Dès lors on peut envisager de préférer la méthode de Jacobi ou Gauss-Seidel à une méthode directe (qui n'exploite pas la structure creuse de A) si

$$2\,k\,n\,p < \tfrac{2}{3}\,n^3 \quad \Rightarrow \quad k < \frac{n^2}{3\,p}$$

ou k est le nombre d'itérations.

9.6 Méthodes itératives par bloc

Certains problèmes peuvent être naturellement décomposés en sous-problèmes plus au moins fortement liés entre eux. Ceci est notamment le cas pour des modèles décrivant l'interdépendance de l'économie de plusieurs pays (multi-country models). Les économies nationales étant liées entre elles par les relations de commerce extérieur seulement. Une autre situation constituent des modèles multi-sectoriels désagrégés ou l'on observe que peu de liens entre les secteurs comparé aux liens existant à l'intérieur d'un secteur.



Une méthode par bloc est alors une technique ou l'on itère sur les blocs. La technique utilisée pour résoudre le bloc peut être quelconque et n'est pas importante pour la définition de la méthode.

Les méthodes itératives par bloc peuvent être appliquées indifféremment aux systèmes linéaires et aux systèmes d'équations non-linéaires. Pour simplifier la présentation on considère dans la suite la solution d'un système linéaire $Ay = b$. Dans le cas non-linéaire la matrice A constitue la matrice Jacobienne et du fait de la non-linéarité la valeur des éléments de la matrice A et du vecteur b changent d'itération en itération.

Supposons la partition suivante de notre matrice A

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{bmatrix}$$

avec A_{ii}, $i = 1, \ldots, N$ des matrices carrées. En partitionnant le système $Ay = b$ de façon correspondante on obtient

$$\begin{bmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & \vdots & \vdots \\ A_{N1} & \cdots & A_{NN} \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$$

que l'on peut aussi écrire

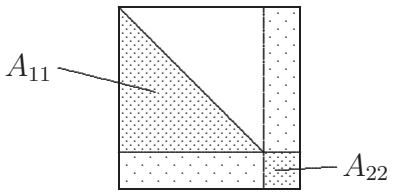
$$\sum_{j=1}^{N} A_{ij}\, y_j = b_i \qquad i = 1, \ldots, N.$$

Si les matrices A_{ii}, $i = 1, \ldots, N$ sont non-singulières alors on peut mettre en oeuvre l'algorithme suivant :

Il existe une pléthore de variantes pour le choix de la partition et la façon dont le bloc est résolu. Une variante consiste à choisir $N = 2$ avec A_{11} triangulaire inférieure, donc facile à résoudre, et A_{22} quelconque mais de taille nettement inférieure que l'on résoudra avec une méthode directe dans le cas d'un système linéaire et la méthode de Newton pour un système non-linéaire.

Algorithme 17 Méthode itérative par bloc

1: Donner un point de départ $y^{(0)} \in \mathbb{R}^n$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: **for** $i = 1 : N$ **do**
4: solve $A_{ii}\, y_i^{(k+1)} = b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} A_{ij}\, y_j^{(k)}$
5: **end for**
6: **end for**



Une autre alternative consiste à résoudre les blocs de manière approximative (incomplete inner loops).

La convergence des méthodes itératives par bloc dépend comme pour les méthodes itératives stationnaires des valeurs propres de $M^{-1}N$.

Chapitre 10

Méthodes itératives non stationnaires

Chapitre 11

Equations non-linéaires

Contrairement à ce qui a été le cas pour les systèmes d'équations linéaires, la résolution des systèmes non-linéaires s'avère beaucoup plus délicate. En général il n'est pas possible de garantir la convergence vers la solution correcte et la complexité des calculs croît très vite en fonction de la dimension du système. Ci après, à titre d'exemple un système de deux équations :

$$y_1 - y_2^3 - \tfrac{1}{2} = 0 \tag{11.1}$$
$$4\,y_1^2 - y_2 + c = 0 \tag{11.2}$$

Dans la Figure 11.1 on montre qu'en fonction de la valeur que prend le paramètre c ce système admet entre zéro et quatre solutions.

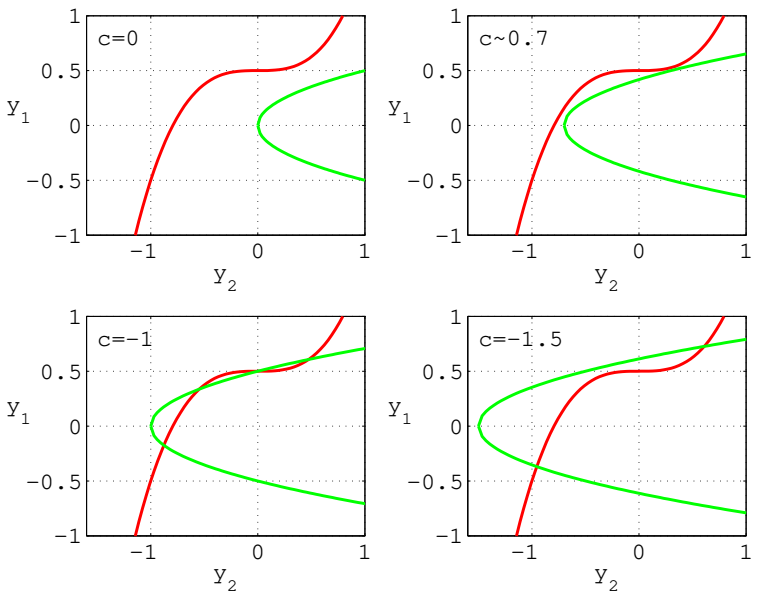


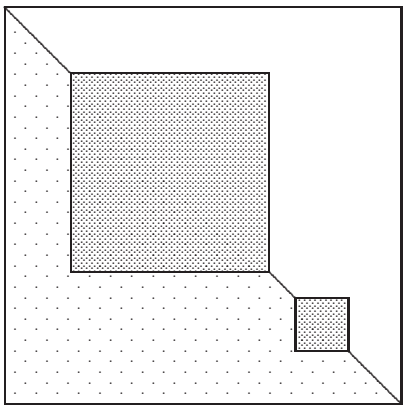
FIG. 11.1 – Solutions du systèmes d'équations (11.1–11.2) en fonction de c.

On considère un système d'équations

$$h(y, z) = 0 \qquad \equiv \qquad F(y) = 0$$

où $h : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ est dérivable dans un voisinage de la solution $y^* \in \mathbb{R}^n$ et $z \in \mathbb{R}^m$ pour les variables exogènes. Si au moins une des équations est non-linéaire, le système sera dit non-linéaire.

Dans la pratique, et surtout dans le domaine de la modélisation macroéconométrique, la matrice Jacobienne $\partial h/\partial y'$ peut souvent être permutée de sorte à faire apparaître une structure récursive par blocs comme indiquée dans la figure ci-après[1] :



Dans ce cas la solution du système d'équations s'obtient en résolvant une succession de systèmes récursifs et de systèmes interdépendants. Dans la suite on supposera toujours que le système à résoudre est interdépendant, c'est-à-dire que sa matrice Jacobienne ne peut se mettre sous forme triangulaire par blocs.

Les trois approches suivantes constituent les techniques les plus souvent utilisés pour la résolution de systèmes d'équations non-linéaires :

– Méthode du point fixe
– Méthode de Newton
– Résolution par minimisation

11.1 Equation à une variable

Toute valeur de x qui satisfait $f(x) = 0$ est appelée *zéro de la fonction*. Souvent dans la pratique, soit il n'existe pas de solution analytique à ce problème, soit son obtention est très laborieuse. Remarquons que toute égalité peut être mise sous cette forme.

Le problème de la recherche des zéros d'une fonction est fréquemment rencontré. Mentionnons ici, par exemple, le calcul de la volatilité implicite des options (donner

[1] La recherche de cette décomposition a été illustré dans la section 8.3.1.

un exemple en économie). En réalité, il s'agit de résoudre une équation où, après avoir ramené tous les termes à gauche du signe de l'égalité, elle se présente comme

$$f(x) = 0\,.$$

Ici on va traiter uniquement le cas où f est univariée, $f : \mathbb{R} \to \mathbb{R}$. Le cas multivarié, $f : \mathbb{R}^m \to \mathbb{R}^n$, sera traité dans le cadre de la résolution de systèmes (linéaires et non-linéaires).

Il existe une grande variété de situations, plus ou moins délicates à traiter. Hormis dans les cas linéaires, on procède toujours par itérations (voir méthodes itératives).

Le cas représenté à gauche dans la figure 11.2 est un des plus simples et favorables. La fonction varie doucement, s'annule et change de signe.

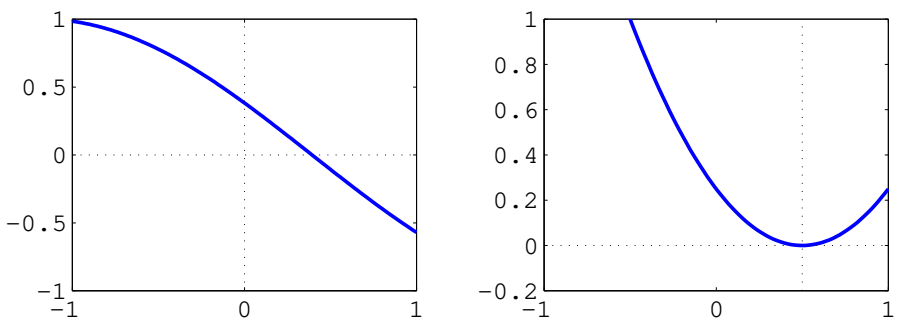


FIG. 11.2 – Gauche : $f(x) = \cos(x + 3\pi/8)$. Droite : $f(x) = (x - 1/2)^2$.

Le cas à droite de la figure 11.2 reste relativement aisé, mais est toutefois moins favorable que le précédent car la fonction varie doucement, s'annule mais ne change pas de signe.

Les cas de la figure 11.3 sont similaires au cas à la droite de la figure 11.2, sauf que les erreurs numériques peuvent conduire à deux situations différentes. Notamment la fonction s'annule presque mais reste "numériquement" strictement positive. Dans ce cas, doit-on considérer que $x = 1/2$ est la solution d'une fonction mal évaluée, ou n'est pas la solution car la fonction ne s'annule effectivement pas ?

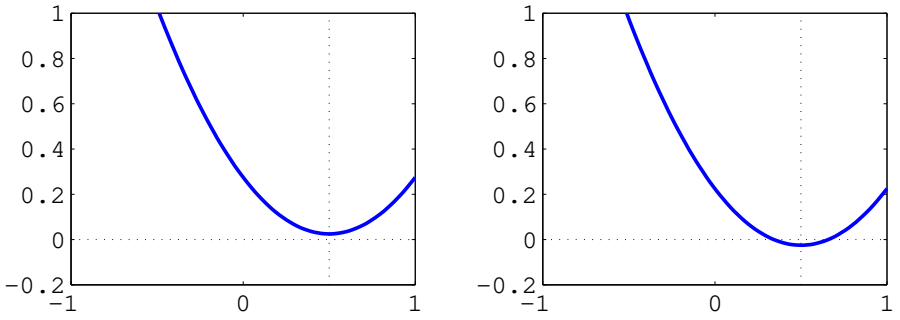


FIG. 11.3 – Gauche : $f(x) = (x - 1/2)^2 + \epsilon$. Droite : $f(x) = (x - 1/2)^2 - \epsilon$.

Dans l'autre cas la fonction s'annule deux fois car elle prend des valeurs très faiblement négatives : dans ce cas, peut-on considérer que les deux solutions distinctes sont acceptables ou sont seulement les conséquences d'une mauvaise évaluation de la fonction ?

Dans la figure 11.4 à nouveau deux situations délicates. Dans le premier cas, la fonction présente plusieurs zéros ; dans ce cas, comment peut-on obtenir toutes les solutions ? Un algorithme donné converge vers une solution particulière, mais laquelle ? Comment passer d'une solution à la suivante ?

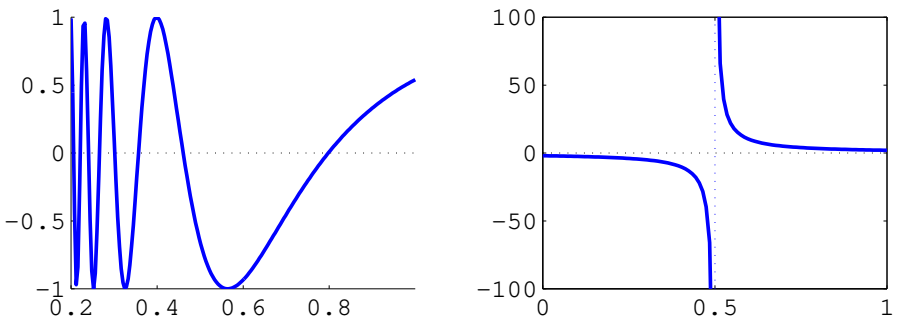


FIG. 11.4 – Gauche : $f(x) = \cos(1/x^2)$. Droite : $f(x) = 1/(x - 1/2)$.

Le second cas correspond à une fonction présentant une singularité et changeant de signe de part et d'autre de la singularité ; dans un tel cas, l'algorithme risque de "croire" que le point de singularité est la solution cherchée ; d'autre part, l'évaluation de la fonction en ce point risque de poser problème (division par zéro, grande erreur de précision, ...)

Les fonctions de la figure 11.5 présentent une discontinuité. La fonction H étant définie comme

$$H(x) = \begin{cases} -1 & x < 0 \\ 1 & x > 0 \end{cases}$$

Une telle discontinuité ne peut pas être prise en compte numériquement, elle nécessiterait une précision infinie ; dans l'exemple illustré ici, la valeur $x = 0$ serait considérée comme solution de l'équation $f(x) = 0$.

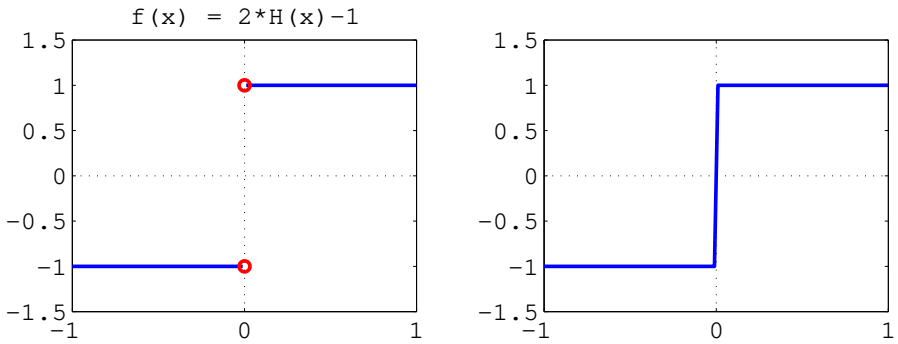


FIG. 11.5 – Gauche : $f(x) = 2H(x) - 1$. Droite : $f(x) = 1/(x - 1/2)$.

Il est en général assez délicat de donner une réponse universelle à ces questions ; seule une étude particulière adaptée à la situation permet de résoudre ce problème.

Ces quelques exemples montrent qu'il est difficile de définir un algorithme numérique universel, et qu'il est indispensable de connaître a priori certaines informations relatives à la fonction et à son/ses zéro(s). C'est alors en fonction de cette connaissance a priori que l'on cherchera la méthode numérique la plus adaptée.

Critères pour le choix de la méthode

Avant de choisir une méthode numérique pour la recherche des zéros d'une fonction, où d'explorer l'existence d'une solution analytique il convient de se poser les questions suivantes :

- Doit-on procéder à des recherches répétées ? Si oui, peut-être convient-il d'explorer l'existence d'une solution analytique.
- Quelle est la précision désirée ? Si quelques digits suffisent, alors une méthode quelconque peut-être utilisée.
- La méthode doit-elle être rapide et robuste (insensible aux choix des points de départ) ?
- S'agit-il d'un polynôme ? Dans ce cas il existe des procédures spéciales.
- La fonction comporte-elle des singularités ?

Principe de la recherche des zéros

Considérons la figure **??** qui illustre une fonction avec deux zéros, x_1 et x_2 dans le domaine des x considéré. Toute recherche de zéro démarre avec une solution initiale

qui sera améliorée dans des itérations successives, et dans des conditions normales, s'approche de la solution. Quelques remarques :

– Le choix du point de départ determine vers quelle solution on se dirigera.
– Il faut définir un critère d'arrêt pour les itérations.
– Il convient de substituer la solution dans la fonction et de vérifier que l'on a $f(x_{\text{sol}}) \approx 0$.
– La figure **??** illustre qu'une fonction qui, pour un intervalle donné, vérifie des signes opposés peut soit contenir un zéro, soit une singularité.

11.1.1 Recoupement par intervalles (bracketing)

Il s'agit de construire des intervalles susceptibles de contenir un zéro. Ultérieurement on raffinera la recherche du zéro à l'intérieur de l'intervalle.

On subdivise un domaine donné en intervalles réguliers et on examine si la fonction traverse l'axe des x en analysant le signe de la fonctions évaluée aux bords de l'intervalle (voir figure **??**).

Algorithme 18 Étant donné $f(x)$, $x_{\min}$, $x_{\max}$ et n on identifie les intervalles susceptibles de contenir des zéros.

$\Delta = (x_{\max} - x_{\min})/n$
$a = x_{\min}$
$i = 0$
while $i < n$ **do**
  $i = i + 1$
  $b = a + \Delta$
  if $\operatorname{sign} f(a) \neq \operatorname{sign} f(b)$ **then**
    $[a, b]$ peut contenir un zéro, imprimer a et b
  end if
  $a = b$
end while

La fonction `bracketing` réalise l'algorithme 18 et la figure 11.6 illustre la recherche de ces intervalles pour la fonction $f(x) = \cos(1/x^2)$ dans le domaine $x \in [0.3, 0.9]$ et $n = 25$.

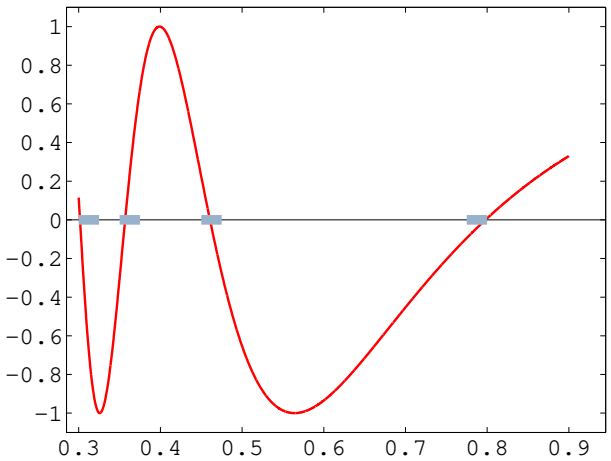


FIG. 11.6 – $f(x) = \cos(1/x^2)$, $n = 25$.

Code `/Teaching/MNE/Ex/bracketing.m`:

```
function ab = bracketing(f,xmin,xmax,n)
k = 0;
x = linspace(xmin,xmax,n);
a  = xmin;
fa = feval(f,a);
for i = 2:n
   b  = x(i);
   fb = feval(f,b);
   if sign(fa)~=sign(fb)
      k = k + 1;
      ab(k,:) = [a b];
   end
   a  = b;
   fa = fb;
end
```

Pour tester si les fonctions sont de signe opposé en a et b on pourrait écrire

$$f(a) \times f(b) < 0\,.$$

D'un point de vue numérique il est cependant préférable d'avoir recours à la fonction `sign`. L'exemple qui suit montre comment pour des valeurs très petites de f le résultat peut correspondre à un *underflow*.

```
>> fa = 1e-140;  fb = 1e-180;
>> fa * fb
ans =
       0
```

11.1.2 Itérations de point fixe

Soit une fonction $f(x) = 0$, on isole un terme contenant x de la sorte à pouvoir écrire

$$x_{\text{new}} = g(x_{\text{old}})\,.$$

On appelle la fonction $g(x)$ la fonction d'itération et une itération du point fixe est définie par l'algorithme qui suit.

Algorithme 19

1: Initialiser $x^{(0)}$
2: **for** $k = 1, 2, \ldots$ jusqu'à convergence **do**
3: $\quad x^{(k)} = g(x^{(k-1)})$
4: **end for**

A propos de l'algorithme du point fixe il convient de faire les remarques suivantes :

– La notion de convergence doit être approfondie.
– Les valeurs successives de x ne doivent pas être conservées dans un vecteur. Il suffit de conserver deux valeurs successives $x^{(k)}$ et $x^{(k+1)}$ dans des variables $x0$ et $x1$ et de remplacer $x0$ par $x1$.

```
while ~converged
   x1 = g(x0)
   x0 = x1
end
```

– Le fait que la solution x_{sol} vérifie $x_{\text{sol}} = g(x_{\text{sol}})$ fait qu'on appelle x_{sol} un *point fixe*.
– Le choix de la fonction $g(x)$ est déterminant pour la convergence de la méthode.

Code `/Teaching/MNE/Ex/FPI.m`:

```
function x1 = FPI(f,x1,tol)
% FPI(f,x0)  Iterations du point fixe
%
if nargin == 2, tol = 1e-8; end
it = 0; itmax = 100; x0 = realmax;
while ~converged1(x0,x1,tol)
   x0 = x1;
   x1 = feval(f,x0);
   it = it + 1;
   if it > itmax
      error('Maxit␣in␣FPI');
   end
end
```

Exemple 1 $f(x) = x - x^{4/5} - 2 = 0$

$$g_1(x) = x^{4/5} + 2 \qquad g_2(x) = (x-2)^{5/4}$$

```
» f = inline('x.^(4/5)+2');
```

```
» FPI(f,8)
ans =
    6.4338

» f = inline('(x-2).^(5/4)');
» FPI(f,8)
??? Error using ==> fpi
Maxit in FPI
```

Exemple 2 $f(x) = x - 2x^{4/5} + 2 = 0$

$$g_1(x) = 2x^{4/5} - 2 \qquad g_2(x) = \left(\frac{x+2}{2}\right)^{5/4}$$

```
» bracketing(f,0,50,30)
ans =
    3.4483    5.1724
   18.9655   20.6897

» f = inline('2*x.^(4/5)-2');
» FPI(f,1)
??? Error using ==> fpi
Maxit in FPI

» FPI(f,9)
??? Error using ==> fpi
Maxit in FPI
» f = inline('((x+2)/2).^(5/4)');
» FPI(f,1)
ans =
    3.7161
» FPI(f,20)
??? Error using ==> fpi
Maxit in FPI
```

Convergence de la méthode de point fixe

Les itérations de point fixe convergent pour $x \in [a, b]$ si l'intervalle contient un zéro et si

$$|g'(x)| < 1 \quad \text{pour} \quad x \in [a, b]\,.$$

Si $-1 < g'(x) < 0$ les itérations oscillent autour de la solution et si $0 < g'(x) < 1$ les itérations convergent de façon monotone. Voir les figures qui suivent.

11.1.3 Bisection

On considère un intervalle qui contient le zéro, on le divise en deux et on cherche le demi-intervalle qui contient le zéro. La procédure est réitérée sur le nouveau

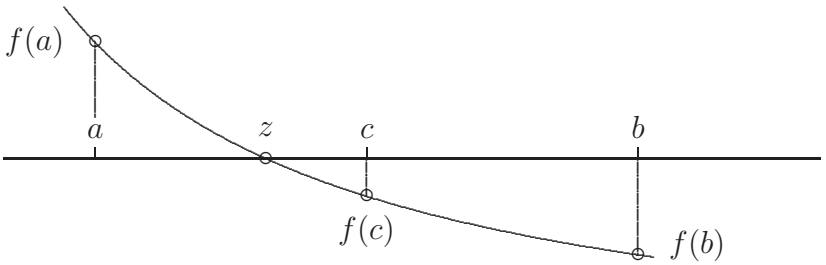
intervalle contenant le zéro. A chaque itération l'intervalle est divisé par deux. Les centres

$$c = \frac{a+b}{2}$$

des intervalles successifs convergent vers le zéro. En évaluant le centre comme

$$c = a + \frac{b-a}{2}$$

on obtient des résultats qui sont numériquement plus stables.



L'algorithme 20 décrit la méthode.

Algorithme 20 Bisection (η est la tolérance d'erreur).

1: Vérifier que $f(a) \times f(b) < 0$
2: **while** $|a - b| > \eta$ **do**
3: $c = a + (b-a)/2$
4: **if** $f(c) \times f(a) < 0$ **then**
5: $b = c$ (z est à gauche de c)
6: **else**
7: $a = c$ (z est à droite de c)
8: **end if**
9: **end while**

Le code qui suit donne la réalisation avec Matlab de l'algorithme de la bisection.

```
function c = bisection(f,a,b,tol)
% Recherche zero d'une fonction avec methode de la bisection
% On cherche zero dans intervalle [a, b] avec tolerance tol
% La fonction doit etre de signe oppose en a et b
%
if nargin == 3, tol = 1e-8; end
fa = feval(f,a);
fb = feval(f,b);
if sign(fa) == sign(fb)
   error('fonction pas de signe oppose en a et b');
end
fait = 0;
while abs(b-a) > 2*tol & ~fait
   c = a + (b - a) / 2;     % Chercher centre intervalle
   fc = feval(f,c);         % Evaluer f au centre
```

```
   if sign(fa) ~= sign(fc) % zero a gauche de c
      b  = c;
      fb = fc;
   elseif sign(fc) ~= sign(fb) % zero a droite de c
      a  = c;
      fa = fc;
   else                % on tombe exactement sur zero
      fait = 1;
   end
end
```

Exemple 3

```
» f = inline('x-2*x.^(4/5)+2');
» bisection(f,0,5,1e-8)
ans =
    3.7161
» bisection(f,5,20,1e-8)
ans =
   19.7603
```

Convergence

Soit $\triangle_0 = b - a$ l'intervalle initial, alors l'intervalle à l'itération n se déduit comme

$$\begin{aligned}
\triangle_1 &= \frac{1}{2}\,\triangle_0 \\
\triangle_2 &= \frac{1}{2}\,\triangle_1 = \frac{1}{4}\,\triangle_0 \\
&\vdots \\
\triangle_n &= \left(\frac{1}{2}\right)^n \triangle_0
\end{aligned}$$

La réduction relative après n itérations est $\triangle_n \;/\; \triangle_0 = 2^{-n}$ d'ou on tire le nombre d'itérations nécessaires pour atteindre une réduction donnée de l'intervalle, soit

$$n = \log_2 \frac{\triangle_n}{\triangle_0}\,.$$

Voir critère de convergence.

11.1.4 Méthode de Newton

Parfois appelée Newton-Raphson.

11.2 Systèmes d'équations

11.2.1 Méthodes itératives

Gauss-Seidel et Jacobi

Les méthodes de Gauss-Seidel et Jacobi vus dans la section 9.1 pour la solution de systèmes linéaires peuvent être étendus pour la résolution de systèmes non-linéaires.

En général on normalise les équations, c'est-à-dire on les réécrit sous la forme

$$y_i = g_i(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n, z) \qquad i = 1, \ldots, n.$$

Rappelons que les équations g_i peuvent maintenant être non-linéaires.

Pour la méthode de Jacobi l'itération générique s'écrit alors :

$$y_i^{(k+1)} = g_i(y_1^{(k)}, \ldots, y_{i-1}^{(k)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}, z) \qquad i = 1, \ldots, n.$$

L'itération générique de la méthode de Gauss-Seidel utilise les $i-1$ composantes mises à jour de $y^{(k+1)}$ aussitôt qu'elles sont disponibles :

$$y_i^{(k+1)} = g_i(y_1^{(k+1)}, \ldots, y_{i-1}^{(k+1)}, y_{i+1}^{(k)}, \ldots, y_n^{(k)}, z) \qquad i = 1, \ldots, n.$$

On utilise le même critère d'arrêt que pour le cas linéaire, c'est-à-dire on stoppera les iterations lorsque la condition suivante

$$\frac{|y_i^{(k+1)} - y_i^{(k)}|}{|y_i^{(k)}| + 1} < \varepsilon \qquad i = 1, 2, \ldots, n$$

est satisfaite ou ε est une tolérance donnée.

La convergence ne peut être établie au préalable car la matrice $M^{-1}N$ du théorème 9.1 change à chaque itération. La structure de l'algorithme itératif est identique à celle présentée pour le cas linéaire :

Initialiser $y^{(1)}$, $y^{(0)}$, ε et le nombre maximum d'itérations
while ~`converged`$(y^{(0)}, y^{(1)}, \varepsilon)$ **do**
  $y^{(0)} = y^{(1)}$ (Conserver l'itération précédente dans $y^{(0)}$)
  Évaluer $y^{(1)}$ avec Jacobi, Gauss-Seidel ou SOR
  Test sur le nombre d'itérations
end while

11.2.2 Formalisation de la méthode du point fixe

Il s'agit d'une formulation générale des méthodes itératives ou le système d'équations est formalisé comme

$$y = g(y)$$

et l'itération du point fixe s'écrit

$$y^{(k+1)} = g(y^{(k)}) \qquad k = 0, 1, 2, \ldots$$

étant donné un vecteur de départ $y^{(0)}$. La condition de convergence est

$$\rho\left(\nabla g(y^{\text{sol}})\right) < 1$$

où

$$\nabla g(y^{\text{sol}}) = \begin{bmatrix} \frac{\partial g_1}{\partial y_1}\Big|_{y_1=y_1^{\text{sol}}} & \cdots & \frac{\partial g_1}{\partial y_n}\Big|_{y_n=y_n^{\text{sol}}} \\ \vdots & & \vdots \\ \frac{\partial g_n}{\partial y_1}\Big|_{y_1=y_1^{\text{sol}}} & \cdots & \frac{\partial g_n}{\partial y_1}\Big|_{y_1=y_1^{\text{sol}}} \end{bmatrix}$$

est la matrice Jacobienne évaluée à la solution y^{sol}.

Exemple 11.1 Illustrons la méthode du point fixe en résolvant le système d'équations non-linéaires suivant :

$$F(y) = 0 \quad \Leftrightarrow \quad \begin{cases} f_1(y_1, y_2): & y_1 + y_2 - 3 = 0 \\ f_2(y_1, y_2): & y_1^2 + y_2^2 - 9 = 0 \end{cases}$$

La Figure 11.7 montre les deux solutions $y = \begin{bmatrix} 0 & 3 \end{bmatrix}'$ et $y = \begin{bmatrix} 3 & 0 \end{bmatrix}'$ qu'admet ce système d'équations.

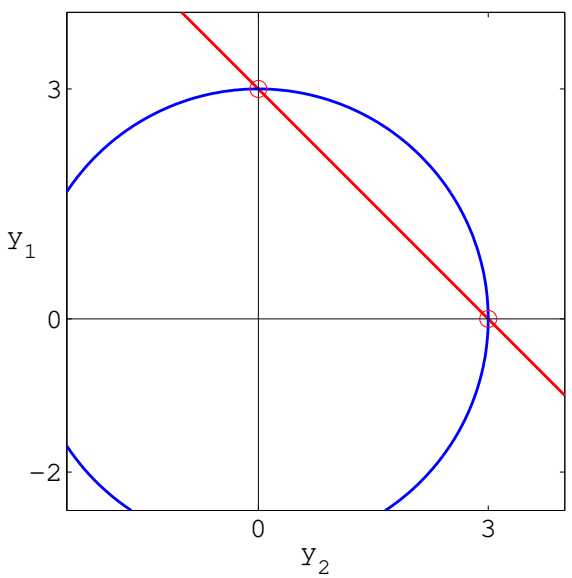
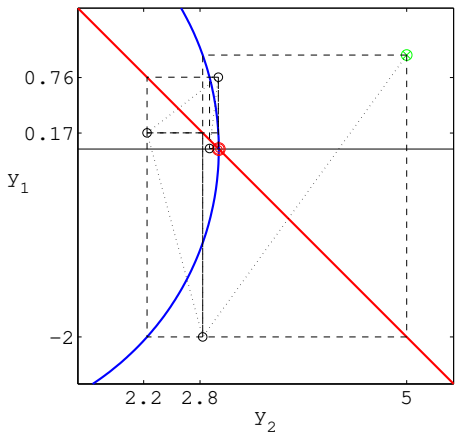


FIG. 11.7 – Solutions du système d'équations de l'exercise 11.1.

En choisissant comme point de départ $y^{(0)} = \begin{bmatrix} 1 & 5 \end{bmatrix}'$ et une tolérance de 10^{-5}, l'algorithme du point fixe pour ce problème peut s'écrire comme :

```
y1 = [1 5]'; tol = 1e-5; y0 = 1+y1; % Pour premier test dans while
while ~converged(y0,y1,tol)
   y0 = y1;
   y1(1) = -y0(2) + 3;
   y1(2) = sqrt(-y0(1)^2 + 9);
end
```

La fonction Matlab `converged` a été présentée à la page 75. La Figure 11.8 montre comment l'algorithme chemine vers la solution.



Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	1.0000	5.0000	1.0000	2.0000
1	−2.0000	2.8284	−2.0000	−0.1716
2	0.1716	2.2361	0.1716	−0.7639
3	0.7639	2.9951	0.7639	−0.0049
4	0.0049	2.9011	0.0049	−0.0989
5	0.0989	3.0000	0.0989	−0.0000
6	0.0000	2.9984	0.0000	−0.0016
7	0.0016	3.0000	0.0016	−0.0000
8	0.0000	3.0000	0.0000	−0.0000

FIG. 11.8 – Fonctionnement de l'algorithme du point fixe avec $y^{(0)} = [1 \;\; 5]$.

En choisissant comme valeur initiale $y = \begin{bmatrix} 0 & 2 \end{bmatrix}'$ l'algorithme oscille entre $y = \begin{bmatrix} 0 & 0 \end{bmatrix}'$ et $y = \begin{bmatrix} 3 & 3 \end{bmatrix}'$ sans converger. Ceci est montré dans la Figure 11.9.

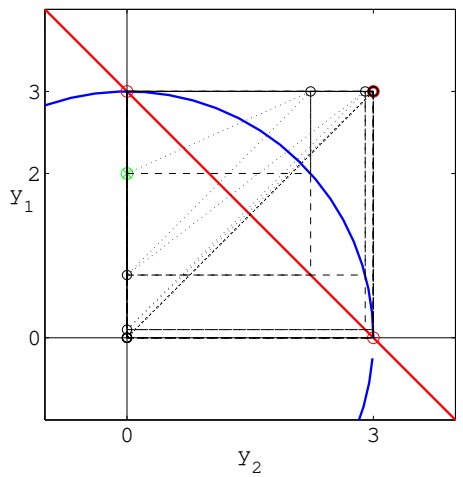


FIG. 11.9 – Fonctionnement de l'algorithme du point fixe avec $y^{(0)} = [0 \;\; 2]$.

11.2.3 Méthode de Newton

La méthode de Newton pour la résolution d'un système d'équations est une généralisation de l'algorithme de Newton pour la recherche du zéro d'une fonction unidimensionnelle. Le fait qu'un algorithme pour un problème unidimensionnel puisse être efficacement généralisé au cas multidimensionnel est une situation exceptionnelle. La méthode de Newton est souvent aussi appelée méthode de *Newton-Raphson*.

Dans la formulation classique on écrit le système d'équations comme :

$$F(y) = 0 \qquad \equiv \qquad \begin{cases} f_1(y) = 0 \\ \vdots \\ f_n(y) = 0 \end{cases}$$

Cette écriture est équivalente à la notation $h(y, z) = 0$, à la différence que les variables z sont directement dans F.

On approche la solution y^* par la séquence $\{y^{(k)}\}_{k=0,1,2,\ldots}$. Étant donné $y^{(k)} \in \mathbb{R}^n$ et une évaluation de la matrice Jacobienne

$$\nabla F(y^{(k)}) = \begin{bmatrix} \left.\frac{\partial f_1}{\partial y_1}\right|_{y_1=y_1^{(k)}} & \cdots & \left.\frac{\partial f_1}{\partial y_n}\right|_{y_n=y_n^{(k)}} \\ \vdots & & \vdots \\ \left.\frac{\partial f_n}{\partial y_1}\right|_{y_1=y_1^{(k)}} & \cdots & \left.\frac{\partial f_n}{\partial y_n}\right|_{y_n=y_n^{(k)}} \end{bmatrix}$$

on construit une approximation meilleure $y^{(k+1)}$ de y^*. Pour ce faire on approche $F(y)$ dans le voisinage de $y^{(k)}$ par une fonction affine :

$$F(y) \approx F(y^{(k)}) + \nabla F(y^{(k)})(y - y^{(k)}).$$

On peut résoudre ce "modèle local" pour obtenir une valeur de y qui satisfasse

$$F(y^{(k)}) + \nabla F(y^{(k)})(y - y^{(k)}) = 0$$

c'est-à-dire

$$y = y^{(k)} - \left(\nabla F(y^{(k)})\right)^{-1} F(y^{(k)}).$$

On construira alors un nouveau modèle local autour de la valeur obtenue pour y. A l'itération k on a

$$y^{(k+1)} = y^{(k)} - \left(\nabla F(y^{(k)})\right)^{-1} F(y^{(k)})$$

et $y^{(k+1)}$ est solution du système linéaire

$$\underbrace{\nabla F(y^{(k)})}_{J} \; \underbrace{(y^{(k+1)} - y^{(k)})}_{s} = - \underbrace{F(y^{(k)})}_{b} .$$

Algorithme 21 Méthode de Newton.

1: Donner $y^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: Évaluer $b = -F(y^{(k)})$ et $J = \nabla F(y^{(k)})$
4: Vérifier la condition de J
5: `solve` $J\,s = b$
6: $y^{(k+1)} = y^{(k)} + s$
7: **end for**

Exemple 11.2 Illustrons la méthode de Newton en résolvant le système d'équations de l'exemple 11.1 soit

$$F(y) = 0 \quad \Leftrightarrow \quad \begin{cases} f_1(y_1, y_2): \; y_1 + y_2 - 3 = 0 \\ f_2(y_1, y_2): \; y_1^2 + y_2^2 - 9 = 0 \end{cases}$$

et dont la matrice Jacobienne s'écrit

$$\nabla F(y^{(k)}) = \begin{bmatrix} \frac{\partial f_1}{\partial y_1}\Big|_{y_1=y_1^{(k)}} & \frac{\partial f_1}{\partial y_2}\Big|_{y_2=y_2^{(k)}} \\ \frac{\partial f_2}{\partial y_1}\Big|_{y_1=y_1^{(k)}} & \frac{\partial f_2}{\partial y_2}\Big|_{y_2=y_2^{(k)}} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2y_1^{(k)} & 2y_2^{(k)} \end{bmatrix}.$$

Si l'on choisit $y^{(0)} = \begin{bmatrix} 1 & 5 \end{bmatrix}'$ alors

$$F(y^{(0)}) = \begin{bmatrix} 3 \\ 17 \end{bmatrix} \qquad \text{et} \qquad \nabla F(y^{(0)}) = \begin{bmatrix} 1 & 1 \\ 2 & 10 \end{bmatrix}.$$

En résolvant le système

$$\begin{bmatrix} 1 & 1 \\ 2 & 10 \end{bmatrix} s^{(0)} = \begin{bmatrix} -3 \\ -17 \end{bmatrix}$$

l'on obtient $s^{(0)} = \begin{bmatrix} -13/8 & -11/8 \end{bmatrix}'$ d'où

$$y^{(1)} = y^{(0)} + s^{(0)} = \begin{bmatrix} -.625 \\ 3.625 \end{bmatrix}, \qquad F(y^{(1)}) = \begin{bmatrix} 0 \\ \frac{145}{32} \end{bmatrix}, \qquad \nabla F(y^{(1)}) = \begin{bmatrix} 1 & 1 \\ -\frac{5}{4} & \frac{29}{4} \end{bmatrix}.$$

En résolvant à nouveau le système

$$\begin{bmatrix} 1 & 1 \\ -\frac{5}{4} & \frac{29}{4} \end{bmatrix} s^{(1)} = \begin{bmatrix} 0 \\ -\frac{145}{32} \end{bmatrix}$$

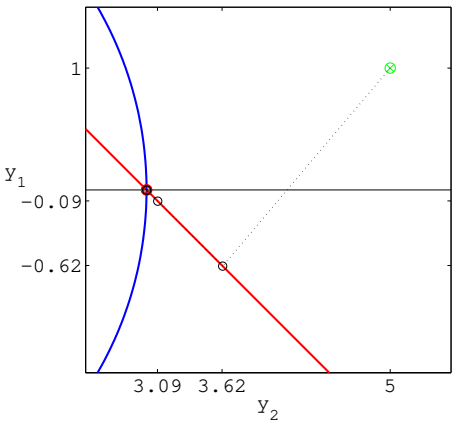
on obtient $s^{(1)} = \begin{bmatrix} 145/272 & -145/272 \end{bmatrix}'$ d'où

$$y^{(2)} = y^{(1)} + s^{(1)} = \begin{bmatrix} -.092 \\ 3.092 \end{bmatrix}.$$

En se servant du code Matlab qui suit (en initialisant `y1`) :

```
tol = 1e-5; y0 = 1+y1; % Pour premier test dans while
while ~converged(y0,y1,tol)
   y0 = y1;
   F(1) = y0(1) + y0(2) - 3;
   F(2) = y0(1)^2 + y0(2)^2 - 9;
   b = -F';
   J = [1 1; 2*y0(1) 2*y0(2)];
   s = J \ b;
   y1 = y0 + s;
end
```

on peut calculer les itérations jusqu'à la convergence et la Figure 11.10 montre comme ces itérations convergent vers la solution $y = \begin{bmatrix} 0 & 3 \end{bmatrix}'$. Rappelons que la fonction Matlab `converged` a été introduite à la page 75.



Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	1.0000	5.0000	1.0000	2.0000
1	−0.6250	3.6250	−0.6250	0.6250
2	−0.0919	3.0919	−0.0919	0.0919
3	−0.0027	3.0027	−0.0027	0.0027
4	−0.0000	3.0000	0.0000	0.0000

FIG. 11.10 – Fonctionnement de l'algorithme de Newton avec $y^{(0)} = [1 \;\; 5]$.

Remarquons que la méthode de Newton converge aussi lorsqu'on choisit comme valeur initiale $y^{(0)} = \begin{bmatrix} 0 & 2 \end{bmatrix}'$ ce qui est illustré dans la Figure 11.11.

Convergence

Le *taux de convergence* r d'une méthode itérative est défini comme

$$\lim_{k\to\infty} \frac{\|e^{(k+1)}\|}{\|e^{(k)}\|^r} = c$$

ou $e^{(k)}$ est l'erreur à l'itération k et c une constante finie. On distingue alors les cas particuliers suivants :

– $r = 1$ et $c < 1$, convergence *linéaire*
– $r > 1$, convergence *super-linéaire*
– $r = 2$, convergence *quadratique*

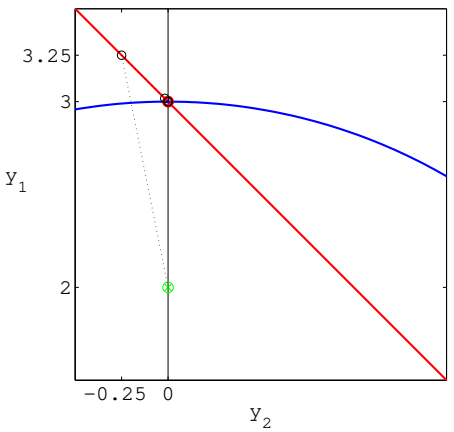


FIG. 11.11 – Fonctionnement de l'algorithme de Newton avec $y^{(0)} = [\,0 \;\; 2\,]$.

Dans la pratique ceci correspond à un gain de précision par itération d'un nombre de digits constant pour une convergence linéaire ; l'accroissement de la précision par itération va en augmentant pour une convergence super-linéaire, et dans le cas d'une convergence quadratique la précision double à chaque itération.

La méthode de Newton converge quadratiquement sous certaines conditions. On vérifie

$$\| \, y^{(k+1)} - y^* \, \| \quad \leq \quad \beta\gamma \, \| \, y^{(k)} - y^* \, \|^2 \quad k = 0, 1, 2, \ldots \tag{11.3}$$

où β mesure la non-linéarité relative $\| \, \nabla F(y^*)^{-1} \, \| \leq \; \beta < 0$ et γ est la constante de Lipschitz.

La convergence est seulement garantie si $y^{(0)}$ se trouve dans un voisinage de y^*, voisinage qu'il faut définir.

Pour la solution de modèles macroéconométriques, $y^{(0)}$ est défini par y_{t-1} ce qui constitue en général un bon voisinage.

La convergence quadratique des itérations de l'exemple 11.2 peut être vérifiée dans le tableau de la Figure 11.10. En effet pour chaque composante $y_i^{(k)}$, on vérifie

$$|y_i^{(k+1)} - y_i^{\text{sol}}| < |y_i^{(k)} - y_i^{\text{sol}}|^2 \, .$$

La complexité de la méthode de Newton est déterminée par la résolution du système linéaire. Si l'on compare les méthodes itératives avec la méthode de Newton pour une itération on remarque que la différence de travail réside dans l'évaluation de la matrice Jacobienne et la solution d'un système linéaire. Ceci peut paraître un désavantage de la méthode de Newton. Dans la pratique cependant on peut montrer que lorsque on résoud le système linéaire avec une méthode directe creuse et lorsqu'on utilise des dérivées analytiques les deux méthodes sont de complexité comparable étant donné que le nombre d'itérations pour Newton est nettement inférieur aux nombre d'itérations pour les méthodes itératives.

11.2.4 Quasi-Newton

La méthode de Newton nécessite à chaque itération le calcul de la matrice Jacobienne, ce qui requiert l'évaluation de n^2 dérivées, et la résolution d'un système linéaire ($O(n^3)$). Dans la situation ou l'évaluation des dérivées est coûteuse on peut remplacer le calcul exact de la Jacobienne par une simple mise à jour. Ceci donne lieu à des nombreuses variantes de la méthode de Newton, appellees méthodes *quasi-Newton*.

Méthode de Broyden

Dans ce cas la mise à jour de la matrice Jacobienne se fait avec des matrices de rang un. Étant donné une approximation $B^{(k)}$ de la matrice Jacobienne à l'itération k on calcule l'approximation de l'itération $k+1$ comme

$$B^{(k+1)} = B^{(k)} + \frac{\left(dF^{(k)} - B^{(k)}\, s^{(k)}\right) s^{(k)\prime}}{s^{(k)\prime}\, s^{(k)}}$$

ou $dF^{(k)} = F(y^{(k+1)}) - F(y^{(k)})$ et $s^{(k)}$ est la solution de $B^{(k)}s^{(k)} = -F(y^{(k)})$. L'algorithme de Broyden est formalisé ci-après.

Algorithme 22 Algorithme de Broyden.

1: Donner $y^{(0)}$ et $B^{(0)}$ (approximation de $\nabla F(y^{(0)})$)
2: **for** $k = 0, 1, 2, \ldots$ `until` convergence **do**
3: `solve` $B^{(k)}\, s^{(k)} = -F(y^{(k)})$
4: $y^{(k+1)} = y^{(k)} + s^{(k)}$
5: $\triangle = F(y^{(k+1)}) - F(y^{(k)})$
6: $B^{(k+1)} = B^{(k)} + \left(\triangle - B^{(k)}\, s^{(k)} \right) s^{(k)\prime} / (s^{(k)\prime}\, s^{(k)})$
7: **end for**

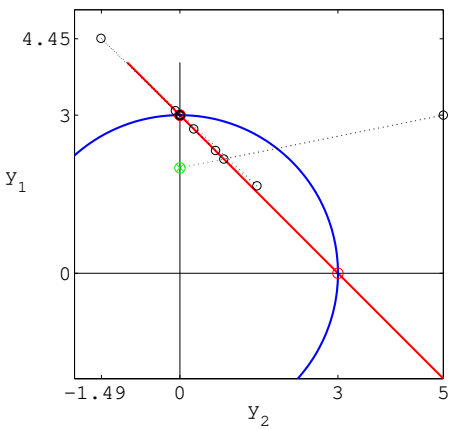
Le code Matlab qui suit réalise la méthode de Broyden.

```
y0 = - y1; tol = 1e-5; B = eye(2);
F1 = feval('ExSNL',y1);
while ~converged(y0,y1,tol)
   y0 = y1;
   F0 = F1;
   s = B \ -F0;
   y1 = y0 + s;
   F1 = feval('ExSNL',y1);
   dF = F1 - F0;
   B = B + ((dF - B*s)*s')/(s'*s);
end
```

L'évaluation de la fonction $F(y)$ a été dans ce cas transférée dans une fonction Matlab `ExSNL` ce qui rend le code indépendant de la résolution d'un système d'équations particulier.

```
function F = ExSNL(y)
F = repmat(NaN,2,1);
F(1) = y(1) + y(2) - 3;
F(2) = y(1)^2 + y(2)^2 - 9;
```

La Figure 11.12 donne les résultats de méthode de Broyden en partant avec $B^{(0)} = I$ et pour une valeur initiale $y^{(0)} = [\,2 \;\; 0\,]$.



Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	2.0000	0	−1.0000	0
1	3.0000	5.0000	0	5.0000
2	2.1667	0.8333	−0.8333	0.8333
3	1.6585	1.4634	−1.3415	1.4634
4	4.4582	−1.4984	1.4582	−1.4984
5	2.3304	0.6762	−0.6696	0.6762
6	2.7384	0.2614	−0.2616	0.2614
7	3.0847	−0.0852	0.0847	−0.0852
8	2.9921	0.0079	−0.0079	0.0079
9	2.9998	0.0002	−0.0002	0.0002
10	3.0000	0.0000	0.0000	0.0000

FIG. 11.12 – Résultats de l'algorithme de Broyden avec $B^{(0)} = I$ et $y^{(0)} = [\,2 \;\; 0\,]$.

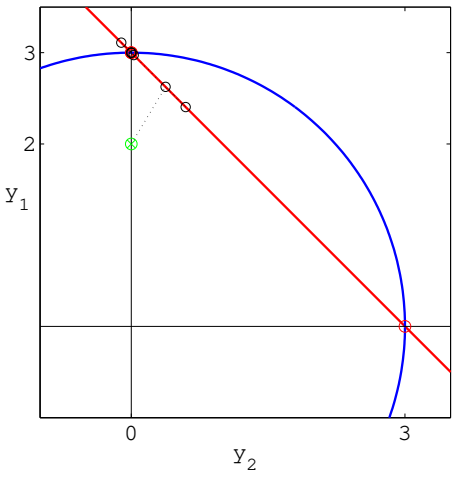
Lorsque l'algorithme démarre avec $B^{(0)} = I$ aucune information sur la matrice Jacobienne n'est donné et il n'est pas surprenant que la performance soit médiocre (comparable à la méthode du point fixe). Ci-après on fait démarrer l'algorithme avec $B^{(0)} = \nabla F(y^{(0)})$ et on peut observer que dans ce cas sa performance est bien supérieure. Ces résultats sont reproduits dans la Figure 11.13.

11.2.5 Newton amorti (*damped*)

Si la valeur initiale se trouve loin de la solution, la méthode de Newton et ses variantes convergent en général très mal. En fait la direction et surtout la longueur du pas sont peu fiables. Afin de définir un pas plus prudent on peut calculer $y^{(k+1)}$ à l'itération k comme

$$y^{(k+1)} = y^{(k)} + \alpha^{(k)}\, s^{(k)}$$

ou $\alpha^{(k)}$ est un scalaire à déterminer. Ainsi on choisira $0 < \alpha^{(k)} < 1$ lorsqu'on est loin de la solution et $\alpha^{(k)} = 1$ lorsque $y^{(k)}$ est proche de la solution. Une façon de contrôler le paramètre $\alpha^{(k)}$ est de le lier à la valeur de $\|F(y^{(k)})\|$.



Iter	Solution		Erreur	
k	$y_1^{(k)}$	$y_2^{(k)}$	$y_1^{(k)} - y_1^{\text{sol}}$	$y_2^{(k)} - y_2^{\text{sol}}$
0	2.0000	0	−1.0000	0
1	2.6250	0.3750	−0.3750	0.3750
2	2.4044	0.5956	−0.5956	0.5956
3	3.1100	−0.1100	0.1100	−0.1100
4	2.9739	0.0261	−0.0261	0.0261
5	2.9991	0.0009	−0.0009	0.0009
6	3.0000	0.0000	0.0000	0.0000

FIG. 11.13 – Algorithme de Broyden avec $B^{(0)} = \nabla F(y^{(0)})$ et $y^{(0)} = [\,2 \;\; 0\,]$.

11.2.6 Solution par minimisation

Pour résoudre $F(y) = 0$ on peut minimiser la fonction objectif suivante

$$g(y) = \| F(y) \|_p$$

où p est une norme quelconque dans $\mathbb{R}^n$. Une raison qui motive cette alternative est qu'elle introduit un critère pour décider si $y^{(k+1)}$ est une approximation meilleure pour y^* que $y^{(k)}$. Comme à la solution $F(y^*) = 0$ on peut être tenté de comparer les vecteurs $F(y^{(k+1)})$ et $F(y^{(k)})$ en calculant leur norme respective. Ce que l'on désire est que

$$\| F(y^{(k+1)}) \|_p \; < \; \| F(y^{(k)}) \|_p$$

ce qui conduit à minimiser la fonction objectif

$$\min_y \; g(y) = \frac{1}{2} F(y)' F(y)$$

si l'on choisit $p = 2$ pour la norme. On utilisera l'algorithme de Gauss-Newton ou Levenberg-Marquardt.

11.3 Tableau synoptique des méthodes

La figure 11.15 résume les combinaisons possibles des méthodes pour la resolution des systèmes d'équations linéaires et non-linéaires.

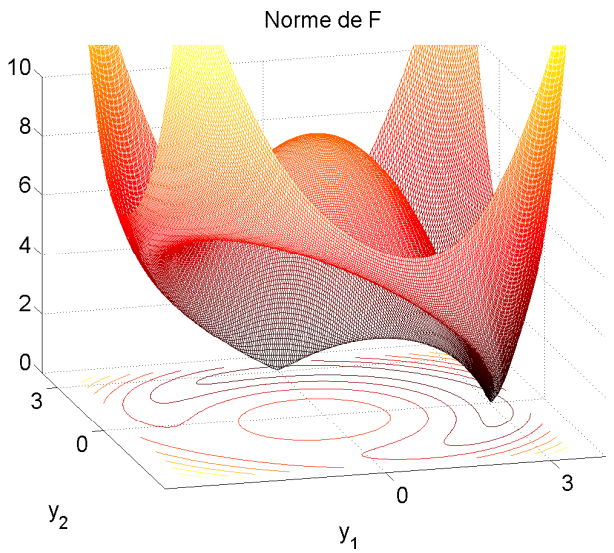


FIG. 11.14 – Graphe de $\| F(y) \|_p$ du système d'équations de l'exemple 11.1.

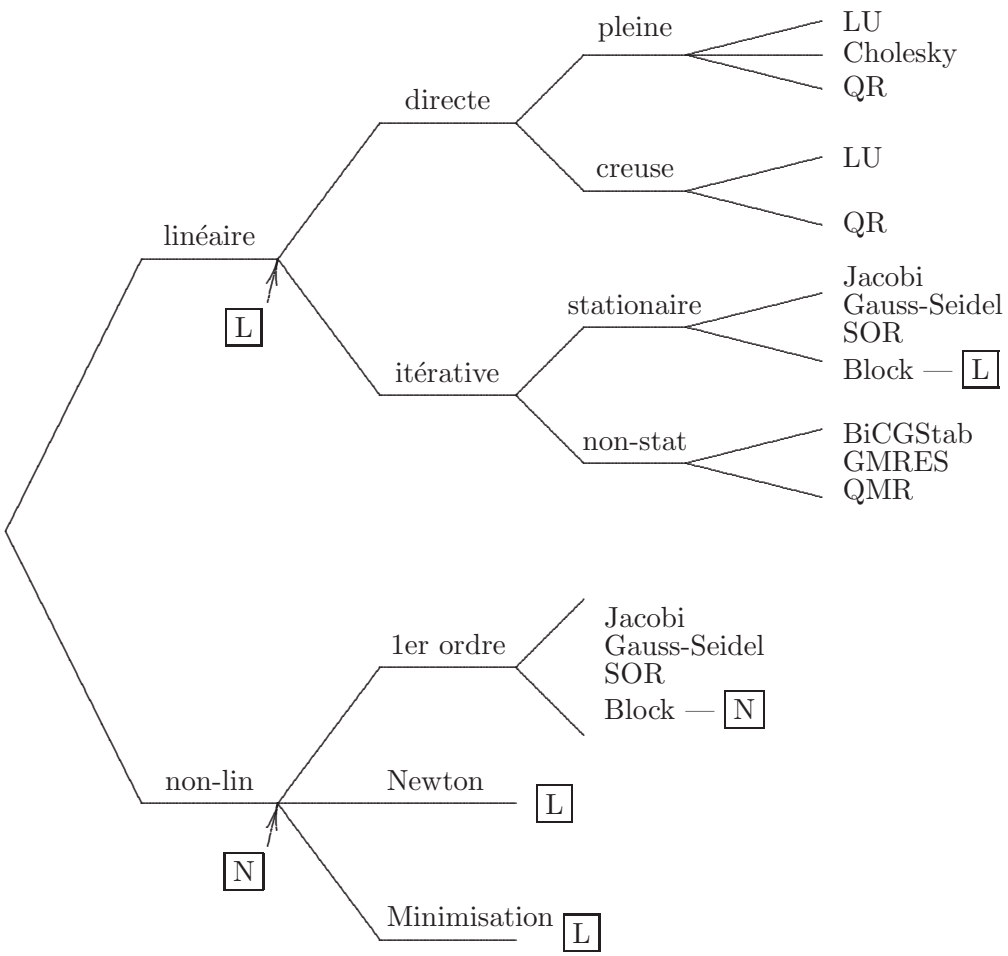


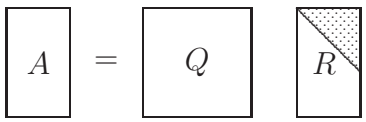
FIG. 11.15 – Tableau synoptique des méthodes pour la solution de systèmes d'équations.

Chapitre 12

Décompositions orthogonales

12.1 La factorisation QR

La factorisation QR d'une matrice $A \in \mathbb{R}^{m\times n}$ est donnée par



avec $Q \in \mathbb{R}^{m\times m}$ une matrice orthogonale et $R \in \mathbb{R}^{m\times n}$ une matrice triangulaire supérieure.

Exemple 12.1 La factorisation QR d'une matrice A s'obtient avec la commande `[Q,R]=qr(A)`

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 1 & 4 \end{bmatrix} \qquad Q = \begin{bmatrix} -\frac{\sqrt{2}}{2} & \frac{\sqrt{3}}{3} & -\frac{\sqrt{6}}{6} \\ 0 & -\frac{\sqrt{3}}{3} & -\frac{2\sqrt{6}}{6} \\ -\frac{\sqrt{2}}{2} & -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} \end{bmatrix} \qquad R = \begin{bmatrix} -\sqrt{2} & -3\sqrt{3} \\ 0 & -\sqrt{3} \\ 0 & 0 \end{bmatrix}$$

et on vérifie que $Q'Q = I$. Le calcul de R nécessite $\sim 3n^2(m - \frac{n}{3}$ et le calcul de Q nécessite $\sim 4m^2n$.

La transformation de Gauss, utilisée pour la factorisation LU, est une technique pour mettre à zero certaines composantes d'un vecteur.

D'autres techniques consistent soit à faire subir au vecteur une rotation de sorte qu'il se trouve dans la direction d'une des axes du repaire orthonormé, soit d'opérer une reflexion du vecteur a travers sa bisectrice ce qui lui donne a nouveau la direction d'une des axes du repère.

12.1.1 Matrices de Givens

Une rotation du vecteur x de α (radians) dans le sens contraire aux aiguilles d'une montre est obtenu avec la transformation Gx avec

$$G = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

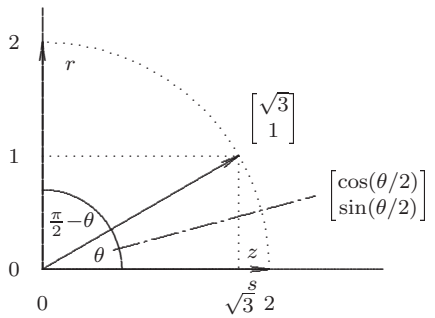
et une reflexion du vecteur x par rapport à sa bisectrice $[\cos(\theta/2)\ \sin(\theta/2)]'$, c'est-à-dire la droite d'angle $\theta/2$ entre x et le repère, avec θ l'angle du vecteur x, est obtenu au moyen de la transformation Fx avec

$$F = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \end{bmatrix} \quad .$$

La matrice G est appellé *matrice de rotation de Givens* et la matrice F est appellé *matrice de reflexion de Givens*.

Les matrices G et F sont des matrices orthogonales et on rapelle q'une matrice $F \in \mathbb{R}^{n\times n}$ est dite orthogonale si $F'F = I_n$. Les matrices orthogonales jouent un rôle important dans les considérations qui suivent.

Exemple 12.2 Soit le vecteur $x = [\sqrt{3}\ 1]'$ et θ son angle (avec MATLAB $\theta =$ `atan2(x(2),x(1))`). Alors la rotation $r = Gx$ avec $\alpha = \pi/2 - \theta$ donne $r = [0\ 2]'$ et une rotation avec $\alpha = -\theta$ donne $s = [2\ 0]'$. Une reflexion $z = Fx$ donne $z = [2\ 0]'$.



On voit dans le graphique que la transformation orthogonale ne modifit pas la longeur euclidienne du vecteur transformé. On a $\parallel Fx \parallel = \parallel z \parallel$ et $\parallel Gx \parallel = \parallel r \parallel$.

12.1.2 Réflexion de Householder

Soit un vecteur $v \in \mathbb{R}^n$, alors une matrice de la forme

$$H = I - \frac{2vv'}{v'v}$$

est appellée une *matrice de Householder*. On peut vérifier que la matrice H est symétrique et orthogonale.

Une matrice de Householder transforme un vecteur x en le reflechissant à travers l'hyperplan qui est perpendiculaire au vecteur v. Ceci est illustré dans l'exemple de gauche de la figure **??**.

Maintenant on voudrait choisir v de telle sorte que Hx devienne un multiple de e_1, ce qui a pour consequence d'annuler toutes les composantes de x sauf la première.

Pour $x \in \mathbb{R}^n$ le produit Hx s'écrit

$$Hx = (I - \frac{2vv'}{v'v})x = x - \frac{2v'x}{v'v}v \quad .$$

Si l'on veut que Hx engendre l'espace défini par e_1 il faut que v soit une combinaison lineaire de x et e_1. Posons donc $v = x + \alpha e_1$, alors on obtient

$$v'x = (x' + \alpha e_1')x = x'x + \alpha x_1$$

et

$$v'v = (x' + \alpha e_1')(x + \alpha e_1) = x'x + 2\alpha x_1 + \alpha^2$$

et donc

$$Hx = (1 - 2\frac{x'x + \alpha x_1}{x'x + 2\alpha x_1 + \alpha^2})x - \frac{2\alpha v'x}{v'v}e_1 \quad .$$

Pour pouvoir annuler le coefficient de x dans l'expression ci-dessus il faut poser $\alpha = \mp \parallel x \parallel_2$ d'ou l'on tire que

$$v = x \mp \parallel x \parallel_2 e_1$$

et on verifie que

$$Hx = (I - 2\frac{vv'}{v'v})x = \mp \parallel x \parallel_2 e_1 \quad .$$

Exemple 12.3 Soit le vecteur $x = [2\ .5]'$ et $v = [-.7\ 1]$. A gauche est representé la reflexion de x à travers l'hyperplan perpendiculaire à v. Ensuite on rédefinit $v = x - \parallel x \parallel_2 * e_1 = [-.0616\ .5]$ et on peut verifier que la reflexion de x à travers l'hyperplan perpendiculaire à v se confond avec e_1.

12.1.3 Algorithme de Householder - Businger - Golub

Il s'agit de factoriser une matrice $A \in \mathbb{R}^{m\times n}$ en $A = QR$ où Q est une matrice orthogonale et R une matrice triangulaire.

On cherche $H_1 = I - \frac{2v_1v_1'}{v_1'v_1}$ telle que

$$H_1A = \begin{bmatrix} \alpha_1 & \times & \cdots & \times \\ 0 & \times & \cdots & \times \\ \vdots & \vdots & & \vdots \\ 0 & \times & \cdots & \times \end{bmatrix} \quad ,$$

puis on cherche $H_2 = I - \frac{2v_2v_2'}{v_2'v_2}$ telle que

$$H_2H_1A = \begin{bmatrix} \alpha_1 & \times & \times & \cdots & \times \\ 0 & \alpha_2 & \times & \cdots & \times \\ 0 & 0 & \times & \cdots & \times \\ 0 & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \times & \cdots & \times \end{bmatrix} \quad .$$

Finalement la decomposition sera donné par

$$\underbrace{H_n \cdots H_2H_1}_{Q'} A = R \quad .$$

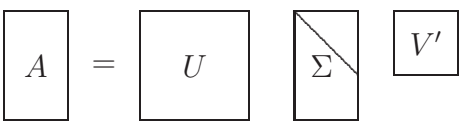
(TP : Golub p.231, prob. 5.3.6)

12.2 Décomposition singulière SVD

Soit une matrice $A \in \mathbb{R}^{m\times n}$ alors il existe deux matrices orthogonales

$$U = [u_1, \ldots, u_m] \in \mathbb{R}^{m\times m} \quad \text{et} \quad V = [v_1, \ldots, v_n] \in \mathbb{R}^{n\times n}$$

telles que



avec $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m\times n}$ et $p = \min(m, n)$. Les σ_i sont appellés les *valeurs singulières* et elles vérifient $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$. Les vecteurs u_i et v_i sont appellés les *vecteurs singuliers* de gauche respectivement de droite.

Exemple 12.4 Avec MATLAB on obtient la décomposition singulière avec la commande `[U,S,V]=svd(A)`.

12.2.1 Approximation d'une matrice par une somme de matrices de rang un

Comme les matrices U et V sont orthogonales on vérifie que la matrice A peut s'exprimer comme une somme de matrices de rang un, soit

$$A = U\Sigma V' = \sigma_1 u_1 v_1' + \ldots + \sigma_p u_p v_p'$$

Si $k < r =$ rang(A) l'approximation de A definie comme

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i'$$

verifie

$$\| A - A_k \|_2 = \sigma_{k+1}$$

ce que l'on peut interpréter comme une mésure de la qualité de l'approximation.

12.2.2 Calcul numérique du rang d'une matrice

La décomposition singulière d'une matrice constitue un algorithme numériquement très stable. De ce fait cet algorithme se prete bien pour établir le rang numérique d'une matrice dans une situation ou le problème est mal conditionné. Si les valeurs singulières d'une matrice A verifient

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = \sigma_p = 0$$

alors on a rang$(A) = r$. Dans MATLAB la fonction `rank` correspond à :

```
s=svd(X);
tol=max(size(X))*s(1)*eps;
rang=sum(s > tol);
```

Il existe d'autres algorithmes dont la complexité est inférieure pour établir le rang d'une matrice. La décomposition singulière offre cependant les resultats les plus fiables dans des situations ou le problème est mal conditionné.

12.2.3 Interprétation des valeurs singulières

Il y a un lien entre les valeurs propres de $A'A$ et AA' avec les valeurs singulières de A. Soit

$$\Sigma = U'AV$$

qui est une matrice diagonale, alors

$$\Sigma'\Sigma = (U'AV)'U'AV = V'A'UU'AV = V'A'AV$$

et on conclut que les valeurs singulières au carré correspondent aux valeurs propres de $A'A$. De façon analogue on peut écrire

$$\Sigma\Sigma' = U'AV(U'AV)' = U'AVV'A'U = UAA'U$$

et on conclut que les σ_i^2 sont aussi les valeurs propres de AA'.

Les valeurs propres de $A'A$ ont une interpretation géometrique. Il s'agit de la demi-longeur des axes de l'hyperellipse $x'A'Ax - 1 = 0$. Ils décrivent comment le nuage des points (observations) est dispersé dans l'espace. Le rapport de deux valeurs singulières correspondantes correspond au rapport des axes de l'ellipse. Ainsi les valeurs singulières caracteristent l'élongation de l'hyperellipse. Ceci est illustré dans la figure 12.1.

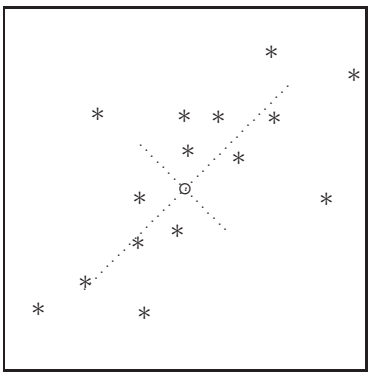


FIG. 12.1 – Ellipse definie par $x'A'Ax - 1 = 0$.

Il s'ensuit que la condition d'une matrice A d'un système linéaire correspond à

$$\kappa_2(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)} \quad .$$

12.2.4 Complexité

La décomposition singulière necessite $4m^2n+8mn^2+9n^3 > 21n^3$ opérations élémentaires. Il existe des versions plus économiques de l'algorithme lorsque on ne s'interesse que soit à S, soit à S et U ou soit à S et V. On remarque que cette décomposition est très couteuse. Son intérêt est néanmoins de grande importance.

12.2.5 Calcul du pseudoinverse

Soit une matrice $A \in \mathbb{R}^{m\times n}$ et U, V et Σ les matrices qui correspondent à sa décomposition singulière. Si l'on définit

$$A^+ = V\Sigma^+U' \quad \text{avec} \quad \Sigma^+ = \text{diag}(\frac{1}{\sigma_1}, \cdots, \frac{1}{\sigma_r}, 0, \ldots, 0)$$

alors A^+ vérifie les 4 conditions de Moore-Penrose (1955).

$$AA^+A = A \qquad A^+AA^+ = A^+ \qquad (AA^+)' = AA^+ \qquad (A^+A)' = A^+A \quad .$$

Exemple 12.5 Soit une matrice A et sa décomposition sigulière obtenu avec la commande `[U,S,V]=svd(A)`

$$A = \begin{bmatrix} 2 & 1 & 6 & 1 \\ 1 & 1 & 1 & 2 \\ 0 & 1 & 2 & 3 \end{bmatrix} \quad U = \begin{bmatrix} 0.863 & 0.505 & -0.002 \\ 0.286 & -0.485 & 0.827 \\ 0.416 & -0.714 & -0.563 \end{bmatrix} \quad S = \begin{bmatrix} 7.304 & 0 & 0 & 0 \\ 0 & 2.967 & 0 & 0 \\ 0 & 0 & 0.918 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.275 & 0.177 & 0.896 & -0.302 \\ 0.214 & -0.234 & 0.285 & 0.905 \\ 0.862 & 0.376 & -0.339 & 0.000 \\ 0.367 & -0.879 & -0.041 & -0.302 \end{bmatrix}$$

alors son pseudoinverse correspond à `V( :,1 :r)*diag(ones(r,1)./s)*U( :,1 :r)'` ou r est le nombre de valeurs singulières non nulles.

$$A^+ = \begin{bmatrix} 0.275 & 0.177 & 0.896 \\ 0.214 & -0.234 & 0.285 \\ 0.862 & 0.376 & -0.339 \\ 0.367 & -0.879 & -0.041 \end{bmatrix} \begin{bmatrix} 0.137 & 0 & 0 \\ 0 & 0.337 & 0 \\ 0 & 0 & 1.089 \end{bmatrix} \begin{bmatrix} 0.863 & 0.286 & 0.416 \\ 0.505 & -0.485 & -0.714 \\ -0.002 & 0.827 & -0.563 \end{bmatrix}$$

$$= \begin{bmatrix} 0.061 & 0.788 & -0.576 \\ -0.015 & 0.303 & -0.106 \\ 0.167 & -0.333 & 0.167 \\ -0.106 & 0.121 & 0.258 \end{bmatrix}$$

Le pseudoinverse peut être obtenu aussi plus simplement avec la commande `pinv(A)`.

Chapitre 13

Moindres carrés

Un problème fondamental en sciences consiste à adapter un modèle à des observations qui sont entachées d'erreurs. Nous nous intéresserons ici au cas où le nombre d'observations est supérieur au nombre de paramètres. Ceci nous conduira à résoudre, suivant le choix du modèle, des systèmes linéaires ou non-linéaires surdéterminés.

La "meilleure solution" d'un système surdéterminé peut être définie de plusieurs façons. Étant donné les observations y, les variables indépendantes X et le modèle $f(X, \beta)$ avec β le vecteur de paramètres, la solution retenue correspond à la solution du problème de minimisation

$$\min_{\beta} \parallel f(X, \beta) - y \parallel_2^2 \ . \tag{13.1}$$

La notation (13.1) est principalement utilisé en statistique. En analyse numérique on note le problème comme suit :

$$\min_{x \in R^n} g(x) = \frac{1}{2} r(x)' r(x) = \frac{1}{2} \sum_{i=1}^{m} r_i(x)^2 \tag{13.2}$$

où $r(x)$ est le vecteur des résidus fonction des paramètres x.

Dans le cas linéaire le modèle s'écrit $Ax \approx b$ avec b les observations, A les variables indépendantes et x le vecteur de paramètres. Le vecteur des résidus est dans ce cas

$$r = Ax - b \ .$$

Si on a un modèle non-linéaire $f(t_i, x)$, la fonction f est non-linéaire par rapport aux paramètres x. Les t_i sont les variables indépendantes et les y_i sont les observations. Le vecteur des résidus s'écrit alors

$$r_i(x) = f(t_i, x) - y_i \qquad i = 1, \ldots, m.$$

Ce choix particulier de minimiser la norm Euclidienne des résidus est motivé par deux raisons. La première raison est de nature statistique car la solution correspond

au BLUE (best linear unbiased estimator) dans le cas du modèle linéaire classique (résidus i.i.d. et $N(0, \sigma)$). La deuxième raison est de nature numérique car la solution peut s'obtenir avec des procédures numériques relativement simples.

Il semble que ce soit Gauss qui en 1795 (à l'age de 18 ans) a découvert la méthode des moindres carrés. Le développement des méthodes numériques modernes pour la résolution des moindres carrés s'est fait dans les années soixante (QR, SVD) et plus récemment on a développé les méthodes pour des systèmes qui sont grands et creux.

On présentera d'abord les méthodes pour résoudre les moindres carrés linéaires puis on présentera le cas non-linéaire. La solution des moindres carrés non-linéaires s'obtient par des méthodes itératives qui nécessitent la solution d'un problème de moindres carrés linéaires à chaque étape.

13.1 Moindres carrés linéaires

Souvent, et c'est le cas plus particulièrement pour des problèmes qui apparaissent en économétrie et statistique, on n'est pas seulement intéressé d'obtenir la solution x du problème des moindres carrés $Ax \approx b$, mais on désire aussi calculer la matrice des variances et covariances $\sigma^2(A'A)^{-1}$.

Ainsi on aura besoin d'évaluer d'une part $\|b - Ax\|_2^2$ à la solution, étant donné que cette valeur intervient dans le calcul de $\sigma^2 = \|b - Ax\|_2^2/(m - n)$, et d'autre part d'une méthode efficace et numériquement stable pour calculer $(A'A)^{-1}$ ou un sous-ensemble d'éléments de cette matrice.

Dans la présentation qui suit on donnera des indications comment obtenir ces éléments numériquement suivant l'approche choisi.

13.1.1 Méthode des équations normales

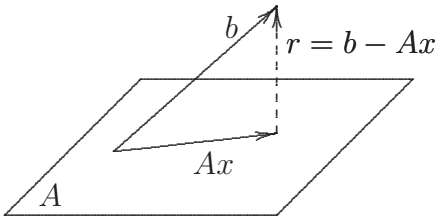
La solution de (13.2) peut être obtenue de plusieurs manières. Une façon consiste à dériver (13.2) par rapport à x et d'écrire les conditions du premier ordre pour le minimum, soit :

$$2A'Ax - 2A'b = 0$$

d'où on tire le système des *équations normales*

$$A'Ax = A'b\ . \tag{13.3}$$

Les équations normales s'obtiennent également en considérant que la solution Ax la plus proche à b s'obtient par une projection orthogonale de b dans l'espace engendré par les colonnes de A.



De ce fait

$$A'r = A'(b - Ax) = 0 \quad \text{d'où} \quad A'Ax = A'b\ .$$

Si la matrice A est de rang complet $A'A$ est définie positive et le système des équations normales peut être résolu en utilisant la factorisation de Cholesky (cf. Théorème 7.3).

En posant

$$C = A'A \qquad \text{et} \qquad c = A'b$$

les étapes de résolution des équations normales $A'Ax = A'b$ procèdent d'abord à la factorisation de Cholesky

$$C = GG'$$

puis à la résolution du système triangulaire inférieur

$$Gz = c$$

et finalement à la résolution du système triangulaire supérieur

$$G'x = z.$$

Factorisation de la matrice bordée

Si l'on désire obtenir $\|r\|_2^2$ il convient de former la matrice bordée

$$\overline{A} = \begin{bmatrix} A & b \end{bmatrix}$$

et de considérer la factorisation de Cholesky de la matrice $\overline{C} = \overline{A}'\overline{A}$

$$\overline{C} = \begin{bmatrix} C & c \\ c' & b'b \end{bmatrix} = \overline{G}\,\overline{G}' \qquad \text{avec} \qquad \overline{G} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix}.$$

La solution x et la norme des résidus s'obtiennent alors à partir de

$$G'x = z \qquad \text{et} \qquad \|Ax - b\|_2 = \rho.$$

Pour démontrer ce résultat il suffit de développer l'égalité entre les matrices $\overline{A}'\overline{A}$ et $\overline{G}\,\overline{G}'$

$$\begin{bmatrix} C & c \\ c' & b'b \end{bmatrix} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix} \begin{bmatrix} G' & z \\ 0 & \rho \end{bmatrix} = \begin{bmatrix} GG' & Gz \\ z'G' & z'z + \rho^2 \end{bmatrix}$$

d'où l'on voit que G correspond à la matrice de Cholesky de $A'A$ et que

$$Gz = c \qquad \text{et} \qquad b'b = z'z + \rho^2.$$

Comme $r = b - Ax$ est orthogonal à Ax,

$$\|Ax\|_2^2 = (r + Ax)'Ax = b'Ax = \underbrace{b'AG'^{-1}}_{z'} z = z'z$$

d'où

$$\|Ax - b\|_2^2 = \underbrace{x'A'Ax}_{z'z} - 2\underbrace{b'Ax}_{z'z} + b'b = b'b - z'z = \rho^2.$$

Calcul de $(A'A)^{-1}$

La matrice $S = (A'A)^{-1}$ peut être obtenu de la façon suivante :

$$\begin{aligned} S &= (A'A)^{-1} \\ &= (GG')^{-1} \\ &= G'^{-1}G^{-1} \end{aligned}$$

ce qui ne nécessite que le calcul de $T = G^{-1}$ et de former le triangle inférieur du produit $T'T$. L'inverse T d'une matrice triangulaire inférieure G est défini par

$$t_{ij} = \begin{cases} 1/g_{ii} & i = j \\ -\left(\sum_{k=j}^{i-1} g_{ik}\, t_{kj}\right)/g_{ii} & i \neq j \end{cases}$$

et nécessite $\frac{n^3}{3} + \frac{3}{2}n^2 - \frac{5}{6}n$ opérations élémentaires. Il est aussi possible de calculer la matrice S sans devoir inverser la matrice G (cf. Björck (1996, p. 119) et Lawson and Hanson (1974, p. 67–73)).

Si l'on n'a besoin que des éléments diagonaux de S, il suffit de calculer le carré de la norme Euclidienne des vecteurs de T, soit

$$s_{ii} = \sum_{j=i}^{n} t_{ij}^2 \qquad i = 1, 2, \ldots, n.$$

Les étapes de la résolution sont résumées dans l'algorithme qui suit :

$C = A'A$ (Calculer seulement la portion triangulaire inférieure)

Former $\overline{C} = \begin{bmatrix} C & A'b \\ b'A & b'b \end{bmatrix}$

Calculer $\overline{G} = \begin{bmatrix} G & 0 \\ z' & \rho \end{bmatrix}$

`solve` $G'x = z$ (système triangulaire)

$\sigma^2 = \rho^2/(m-n)$

Calculer $T = G^{-1}$ (inversion matrice triangulaire)

$S = \sigma^2 T'T$ (matrice des variances et covariances)

L'algorithme nécessite $\times\times$ opérations élémentaires.

La méthode des équations normales est séduisante car elle repose sur des procédures très simples (factorisation de Cholesky, produit matriciel, forward and backward substitution). D'autre part la matrice initiale de dimension $m \times n$ est comprimée en une matrice $n \times n$ ce qui peut être un avantage certain dans des situations avec $m \gg n$.

Exemple 13.1 Soit le problème $Ax \cong b$. Alors la solution des moindres carrés par la méthode des équations normales avec factorisation de Cholesky est :

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \overline{C} = \begin{bmatrix} 4 & 10 & 5 \\ 10 & 30 & 11 \\ 5 & 11 & 7 \end{bmatrix} \quad \overline{G} = \begin{bmatrix} 2.0 & 0 & 0 \\ 5.0 & 2.236 & 0 \\ 2.5 & -.670 & 0.547 \end{bmatrix}$$

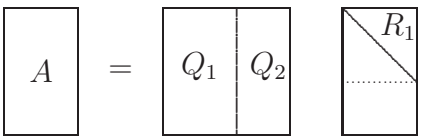
$$\texttt{solve}: \; G'x = z \qquad x = \begin{bmatrix} 2.0 \\ -0.3 \end{bmatrix} \qquad \sigma^2 = .547^2/2 = 0.15$$

$$T = G^{-1} = \begin{bmatrix} 0.500 & 0 \\ -1.118 & 0.447 \end{bmatrix} \qquad S = \sigma^2 T'T = \begin{bmatrix} 0.225 & -.075 \\ -.075 & 0.030 \end{bmatrix}$$

13.1.2 Solution à partir de la factorisation QR

En appliquant une transformation orthogonale au vecteur des résidus on ne modifie pas la norme Euclidienne du vecteur et la minimisation conduira aux mêmes résultats.

Pour la discussion qui suit il convient de partitioner les matrices Q et R comme suit :



Soit alors la matrice $A \in \mathbb{R}^{m \times n}$ et sa factorisation $A = QR$. Le problème est de minimiser

$$\| Ax - b \|_2^2 \quad .$$

On remplace A par sa factorisation QR et on transforme le vecteur $Ax - b$ avec Q' sans que ceci modifie sa norme

$$\| \underbrace{Q'Q}_{I} Rx - Q'b \|_2^2 \; .$$

On a alors

$$\left\| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} Q_1' \\ Q_2' \end{bmatrix} b \right\|_2^2$$

que l'on peut également écrire

$$\| R_1 x - Q_1' b \|_2^2 + \| Q_2' b \|_2^2$$

Du système triangulaire $R_1 x = Q_1' b$ on obtient la solution pour x et la somme des résidus au carré $g(x)$ correspond à $\| Q_2' b \|_2^2$.

Dans la pratique on ne conservera pas la matrice Q mais on calcule ses colonnes au fur et à mesure que l'on en a besoin.

D'autre part on a $A'A = R'Q'QR = R'R$ étant donné que Q est orthogonale et pour le calcul de l'inverse de $A'A$ on procédera comme on l'a indiqué précédemment dans le cas de la factorisation de Cholesky.

Exemple 13.2 Soit le problème $Ax \cong b$ avec A et b donnés à l'exemple 13.1. Alors la solution des moindres carrés par la factorisaton QR est :

$$Q_1 = \begin{bmatrix} -.500 & .670 \\ -.500 & .223 \\ -.500 & -.223 \\ -.500 & -.670 \end{bmatrix} \quad Q_2 = \begin{bmatrix} .023 & .547 \\ -.439 & -.712 \\ .807 & -.217 \\ -.392 & .382 \end{bmatrix} \quad R_1 = \begin{bmatrix} -2.000 & -5.000 \\ 0 & -2.236 \end{bmatrix}$$

$$Q_1' b = c = \begin{bmatrix} -2.500 \\ .670 \end{bmatrix} \quad \texttt{solve}: \; R_1 x = c \quad x = \begin{bmatrix} 2.0 \\ -.3 \end{bmatrix}$$

$$Q_2' b = d = \begin{bmatrix} 0.023 \\ .547 \end{bmatrix} \qquad \text{et} \qquad \sigma^2 = d'd/2 = 0.15$$

13.1.3 Décomposition à valeurs singulières SVD

La décomposition SVD est un outil particulièrement approprié pour la résolution du problème des moindres carrés. Comme précédemment avec la factorisation QR, on exploite le fait que les transformations orthogonales ne modifient pas la norme Euclidienne d'un vecteur.

Étant donné $Ax \cong b$, avec $A \in \mathbb{R}^{m \times n}$ et rang de A égal à $r \leq p = \min(m, n)$, la solution du problème des moindres carrés peut s'écrire

$$x = \underbrace{V \begin{bmatrix} \Sigma_r^{-1} & 0 \\ 0 & 0 \end{bmatrix} U'}_{A^+} b \qquad \text{où bien} \qquad x = \sum_{i=1}^{r} \frac{u_i' b}{\sigma_i} v_i \tag{13.4}$$

où les matrices V, Σ et U sont celles de la décomposition singulière de $A = U\Sigma V'$.

Notons

$$z = V'x = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \qquad \text{et} \qquad c = U'b = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

avec z_1 et $c_1 \in \mathbb{R}^r$. Alors considérons la norme Euclidienne du vecteur des résidus et appliquons lui une transformation orthogonale

$$\begin{aligned} \| b - Ax \|_2 &= \| U'(b - AVV'x) \|_2 \\ &= \left\| \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} - \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right\|_2 \\ &= \left\| \begin{matrix} c_1 - \Sigma_r z_1 \\ c_2 \end{matrix} \right\|_2 \end{aligned}$$

Le vecteur x tel que défini par (13.4) minimise la norme Euclidienne des résidus étant donné que $c_1 - \Sigma_r z_1 = 0$. Donc $\texttt{SSR} = \sum_{i=r+1}^{m} (u_i'b)^2$. Remarquons que (13.4) fournit aussi la solution lorsque A n'est pas de rang plein.

A partir de la décomposition à valeurs singulières de la matrice A on déduit que $(A'A)^{-1} = V\Sigma_r^{-2}V'$ où Σ_r est la sous-matrice carré de Σ. Il est alors possible d'expliciter un élément particulier de la matrice $S = (A'A)^{-1}$ comme

$$s_{ij} = \sum_{k=1}^{n} \frac{v_{ik}v_{jk}}{\sigma_k^2}.$$

Exemple 13.3 Soit le problème $Ax \cong b$ avec A et b donnés à l'exemple 13.1. Alors la solution des moindres carrés par la factorisaton SVD est :

$$U = \begin{bmatrix} 0.219 & -.807 & .023 & 0.547 \\ 0.383 & -.391 & -.439 & -.712 \\ 0.547 & .024 & .807 & -.217 \\ 0.711 & .441 & .392 & 0.382 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 5.779 & 0 \\ 0 & .773 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad V = \begin{bmatrix} .322 & -.946 \\ .946 & .322 \end{bmatrix}$$

$$c = U'b = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \left[\begin{array}{r} 2.080 \\ -1.539 \\ \hline 0.023 \\ 0.547 \end{array}\right] \qquad z_1 = \Sigma_r^{-1} c_1 = \begin{bmatrix} 0.360 \\ -1.990 \end{bmatrix}$$

$$x = Vz_1 = \begin{bmatrix} 2.0 \\ -.3 \end{bmatrix} \qquad \sigma^2 = c_2'c_2/(m-n) = 0.15.$$

13.1.4 Comparaison des méthodes

– Dans la situation ou $\| Ax - b \|_2^2$ est petit et $\kappa_2(A)$ est grand l'approche QR est supérieure à l'approche des équations normales.

– Dans la situation ou $\| Ax - b \|_2^2$ est grand et $\kappa_2(A)$ est grand les deux approches éprouvent des difficultés comparables.

– $\kappa_2(A'A) = \Big(\kappa_2(A)\Big)^2$!!!

– Dans une situation avec $m \gg n$ l'approche des équations normales nécessite a peu près la moitié moins d'opérations élémentaires et de place mémoire.

Bien qu'il ne soit pas possible de désigner un algorithme qui soit préférable dans toutes les situation, la factorisation QR est a considérer comme la méthode standard pour ce type de problème (avec Matlab on obtient cette solution avec la commande `x = A \ b`).

SVD produit les résultats qui sont les plus stables numériquement. La méthode a cependant un coût qui est plus élevé.

13.2 Moindres carrés non-linéaires

Dans le cas des moindres carrés non-linéaires il s'agit de minimiser la fonction

$$g(x) = \frac{1}{2} r(x)'r(x) = \frac{1}{2} \sum_{i=1}^{m} r_i(x)^2 \tag{13.5}$$

avec

$$r_i(x) = y_i - f(t_i, x) \qquad i = 1, \ldots, m$$

où $f(t_i, x)$ est une fonction non-linéaire (le modèle) avec t_i les variables indépendantes et $x \in \mathbb{R}^n$ le vecteur de paramètres à estimer.

Afin d'écrire le modèle quadratique pour la minimisation de (13.5) nous avons besoin des dérivées premières et secondes de $g(x)$. La dérivée première s'écrit

$$\nabla g(x) = \sum_{i=1}^{m} r_i(x) \cdot \nabla r_i(x) = \nabla r(x)'r(x) \tag{13.6}$$

avec

$$\nabla r(x) = \begin{bmatrix} \frac{\partial r_1(x)}{\partial x_1} & \cdots & \frac{\partial r_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial r_m(x)}{\partial x_1} & \cdots & \frac{\partial r_m(x)}{\partial x_n} \end{bmatrix} .$$

la matrice dite Jacobienne. Le vecteur

$$\nabla r_i(x) = \begin{bmatrix} \frac{\partial r_i(x)}{\partial x_1} \\ \vdots \\ \frac{\partial r_i(x)}{\partial x_n} \end{bmatrix}$$

correspond à la ligne i de la matrice Jacobienne.

La dérivée seconde s'écrit

$$\begin{aligned} \nabla^2 g(x) &= \sum_{i=1}^{m} \Big(\nabla r_i(x) \cdot \nabla r_i(x)' + r_i(x) \cdot \nabla^2 r_i(x) \Big) \\ &= \nabla r(x)' \nabla r(x) + S(x) \end{aligned} \tag{13.7}$$

avec $S(x) = \sum_{i=1}^{m} r_i(x) \nabla^2 r_i(x)$.

Exemple 13.4 Soit le modèle non-linéaire $f(t,x) = x_1 e^{x_2 t}$ et les données suivantes :

t	0.0	1.0	2.0	3.0
y	2.0	0.7	0.3	0.1

.

On a le vecteur des résidus

$$r(x) = \begin{bmatrix} y_1 - x_1 e^{x_2 t_1} \\ y_2 - x_1 e^{x_2 t_2} \\ y_3 - x_1 e^{x_2 t_3} \\ y_4 - x_1 e^{x_2 t_4} \end{bmatrix}$$

et la matrice Jacobienne

$$\nabla r(x) = \begin{bmatrix} \frac{\partial r_1(x)}{\partial x_1} & \frac{\partial r_1(x)}{\partial x_2} \\ \frac{\partial r_2(x)}{\partial x_1} & \frac{\partial r_2(x)}{\partial x_2} \\ \frac{\partial r_3(x)}{\partial x_1} & \frac{\partial r_3(x)}{\partial x_2} \\ \frac{\partial r_4(x)}{\partial x_1} & \frac{\partial r_4(x)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -e^{x_2 t_1} & -x_1 t_1 e^{x_2 t_1} \\ -e^{x_2 t_2} & -x_1 t_2 e^{x_2 t_2} \\ -e^{x_2 t_3} & -x_1 t_3 e^{x_2 t_3} \\ -e^{x_2 t_4} & -x_1 t_4 e^{x_2 t_4} \end{bmatrix} .$$

Les dérivées premières sont :

$$\nabla g(x) = \begin{bmatrix} \frac{\partial g(x)}{\partial x_1} \\ \frac{\partial g(x)}{\partial x_2} \end{bmatrix} = \nabla r(x)' r(x) = \begin{bmatrix} -\sum_{i=1}^4 r_i(x) e^{x_2 t_i} \\ -\sum_{i=1}^4 r_i(x) x_1 t_i e^{x_2 t_i} \end{bmatrix}$$

On a les m matrices de dérivées secondes

$$\nabla^2 r_i(x) = \begin{bmatrix} \frac{\partial^2 r_i(x)}{\partial x_1^2} & \frac{\partial^2 r_i(x)}{\partial x_1 \partial x_2} \\ \frac{\partial^2 r_i(x)}{\partial x_1 \partial x_2} & \frac{\partial^2 r_i(x)}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 0 & -t_i e^{x_2 t_i} \\ -t_i e^{x_2 t_i} & -x_1 t_i^2 e^{x_2 t_i} \end{bmatrix}$$

et la matrice des dérivées secondes de $g(x)$ est

$$\nabla^2 g(x) = \begin{bmatrix} \sum_{i=1}^4 (e^{x_2 t_i})^2 & \sum_{i=1}^4 x_1 t_i (e^{x_2 t_i})^2 \\ \sum_{i=1}^4 x_1 (t_i e^{x_2 t_i})^2 & \sum_{i=1}^4 (x_1 t_i e^{x_2 t_i})^2 \end{bmatrix} - \sum_{i=1}^4 r_i(x) \begin{bmatrix} 0 & t_i e^{x_2 t_i} \\ t_i e^{x_2 t_i} & x_1 t_i^2 e^{x_2 t_i} \end{bmatrix} .$$

Considérons $m_c(x)$, l'approximation quadratique de $g(x)$ dans un voisinage de x_c

$$m_c(x) = g(x_c) + \nabla g(x_c)'(x - x_c) + \frac{1}{2}(x - x_c)' \nabla^2 g(x_c)(x - x_c)$$

et cherchons le point $x_+ = x_c + s_{\mathrm{N}}$ pour lequel $\nabla m_c(x_+) = 0$ ce qui est la condition du premier ordre nécessaire pour que x_+ minimise m_c. On a

$$\nabla m_c(x_+) = \nabla g(x_c) + \nabla^2 g(x_c) \underbrace{(x_+ - x_c)}_{s_{\mathrm{N}}} = 0$$

où s_{N} est le pas de Newton que l'on determine en résolvant le système linéaire

$$\nabla^2 g(x_c)\, s_{\mathrm{N}} = -\nabla g(x_c)\ .$$

Dès lors on pourrait envisager la solution du problème (13.5) avec la méthode de Newton dont l'itération k est définie comme

$$\begin{aligned} \texttt{Solve} \quad \nabla^2 g(x^{(k)})\, s_{\mathrm{N}}^{(k)} &= -\nabla g(x^{(k)}) \\ x^{(k+1)} &= x^{(k)} + s_{\mathrm{N}}^{(k)} \end{aligned}$$

Dans la pratique on procède cependant différemment étant donné que l'évaluation de $S(x)$ dans (13.7) peut s'avérer très difficile, voire impossible.

Les méthodes qui seront présentées ci-après se différencient par la façon dont ils approchent la matrice des dérivées secondes $\nabla^2 g(x)$.

13.2.1 Méthode de Gauss-Newton

Cette méthode utilise une approximation de la matrice des dérivées secondes (13.7) en omettant le terme $S(x)$. Comme $S(x)$ est composé d'une somme de termes $r_i(x)\nabla^2 r_i(x)$ cette simplification se justifie dans une situation ou les résidus $r_i(x)$ sont petits. C'est une situation ou le modèle épouse bien les données. La méthode de Gauss-Newton se résume dans l'algorithme qui suit.

Algorithme 23 Méthode de Gauss-Newton pour les moindres carrés non-linéaires

1: Choisir $x^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until convergence` **do**
3: `Calculer` $\nabla r(x^{(k)})$
4: `Solve` $\left(\nabla r(x^{(k)})'\nabla r(x^{(k)})\right) s_{\mathrm{GN}}^{(k)} = -\nabla r(x^{(k)})' r(x^{(k)})$
5: `Update` $x^{(k+1)} = x^{(k)} + s_{\mathrm{GN}}^{(k)}$
6: **end for**

On remarque que le système linéaire qui définit le pas de Gauss-Newton $s_{\mathrm{GN}}^{(k)}$ est un système d'équations normales et que le pas est aussi la solution d'un problème des moindres carrés. En accord à ce qui a été dit précédemment au sujet des moindres carrés linéaires on préféra la solution du système surdéterminé

$$\nabla r(x^{(k)})\, s_{\mathrm{GN}}^{(k)} \approx -r(x^{(k)})$$

obtenu à partir de la factorisation QR à la solution des équations normales.

L'algorithme peut ne pas converger si le point de départ est choisi trop loin de la solution.

Lorsque les résidus sont grands au point de la solution, l'approximation de la matrice des dérivées secondes peut s'avérer insuffisante, avec comme conséquence soit une convergence très lente ou pas de convergence du tout.

13.2.2 Méthode de Levenberg-Marquardt

Lorsque la méthode de Gauss-Newton échoue, notamment lorsque la solution du sous-problème de la solution du système linéaire n'est pas de rang complet, la méthode de Levenberg-Marquardt constitue une alternative intéressante. Dans cette méthode on approche la matrice $S(x)$ par une matrice diagonale μI. Nous avons alors l'algorithme suivant :

Algorithme 24 Méthode de Levenberg-Marquardt pour les moindres carrés non-linéaires

1: Choisir $x^{(0)}$
2: **for** $k = 0, 1, 2, \ldots$ `until convergence` **do**
3: `Calculer` $\nabla r(x^{(k)})$ et μ_k
4: `Solve` $\left(\nabla r(x^{(k)})'\nabla r(x^{(k)}) + \mu_k I\right) s_{\text{LM}}^{(k)} = -\nabla r(x)'r(x)$
5: `Update` $x^{(k+1)} = x^{(k)} + s_{\text{LM}}^{(k)}$
6: **end for**

Dans ce cas la solution du pas $s_{LM}^{(k)}$ est définie à partir du problème des moindres carrés linéaires

$$\begin{bmatrix} \nabla r(x^{(k)}) \\ \mu_k^{1/2} I \end{bmatrix} s_{\text{LM}}^{(k)} \approx - \begin{bmatrix} r(x^{(k)}) \\ 0 \end{bmatrix}$$

que l'on résoudra en utilisant la factorisation QR. Ainsi on évite le calcul du produit $\nabla r(x^{(k)})'\nabla r(x^{(k)})$ qui peut introduire des instabilités numériques.

μ_k est choisi à partir des considérations sur la "trust region". Dans la pratique on choisit la valeur 10^{-2}.

Suivant le choix de μ_k, $0 \leq \mu_k < \infty$ le pas de Levenberg-Marquardt se situe entre le pas de Gauss-Newton pour $\mu_k = 0$ et un pas "steepest descent".

Si μ_k est choisi judicieusement la méthode de Levenberg-Marquardt s'avère très robuste en pratique. Elle constitue l'algorithme de base dans un grand nombre de logiciels spécialisés.

L'algorithme peut ne pas converger si le point de départ est choisi trop loin de la solution.

Lorsque les résidus sont grands au point de la solution, l'approximation de la matrice des dérivées secondes peut s'avérer insuffisante, avec comme conséquence soit une convergence très lente ou pas de convergence du tout.

Bibliographie

Björck, A. (1996). *Numerical Methods for Least Squares Problems*. SIAM. Philadelphia, PA.

Dulmage, A.L. and N.S. Mendelsohn (1963). Two algorithms for bipartite graphs. *SIAM* **7**, 183–194.

Gilli, M. (1995). Graph-Theory Based Tools in the Practice of Macroeconometric Modeling. In : *Methods and Applications of Economic Dynamics* (S. K. Kuipers, L. Schoonbeek and E. Sterken, Ed.). Contributions to Economic Analysis. North Holland. Amsterdam.

Gilli, M. and M. Garbely (1996). Matching, Covers, and Jacobian Matrices. *Journal of Economic Dynamics and Control* **20**, 1541–1556.

Golub, G. H. and C. F. Van Loan (1989). *Matrix Computations*. Johns Hopkins. Baltimore.

Lawson, C. L. and R. J. Hanson (1974). *Solving Least Squares Problems*. Prentice-Hall. Englewood Cliffs, NJ.

Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling (1986). *Numerical Recipes*. Cambridge University Press. Cambridge.

Index

élimination
    de Gauss, 31, 32, 35
    de Gauss-Jordan, 45

algorithme
    `Broyden`, 99
    `Gauss-Newton`, 122
    `Gauss-Seidel`, 69
    `Jacobi`, 69
    `Levenberg-Marquardt`, 123
    `Newton`, 96
    `SOR`, 74
    `bsub`, 28
    `cholesky`, 53
    `egpp`, 42
    `eg`, 37
    `fsub`, 28
    `ldl`, 51
    `ldm`, 49
    `lu3diag`, 55
    `perm`, 40
    `rsl`, 43
    `tgm1`, 34
    `fsub1`, 43

back substitution, 28
base, 2

catastophic cancellation, 10
chopping, 4
classification
    des algorithmes, 20
complexité
    d'un algorithme, 18
    non-polynômiale, 20
    polynomiale, 20
condition
    d'un problème, 11
    d'une matrice, 12, 13
convergence
    linéaire, 97
    quadratique, 97
    super-linéaire, 97
    taux de, 97

digits significatifs, 8

eps, 6
erreur
    absolue, 10
    d'arrondi, 5
    de troncature, 1
    résiduelle, 14
    relative, 9
exposant, 2

factorisation
    de Cholesky, 52
    LDL', 50
    LDM', 47, 48
    LU, 31
flops, 20
forward substitution, 27

Gauss
    élimination de, 32
    matrice d'élimination de, 33
    multiplicateur de, 33
Gauss-Jordan, élimination de, 45
Gauss-Seidel
    non-linéaire, 92

Inf, 4
instabilité numérique, 11
itération
    de Gauss-Seidel, 68
    de Jacobi, 68

Jacobi

non-linéaire, 92

mantisse, 2
normalisation, 2, 3
Matlab code
`FPI`, 88
`bracketing`, 87
`converged`, 75
matrice
d'échange, 39
d'élimination de Gauss, 33
définie positive, 51
de Hessenberg, 24
de permutation, 38
de Töplitz, 24
par bande, 55
transformation de Gauss, 33
triangulaire unitaire, 28
tridiagonale, 55
multiplicateur de Gauss, 33

Newton, 95
amorti, 100
convergence, 98
Newton-Raphson, 95

offset, 4
opérations élémentaires, 20
ordre d'une fonction, 19
overflow, 4

perfect rounding, 4
pivot, 33, 35
pivotage
complet, 41
partiel, 41, 42
point fixe, 93
précision machine, 5, 6
problème mal conditionné, 11, 12

Quasi-Newton, 99

rang structurel, 61
rayon spectral, 72
roundoff error, 9

SOR, 73
paramètre de relaxation, 73
sur-relaxation successive, 73
système
triangulaire inférieur, 27
triangulaire supérieur, 28

underflow, 4

virgule flottante, 1–3, 8
ensemble des nombres en, 3
opérateur, 4