Eric Walter

# Méthodes Numériques & Optimisation

## un guide du consommateur

*Fluctuant Nec Merguntur*

Éric Walter

# Méthodes numériques et optimisation

## un guide du consommateur

Fluctuant Nec Merguntur

Éric Walter
Laboratoire des Signaux et Systèmes
CNRS – CentraleSupélec – Université Paris-Sud
91192 Gif-sur-Yvette
France

Merci d'envoyer vos commentaires et questions à
eric.p.h.walter@gmail.com



Fluctuant Nec Merguntur, Paris, 2015

Traduction abrégée de l'édition en langue anglaise :
*Numerical Methods and Optimization, A Consumer Guide* d'Éric Walter

*À mes petits-enfants*

# Table des matières

# Chapitre 1
# Du calcul exact à l'évaluation numérique approchée

Les cours du lycée nous ont conduits à voir la résolution de problèmes en physique et en chimie comme l'élaboration de *solutions formelles explicites* en fonction de paramètres inconnus puis l'utilisation de ces solutions dans des *applications numériques* pour des valeurs numériques spécifiques de ces paramètres. Cette démarche limitait la classe des problèmes solubles à un ensemble très limité de problèmes suffisamment simples pour qu'une telle solution explicite existe.

Malheureusement, la plupart des problèmes réels en sciences pures et appliquées ne peuvent bénéficier d'une telle solution mathématique explicite. On est donc conduit à remplacer le calcul exact de celle-ci par une évaluation numérique approchée. Ceci est particulièrement évident en ingénierie, où la conception assistée par ordinateur sur la base de simulations numériques est la règle.

Ce livre porte sur le calcul numérique sur ordinateur, et ne dit presque rien du calcul formel sur ordinateur (*computer algebra*), bien qu'ils se complètent utilement. L'utilisation d'approximations à virgule flottante pour les nombres réels a pour conséquence que des opérations approximatives sont effectuées sur des nombres approximatifs. Pour se protéger contre le risque de désastre numérique, on doit alors choisir des méthodes qui rendent les erreurs finales aussi petites que possible. Il se trouve que nombre des méthodes de résolution des problèmes mathématiques élémentaires apprises au lycée ou dans les premières années des études supérieures conviennent mal au calcul à virgule flottante et doivent donc être abandonnées.

Remplaçant le *calcul exact* par une *évaluation numérique approchée*, nous tenterons de

— découvrir comment échapper à la dictature de cas particuliers suffisamment simples pour recevoir une solution analytique explicite, et gagner ainsi le pouvoir de résoudre des problèmes complexes posés par de vraies applications,
— comprendre les principes à la base de méthodes reconnues et utilisées dans des logiciels numériques à l'état de l'art,
— apprendre les avantages et les limites de ces méthodes, pour pouvoir choisir à bon escient les briques *pré-existantes* à assembler pour résoudre un problème donné.

La présentation est à un niveau introductif, bien loin du niveau de détails requis pour implémenter ces méthodes de façon efficace. Notre but principal est d'aider le lecteur à devenir un meilleur *consommateur* de méthodes numériques, capable de faire son choix parmi celles qui sont disponibles pour une tâche donnée, comprenant ce qu'elles peuvent et ne peuvent pas faire et outillé pour conduire une évaluation critique de la validité de leurs résultats.

Suggérons au passage de résister au désir d'écrire toutes les lignes du code qu'on compte utiliser. Tant de temps et d'efforts ont été consacrés à polir le code mettant en œuvre les méthodes numériques de base que la probabilité de faire mieux est très faible. Le codage devrait donc se limiter à ce qui ne peut être évité ou à ce dont on peut attendre une amélioration par rapport à l'état de l'art des logiciels facilement disponibles (c'est beaucoup demander). Ceci libérera du temps pour une réflexion plus large :

— quel est le *vrai* problème que je veux résoudre ? (Comme le dit Richard Hamming [94] : *Le calcul sur ordinateur est, ou du moins devrait être, intimement lié avec à la fois la source du problème et l'usage qui sera fait des réponses— ce n'est pas une étape à considérer de façon isolée*.)
— comment puis-je mettre ce problème sous forme mathématique *sans trahir son sens* ?
— comment décomposer le problème mathématique résultant en tâches bien définies et numériquement réalisables ?
— quels sont les avantages et les limites des méthodes numériques facilement disponibles pour ces tâches ?
— faut-il faire un choix parmi ces méthodes, ou trouver une stratégie alternative ?
— quelle est la façon la plus efficace d'utiliser mes ressources (temps, ordinateurs, librairies de programmes, etc.) ?
— comment puis-je évaluer la qualité de mes résultats ?
— quelles mesures dois-je prendre s'il s'avère que mes choix n'ont pas conduit à une solution satisfaisante du problème initial ?

Plusieurs livres légitimement populaires sur les algorithmes numériques contiennent *Numerical Recipes* dans leur titre [186]. Filant cette métaphore culinaire, on peut dire qu'on obtiendra un repas bien plus sophistiqué en choisissant les plats qui conviennent dans un menu de programmes scientifiques facilement disponibles qu'en bricolant l'équivalent d'un sandwich à la dinde avec mayonnaise dans sa cuisine numérique.

Décider de ne pas coder des algorithmes pour lesquels des programmes de niveau professionnel sont disponibles n'impose pas de les traiter comme des boîtes noires magiques, et c'est pourquoi nous expliquerons les idées à la base des principales méthodes disponibles pour résoudre une classe de problème donnée.

Le niveau des connaissances mathématiques nécessaires pour la lecture de ce qui suit est une compréhension de l'algèbre linéaire comme on l'enseigne lors des premières années d'un cursus universitaire. J'espère que ceux qui haïssent les mathématiques trouveront ici des raisons pour reconsidérer leur position en voyant combien elles se révèlent utiles pour la résolution de problèmes de la vie réelle,

et que ceux qui les aiment me pardonneront des simplifications audacieuses et découvriront des aspects pratiques fascinants des mathématiques en action.

Nos ingrédients principaux seront inspirés par une *cuisine bourgeoise* classique, avec quelques mots au sujet de recettes qu'il vaut mieux éviter et un zeste de *nouvelle cuisine*.

## 1.1 Pourquoi ne pas utiliser des méthodes mathématiques naïves ?

Il y a au moins trois raisons pour lesquelles une méthode naïve apprise au lycée où en premier cycle universitaire peut ne pas convenir.

### *1.1.1 Trop de calcul*

Considérons le cas (pas si fréquent) d'un problème pour lequel un algorithme est disponible qui donnerait une solution exacte en un nombre fini de pas si toutes les opérations nécessaires étaient effectuées de façon exacte. Une première raison pour laquelle un tel *algorithme fini exact* peut ne pas convenir est quand il nécessite beaucoup plus d'opérations qu'il n'est nécessaire.

*Exemple 1.1. Évaluation de déterminants*
Évaluer le déterminant d'une matrice $\mathbf{A}$ pleine $n \times n$ par développement des cofacteurs nécessite plus de $n!$ opérations flottantes (ou flops), alors que des méthodes à base d'une factorisation de $\mathbf{A}$ le font en environ $n^3$ flops. Pour $n = 100$, par exemple, $n!$ est un peu moins de $10^{158}$, quand on estime que le nombre d'atomes dans l'univers observable est inférieur à $10^{81}$, et quand $n^3 = 10^6$. □

### *1.1.2 Trop sensible aux erreurs numériques*

Développées sans tenir compte des effets des arrondis, les méthodes classiques pour la résolution de problèmes numériques peuvent donner des résultats complètement erronés lors de calculs à virgule flottante.

*Exemple 1.2. Évaluation des racines d'une équation polynomiale du second degré*
Soient à évaluer les solutions $x_1$ et $x_2$ de l'équation

$$ax^2 + bx + c = 0, \tag{1.1}$$

avec $a$, $b$ et $c$ des nombres à virgule flottante tels que $x_1$ et $x_2$ soient réels. Nous avons appris au lycée que

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad \text{et} \quad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}. \tag{1.2}$$

Ceci est un exemple d'*algorithme vérifiable*, puisqu'il suffit de vérifier que le polynôme vaut zéro en $x_1$ ou $x_2$ pour s'assurer que $x_1$ ou $x_2$ est une solution.

Cet algorithme convient pourvu qu'il n'implique pas le calcul de la différence de nombres à virgule flottante proches, comme ce serait le cas si $|4ac|$ était trop petit par rapport à $b^2$. Une telle différence peut en effet se révéler *numériquement désastreuse*, et doit donc être évitée. On peut utiliser à cette fin l'algorithme suivant, qui est aussi vérifiable et exploite le fait que $x_1 x_2 = c/a$ :

$$q = \frac{-b-\text{signe}(b)\sqrt{b^2-4ac}}{2}, \tag{1.3}$$

$$x_1 = \frac{q}{a}, \quad x_2 = \frac{c}{q}. \tag{1.4}$$

Bien que ces deux algorithmes soient mathématiquement équivalents, le second est beaucoup plus robuste que le premier vis à vis des erreurs induites par les opérations à virgule flottante (voir la section 14.7 pour une comparaison numérique). Ceci ne résout pas, cependant, le problème qui apparaît quand $x_1$ et $x_2$ tendent l'un vers l'autre, puisque $b^2 - 4ac$ tend alors vers zéro. □

Nous rencontrerons de nombreuses situations similaires, où des algorithmes naïfs doivent être remplacés par des variantes plus robustes ou moins coûteuses.

### *1.1.3 Pas disponible*

Très souvent, il n'existe pas de méthode mathématique pour trouver la solution numérique exacte du problème considéré. Tel sera le cas, par exemple, pour la plupart des problèmes de simulation ou d'optimisation, ainsi que pour la résolution de la plupart des systèmes d'équations non linéaires.

## 1.2 Que faire, alors ?

Les mathématiques ne doivent pas être abandonnées en chemin, car elles jouent un rôle central dans l'élaboration d'algorithmes numériques efficaces. Il devient possible de trouver des solutions approchées d'une précision étonnante pourvu que le caractère spécifique du calcul en virgule flottante soit pris en compte.

## 1.3 Comment est organisé ce livre ?

Nous aborderons les problèmes les plus simples en premier, avant de considérer des problèmes plus ambitieux en nous appuyant sur ce qui aura déjà été décrit. L'ordre de présentation est comme suit :

— notations et notions de base,
— algorithmes pour l'algèbre linéaire (résolution de systèmes d'équations linéaires, inversion de matrices, calcul de déterminants, de valeurs propres et de vecteurs propres),
— interpolation et extrapolation,
— intégration et différentiation,
— résolution de systèmes d'équations non linéaires,
— optimisation sans contrainte,
— optimisation sous contraintes,
— optimisation combinatoire,
— résolution d'équations différentielles ordinaires,
— résolution d'équations aux dérivées partielles,
— évaluation de la précision de résultats numériques.

Cette classification n'est pas étanche. Ce peut être une bonne idée de transformer un problème en un autre. En voici quelques exemples :

— pour trouver les racines d'une équation polynomiale, on peut chercher les valeurs propres d'une matrice, comme dans l'exemple 4.3 ;
— pour évaluer une intégrale définie, on peut résoudre une équation différentielle ordinaire, comme dans la section 6.2.4 ;
— pour résoudre un système d'équations, on peut minimiser une norme de la différence entre ses deux membres, comme dans l'exemple 9.8 ;
— pour résoudre un problème d'optimisation sans contrainte, on peut introduire de nouvelles variables et des contraintes, comme dans l'exemple 10.7.

La plupart des méthodes numériques présentées sont des ingrédients importants de codes numériques de niveau professionnel. Des exceptions ont été faites pour

— des méthodes fondées sur les idées qui viennent facilement à l'esprit mais qui sont en fait si mauvaises qu'il convient de les dénoncer, comme dans l'exemple 1.1,
— des prototypes de méthodes qui peuvent aider à comprendre des approches plus sophistiquées, comme quand des problèmes à une variable sont considérés avant le cas à plusieurs variables,
— des méthodes prometteuses principalement fournies par des institutions de recherche académique, comme des méthodes d'optimisation et de simulation garanties.

MATLAB est utilisé pour montrer, à travers des exemples simples mais pas forcément triviaux composés en `typewriter`, à quel point il est facile d'utiliser des méthodes classiques. Il serait dangereux, cependant, de tirer des conclusions définitives sur les mérites de ces méthodes sur la seule base de ces exemples particuliers. Le lecteur est invité à consulter la documentation MATLAB pour plus de détails sur les fonctions disponibles et leurs arguments optionnels. Des informa-

tions complémentaires, y compris des exemples éclairants peuvent être trouvés dans [158], avec un matériel auxiliaire disponible sur le WEB, et dans [6]. Bien que MATLAB soit le seul langage de programmation utilisé dans ce livre, il ne convient pas pour résoudre tous les problèmes numériques dans tous les contextes. Des solutions alternatives sont décrites au chapitre 15.

Ce livre se termine par un chapitre sur des ressources WEB utilisables pour aller plus loin.

Cet ouvrage a été composé avec TeXmacs avant d'être exporté vers LaTeX. Merci beaucoup à Joris van der Hoeven et à ses collaborateurs pour ce remarquable logiciel WYSIWYG, gratuitement disponible à www.texmacs.org.

# Chapitre 2
# Notations et normes

## 2.1 Introduction

Ce chapitre rappelle les conventions usuelles pour distinguer les variables scalaires, vectorielles et matricielles. La notation de Vetter pour les dérivations matricielles est ensuite expliquée, ainsi que le sens des expressions *petit o* et *grand O* employées pour comparer les comportements locaux ou asymptotiques de fonctions. Les principales normes de vecteurs et de matrices sont finalement décrites. Les normes trouvent une première application dans la définition de types de vitesses de convergence pour des algorithmes itératifs.

## 2.2 Scalaires, vecteurs et matrices

Sauf indication contraire, les variables scalaires sont à valeur réelle, ainsi que les éléments des vecteurs et des matrices.

Les *variables scalaires* sont en italique ($v$ ou $V$), les *vecteurs colonne* sont représentés par des minuscules grasses ($\mathbf{v}$), et les matrices par des majuscules grasses ($\mathbf{M}$). La *transposition*, qui transforme les colonnes en lignes dans un vecteur ou une matrice, est notée par l'exposant $^{\mathrm{T}}$, et s'applique à ce qui le *précède*. Ainsi, $\mathbf{v}^{\mathrm{T}}$ est un *vecteur ligne* et dans $\mathbf{A}^{\mathrm{T}}\mathbf{B}$ c'est $\mathbf{A}$ qui est transposée, pas $\mathbf{B}$.

La *matrice identité* est $\mathbf{I}$, avec $\mathbf{I}_n$ la matrice identité de dimensions $n \times n$. Le $i$-ème vecteur colonne de $\mathbf{I}$ est le *vecteur canonique* $\mathbf{e}^i$.

L'élément à l'intersection de la $i$-ème ligne et de la $j$-ème colonne de $\mathbf{M}$ est $m_{i,j}$. Le produit de matrices

$$\mathbf{C} = \mathbf{AB} \tag{2.1}$$

implique ainsi que

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}, \tag{2.2}$$

et le nombre des colonnes de $\mathbf{A}$ doit être égal au nombre des lignes de $\mathbf{B}$. Rappelons que le produit de matrices (ou de vecteurs) *n'est pas commutatif*, en général. Ainsi, par exemple, quand $\mathbf{v}$ et $\mathbf{w}$ sont des vecteurs colonne de même dimension, $\mathbf{v}^{\mathrm{T}}\mathbf{w}$ est un scalaire, tandis que $\mathbf{w}\mathbf{v}^{\mathrm{T}}$ est une matrice (de rang un).

Il est utile de retenir que

$$(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}, \tag{2.3}$$

et que, pourvu que $\mathbf{A}$ et $\mathbf{B}$ soient inversibles,

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}. \tag{2.4}$$

Si $\mathbf{M}$ est carrée et symétrique, toutes ses valeurs propres sont réelles. $\mathbf{M} \succ 0$ signifie alors que chacune de ces valeurs propres est strictement positive ($\mathbf{M}$ est *symétrique définie positive*), tandis que $\mathbf{M} \succcurlyeq 0$ signifie qu'elles sont positives ou nulles ($\mathbf{M}$ est *symétrique définie non-négative*).

## 2.3 Dérivées

Pourvu que $f(\cdot)$ soit une fonction suffisamment différentiable de $\mathbb{R}$ vers $\mathbb{R}$, nous poserons

$$\dot{f}(x) = \frac{\mathrm{d}f}{\mathrm{d}x}(x), \tag{2.5}$$

$$\ddot{f}(x) = \frac{\mathrm{d}^2 f}{\mathrm{d}x^2}(x), \tag{2.6}$$

$$f^{(k)}(x) = \frac{\mathrm{d}^k f}{\mathrm{d}x^k}(x). \tag{2.7}$$

La notation de Vetter [235] sera utilisée pour les dérivées de matrices par rapport à des matrices : si $\mathbf{A}$ est de dimensions $n_{\mathrm{A}} \times m_{\mathrm{A}}$ et $\mathbf{B}$ de dimensions $n_{\mathrm{B}} \times m_{\mathrm{B}}$, alors

$$\mathbf{M} = \frac{\partial \mathbf{A}}{\partial \mathbf{B}} \tag{2.8}$$

est de dimensions $n_{\mathrm{A}} n_{\mathrm{B}} \times m_{\mathrm{A}} m_{\mathrm{B}}$, et telle que sa sous-matrice de dimensions $n_{\mathrm{A}} \times m_{\mathrm{A}}$ en position $(i, j)$ est

$$\mathbf{M}_{i,j} = \frac{\partial \mathbf{A}}{\partial b_{i,j}}. \tag{2.9}$$

*Remarque 2.1.* $\mathbf{A}$ et $\mathbf{B}$ dans (2.8) peuvent être des vecteurs ligne ou colonne. □

*Remarque 2.2.* Il existe d'autres notations incompatibles, et il faut être prudent quand on combine des formules de sources différentes. □

*Exemple 2.1.* Si $\mathbf{v}$ est un vecteur colonne générique de $\mathbb{R}^n$, alors

$$\frac{\partial \mathbf{v}}{\partial \mathbf{v}^{\mathrm{T}}} = \frac{\partial \mathbf{v}^{\mathrm{T}}}{\partial \mathbf{v}} = \mathbf{I}_n. \tag{2.10}$$

□

*Exemple 2.2.* Si $J(\cdot)$ est une fonction différentiable de $\mathbb{R}^n$ vers $\mathbb{R}$, et si $\mathbf{x}$ est un vecteur de $\mathbb{R}^n$, alors

$$\frac{\partial J}{\partial \mathbf{x}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_n} \end{bmatrix} (\mathbf{x}) \tag{2.11}$$

est le *vecteur gradient*, ou *gradient*, de $J(\cdot)$ en $\mathbf{x}$. □

*Exemple 2.3.* Si $J(\cdot)$ est une fonction deux fois différentiable de $\mathbb{R}^n$ vers $\mathbb{R}$, et si $\mathbf{x}$ est un vecteur de $\mathbb{R}^n$, alors

$$\frac{\partial^2 J}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 J}{\partial x_1^2} & \frac{\partial^2 J}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 J}{\partial x_1 \partial x_n} \\ \frac{\partial^2 J}{\partial x_2 \partial x_1} & \frac{\partial^2 J}{\partial x_2^2} & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 J}{\partial x_n \partial x_1} & \cdots & \cdots & \frac{\partial^2 J}{\partial x_n^2} \end{bmatrix} (\mathbf{x}) \tag{2.12}$$

est la *matrice hessienne* $(n \times n)$, ou *hessienne*, de $J(\cdot)$ en $\mathbf{x}$. D'après le théorème de Schwarz,

$$\frac{\partial^2 J}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{\partial^2 J}{\partial x_j \partial x_i}(\mathbf{x}), \tag{2.13}$$

pourvu que les deux dérivées soient continues en $\mathbf{x}$ et que $\mathbf{x}$ appartienne à un ensemble ouvert dans lequel elles sont définies toutes les deux. Les hessiennes sont donc symétriques, sauf dans des cas pathologiques non considérés ici. □

*Exemple 2.4.* Si $\mathbf{f}(\cdot)$ est une fonction différentiable de $\mathbb{R}^n$ vers $\mathbb{R}^p$, et si $\mathbf{x}$ est un vecteur de $\mathbb{R}^n$, alors

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_p}{\partial x_1} & \cdots & \cdots & \frac{\partial f_p}{\partial x_n} \end{bmatrix} \tag{2.14}$$

est la *matrice jacobienne* $(p \times n)$, ou *jacobienne*, de $\mathbf{f}(\cdot)$ en $\mathbf{x}$. Quand $p = n$, la jacobienne est carrée et son déterminant est le *jacobien*. □

*Remarque 2.3.* Les trois derniers exemples montrent que la hessienne de $J(\cdot)$ en $\mathbf{x}$ est la jacobienne de sa fonction gradient évaluée en $\mathbf{x}$. □

*Remarque 2.4.* Gradients et hessiennes sont fréquemment utilisés dans le contexte de l'optimisation, et les jacobiennes quand on résout des systèmes d'équations non linéaires. □

*Remarque 2.5.* L'*opérateur Nabla* $\nabla$, un vecteur de dérivées partielles par rapport à toutes les variables de la fonction sur laquelle il opère

$$\nabla = \left( \frac{\partial}{\partial x_1}, \cdots, \frac{\partial}{\partial x_n} \right)^{\mathrm{T}}, \tag{2.15}$$

est souvent utilisé par souci de concision, surtout pour les équations aux dérivées partielles. En appliquant $\nabla$ à une fonction scalaire $J$ et en évaluant le résultat en $\mathbf{x}$, on obtient le gradient

$$\nabla J(\mathbf{x}) = \frac{\partial J}{\partial \mathbf{x}}(\mathbf{x}). \tag{2.16}$$

Si on remplace la fonction scalaire par une fonction vectorielle $\mathbf{f}$, on obtient la jacobienne

$$\nabla \mathbf{f}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}), \tag{2.17}$$

où $\nabla \mathbf{f}$ est interprété comme $\left(\nabla \mathbf{f}^{\mathrm{T}}\right)^{\mathrm{T}}$.

En appliquant $\nabla$ deux fois à une fonction scalaire $J$ et en évaluant le résultat en $\mathbf{x}$, on obtient la hessienne

$$\nabla^2 J(\mathbf{x}) = \frac{\partial^2 J}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}). \tag{2.18}$$

($\nabla^2$ est parfois utilisé pour dénoter l'*opérateur laplacien* $\Delta$, tel que

$$\Delta f(\mathbf{x}) = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}(\mathbf{x}) \tag{2.19}$$

est un scalaire. Le contexte et des considérations de dimensions permettent de clarifier le sens des expressions impliquant Nabla.) □

*Exemple 2.5.* Si $\mathbf{v}$, $\mathbf{M}$ et $\mathbf{Q}$ ne dépendent pas de $\mathbf{x}$ et $\mathbf{Q}$ est symétrique, alors

$$\frac{\partial}{\partial \mathbf{x}} \left( \mathbf{v}^{\mathrm{T}} \mathbf{x} \right) = \mathbf{v}, \tag{2.20}$$

$$\frac{\partial}{\partial \mathbf{x}^{\mathrm{T}}} (\mathbf{M}\mathbf{x}) = \mathbf{M}, \tag{2.21}$$

$$\frac{\partial}{\partial \mathbf{x}} \left( \mathbf{x}^{\mathrm{T}} \mathbf{M} \mathbf{x} \right) = \left( \mathbf{M} + \mathbf{M}^{\mathrm{T}} \right) \mathbf{x} \tag{2.22}$$

et

$$\frac{\partial}{\partial \mathbf{x}} \left( \mathbf{x}^{\mathrm{T}} \mathbf{Q} \mathbf{x} \right) = 2\mathbf{Q}\mathbf{x}. \tag{2.23}$$

Ces formules seront souvent utilisées. □

## 2.4 Petit o et grand O

La fonction $f(x)$ est $o(g(x))$ quand $x$ tend vers $x_0$ si

$$\lim_{x\to x_0} \frac{f(x)}{g(x)} = 0, \tag{2.24}$$

de sorte que $f(x)$ devient négligeable par rapport à $g(x)$ quand $x$ est suffisamment proche de $x_0$. Dans ce qui suit, $x_0$ est toujours pris égal à zéro et nous écrivons juste $f(x) = o(g(x))$.

La fonction $f(x)$ est $O(g(x))$ quand $x$ tend vers $+\infty$ s'il existe des nombres réels $x_0$ et $M$ tels que

$$x > x_0 \Rightarrow |f(x)| \leqslant M|g(x)|. \tag{2.25}$$

La fonction $f(x)$ est $O(g(x))$ quand $x$ tend vers zéro s'il existe des nombres réels $\delta$ et $M$ tels que

$$|x| < \delta \Rightarrow |f(x)| \leqslant M|g(x)|. \tag{2.26}$$

La notation $O(x)$ ou $O(n)$ sera utilisée

- pour des développements de Taylor, où $x$ sera un réel tendant vers zéro,
- pour des analyses de complexité d'algorithmes, où $n$ sera un entier positif tendant vers l'infini.

*Exemple 2.6.* La fonction

$$f(x) = \sum_{i=2}^{m} a_i x^i,$$

avec $m \geqslant 2$, est telle que

$$\lim_{x\to 0} \frac{f(x)}{x} = \lim_{x\to 0} \left( \sum_{i=2}^{m} a_i x^{i-1} \right) = 0,$$

de sorte que $f(x)$ est $o(x)$. Par ailleurs, si $|x| < 1$, alors

$$\frac{|f(x)|}{x^2} < \sum_{i=2}^{m} |a_i|,$$

de sorte que $f(x)$ est $O(x^2)$ quand $x$ tend vers zéro. Par contre, si $x$ est remplacé par un (grand) entier positif $n$, alors

$$f(n) = \sum_{i=2}^{m} a_i n^i \leqslant \sum_{i=2}^{m} |a_i n^i| \leqslant \left( \sum_{i=2}^{m} |a_i| \right) \cdot n^m,$$

de sorte que $f(n)$ est $O(n^m)$ quand $n$ tend vers l'infini. □

## 2.5 Normes

Une fonction $f(\cdot)$ d'un espace vectoriel $\mathbb{V}$ vers $\mathbb{R}$ est une *norme* si elle satisfait les trois propriétés suivantes :

1. $f(\mathbf{v}) \geqslant 0$ pour tout $\mathbf{v} \in \mathbb{V}$ (*positivité*),
2. $f(\alpha\mathbf{v}) = |\alpha| \cdot f(\mathbf{v})$ pour tout $\alpha \in \mathbb{R}$ et $\mathbf{v} \in \mathbb{V}$ (*mise à l'échelle positive*),
3. $f(\mathbf{v}^1 \pm \mathbf{v}^2) \leqslant f(\mathbf{v}^1) + f(\mathbf{v}^2)$ pour tout $\mathbf{v}^1 \in \mathbb{V}$ et $\mathbf{v}^2 \in \mathbb{V}$ (*inégalité triangulaire*).

Ces propriétés impliquent que $f(\mathbf{v}) = 0 \Rightarrow \mathbf{v} = \mathbf{0}$ (*non-dégénérescence*). Une autre relation utile est

$$|f(\mathbf{v}^1) - f(\mathbf{v}^2)| \leqslant f(\mathbf{v}^1 \pm \mathbf{v}^2). \tag{2.27}$$

Les normes sont utilisées pour quantifier les distances entre vecteurs. Elles jouent un rôle essentiel, par exemple, dans la caractérisation de la difficulté intrinsèque de problèmes numériques à travers la notion de conditionnement (voir la section 3.3) ou dans la définition de fonctions de coût pour l'optimisation.

### *2.5.1 Normes vectorielles*

Les normes les plus utilisées dans $\mathbb{R}^n$ sont les normes $l_p$

$$\|\mathbf{v}\|_p = \left( \sum_{i=1}^{n} |v_i|^p \right)^{\frac{1}{p}}, \tag{2.28}$$

avec $p \geqslant 1$. Elles incluent

— la *norme euclidienne* (ou norme $l_2$)

$$||\mathbf{v}||_2 = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{\mathbf{v}^{\mathrm{T}}\mathbf{v}}, \tag{2.29}$$

— la norme $l_1$

$$||\mathbf{v}||_1 = \sum_{i=1}^{n} |v_i|, \tag{2.30}$$

— la norme $l_\infty$

$$||\mathbf{v}||_\infty = \max_{1 \leqslant i \leqslant n} |v_i|. \tag{2.31}$$

Elles sont telles que

$$||\mathbf{v}||_2 \leqslant ||\mathbf{v}||_1 \leqslant n||\mathbf{v}||_\infty, \tag{2.32}$$

et

$$\mathbf{v}^{\mathrm{T}}\mathbf{w} \leqslant \|\mathbf{v}\|_2 \cdot \|\mathbf{w}\|_2. \tag{2.33}$$

Ce dernier résultat est l'*inégalité de Cauchy-Schwarz*.

*Remarque 2.6.* Si les éléments de $\mathbf{v}$ étaient complexes, les normes seraient définies différemment. La norme euclidienne, par exemple, deviendrait

$$||\mathbf{v}||_2 = \sqrt{\mathbf{v}^{\mathrm{H}}\mathbf{v}}, \tag{2.34}$$

où $\mathbf{v}^{\mathrm{H}}$ est le *transconjugué* de $\mathbf{v}$, c'est à dire le vecteur ligne obtenu en transposant le vecteur colonne $\mathbf{v}$ et en remplaçant chaque élément du résultat par son complexe conjugué. □

*Exemple 2.7.* Pour le vecteur complexe

$$\mathbf{v} = \begin{bmatrix} a \\ a\mathrm{i} \end{bmatrix},$$

où $a$ est un nombre réel non nul et où i est l'unité imaginaire (telle que $\mathrm{i}^2 = -1$), $\mathbf{v}^{\mathrm{T}}\mathbf{v} = 0$. Ceci prouve que $\sqrt{\mathbf{v}^{\mathrm{T}}\mathbf{v}}$ n'est pas une norme. La norme euclidienne de $\mathbf{v}$ vaut $\sqrt{\mathbf{v}^{\mathrm{H}}\mathbf{v}} = \sqrt{2}|a|$. □

*Remarque 2.7.* La mal-nommée norme $l_0$ d'un vecteur est le nombre de ses éléments non nuls. Elle est utilisée dans le contexte de l'estimation creuse, où l'on cherche un vecteur de paramètres estimés avec aussi peu d'éléments non nuls que possible. Ce n'est *pas* une norme, puisqu'elle ne satisfait pas la propriété de mise à l'échelle positive. □

### 2.5.2 Normes matricielles

Chaque norme vectorielle *induit* une *norme matricielle*, définie comme

$$||\mathbf{M}|| = \max_{||\mathbf{v}||=1} ||\mathbf{M}\mathbf{v}||, \tag{2.35}$$

de sorte que

$$\|\mathbf{M}\mathbf{v}\| \leqslant \|\mathbf{M}\| \cdot \|\mathbf{v}\| \tag{2.36}$$

pour tout $\mathbf{M}$ et tout $\mathbf{v}$ pour lesquels le produit $\mathbf{M}\mathbf{v}$ fait sens. Cette norme matricielle est *subordonnée* à la norme qui l'induit. Ces normes matricielle et vectorielle sont

alors dites *compatibles*, et cette propriété est importante pour l'étude de produits de matrices et de vecteurs.

— La norme matricielle induite par la norme vectorielle $l_2$ est la *norme spectrale*, ou *norme 2* ,

$$||\mathbf{M}||_2 = \sqrt{\rho(\mathbf{M}^{\mathrm{T}}\mathbf{M})}, \tag{2.37}$$

avec $\rho(\cdot)$ la fonction qui calcule le *rayon spectral* de son argument, c'est à dire le module de sa valeur propre de plus grand module. Comme toutes les valeurs propres de $\mathbf{M}^{\mathrm{T}}\mathbf{M}$ sont réelles et non-négatives, $\rho(\mathbf{M}^{\mathrm{T}}\mathbf{M})$ est la plus grande de ces valeurs propres. Sa racine carrée est la plus grande *valeur singulière* de $\mathbf{M}$, notée $\sigma_{\max}(\mathbf{M})$. On a donc

$$||\mathbf{M}||_2 = \sigma_{\max}(\mathbf{M}). \tag{2.38}$$

— La norme matricielle induite par la norme vectorielle $l_1$ est la *norme 1*

$$||\mathbf{M}||_1 = \max_j \sum_i |m_{i,j}|, \tag{2.39}$$

ce qui revient à sommer les valeurs absolues des éléments de chaque *colonne* successivement pour garder le résultat le plus grand.

— La norme matricielle induite par la norme vectorielle $l_\infty$ est la *norme infinie*

$$||\mathbf{M}||_\infty = \max_i \sum_j |m_{i,j}|, \tag{2.40}$$

ce qui revient à sommer les valeurs absolues des éléments de chaque *ligne* successivement pour garder le résultat le plus grand. Ainsi

$$||\mathbf{M}||_1 = ||\mathbf{M}^{\mathrm{T}}||_\infty. \tag{2.41}$$

Puisque chaque norme subordonnée est compatible avec la norme vectorielle qui l'induit,

$$||\mathbf{v}||_1 \text{ est compatible avec } ||\mathbf{M}||_1, \tag{2.42}$$
$$||\mathbf{v}||_2 \text{ est compatible avec } ||\mathbf{M}||_2, \tag{2.43}$$
$$||\mathbf{v}||_\infty \text{ est compatible avec } ||\mathbf{M}||_\infty. \tag{2.44}$$

La *norme de Frobenius*

$$||\mathbf{M}||_{\mathrm{F}} = \sqrt{\sum_{i,j} m_{i,j}^2} = \sqrt{\mathrm{trace}\,(\mathbf{M}^{\mathrm{T}}\mathbf{M})} \tag{2.45}$$

mérite une mention spéciale, car elle n'est induite par aucune norme vectorielle et pourtant

$$||\mathbf{v}||_2 \text{ est compatible avec } ||\mathbf{M}||_{\mathrm{F}}. \tag{2.46}$$

*Remarque 2.8.* Pour évaluer une norme vectorielle ou matricielle avec MATLAB (ou tout autre langage interprété à base de matrices), il est beaucoup plus efficace d'utiliser la fonction dédiée que d'accéder aux éléments du vecteur ou de la matrice pour appliquer la définition de la norme. Ainsi, `norm(X,p)` retourne la norme p de `X`, qui peut être un vecteur ou une matrice, tandis que `norm(M,'fro')` retourne la norme de Frobenius de la matrice `M`. □

### 2.5.3 Vitesses de convergence

Les normes peuvent servir à étudier la vitesse avec laquelle une méthode itérative convergerait vers la solution $\mathbf{x}^\star$ si les calculs étaient exacts. Définissons l'erreur à l'itération $k$ comme

$$\mathbf{e}^k = \mathbf{x}^k - \mathbf{x}^\star, \tag{2.47}$$

où $\mathbf{x}^k$ est l'estimée de $\mathbf{x}^\star$ à l'itération $k$. La vitesse de convergence *asymptotique* est *linéaire* si

$$\limsup_{k\to\infty} \frac{\|\mathbf{e}^{k+1}\|}{\|\mathbf{e}^k\|} = \alpha < 1, \tag{2.48}$$

avec $\alpha$ le taux de convergence.

Elle est *superlinéaire* si

$$\limsup_{k\to\infty} \frac{\|\mathbf{e}^{k+1}\|}{\|\mathbf{e}^k\|} = 0, \tag{2.49}$$

et *quadratique* si

$$\limsup_{k\to\infty} \frac{\|\mathbf{e}^{k+1}\|}{\|\mathbf{e}^k\|^2} = \alpha < \infty. \tag{2.50}$$

Une méthode à convergence quadratique a donc aussi une convergence superlinéaire et une convergence linéaire. Il est cependant d'usage d'attribuer à une méthode la meilleure convergence dont elle est capable. Asymptotiquement, une convergence quadratique est plus rapide qu'une convergence superlinéaire, elle-même plus rapide qu'une convergence linéaire.

Retenons que ces vitesses de convergence sont asymptotiques, valides quand l'erreur est devenue suffisamment petite, et qu'elles ne tiennent pas compte de l'effet des arrondis. Elles n'ont pas de sens quand le vecteur initial $\mathbf{x}^0$ est trop mal choisi pour que la méthode converge vers $\mathbf{x}^\star$. Quand celle-ci converge vers $\mathbf{x}^\star$, elles ne décrivent pas forcément bien son comportement initial, et ne seront plus vraies quand les erreurs dues aux arrondis deviendront prédominantes. Elles apportent néanmoins une information intéressante sur ce qu'on peut espérer au mieux.

# Chapitre 3
# Résoudre des systèmes d'équations linéaires

## 3.1 Introduction

Les équations linéaires sont des équations polynomiales du premier degré en leurs inconnues. Un système d'équations linéaires peut donc s'écrire

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.1}$$

où la matrice $\mathbf{A}$ et le vecteur $\mathbf{b}$ sont *connus* et où $\mathbf{x}$ est un vecteur d'inconnues. Nous supposons dans ce chapitre que

— tous les éléments de $\mathbf{A}$, $\mathbf{b}$ et $\mathbf{x}$ sont des nombres réels,
— il y a $n$ équations scalaires à $n$ inconnues scalaires ($\mathbf{A}$ est une matrice carrée $n \times n$ et $\dim \mathbf{x} = \dim \mathbf{b} = n$),
— ces équations définissent $\mathbf{x}$ uniquement ($\mathbf{A}$ est inversible).

Quand $\mathbf{A}$ est inversible, la solution de (3.1) est donnée *mathématiquement* sous forme explicite par $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Nous ne sommes *pas* intéressés ici par cette solution explicite, et voulons au contraire calculer numériquement $\mathbf{x}$ à partir des valeurs numériquement connues de $\mathbf{A}$ et de $\mathbf{b}$. Ce problème joue un rôle central dans tant d'algorithmes qu'il mérite son propre chapitre. Les systèmes d'équations linéaires à plus d'équations que d'inconnues seront considérés en section 9.2.

*Remarque 3.1.* Quand $\mathbf{A}$ est carrée mais singulière (c'est à dire non inversible), ses colonnes ne forment plus une base de $\mathbb{R}^n$, de sorte que le vecteur $\mathbf{A}\mathbf{x}$ ne peut pas prendre une direction arbitraire dans $\mathbb{R}^n$. La direction de $\mathbf{b}$ déterminera alors si (3.1) a une infinité de solutions ou aucune.

Quand $\mathbf{b}$ peut être exprimé comme une combinaison linéaire de colonnes de $\mathbf{A}$, des équations sont linéairement dépendantes et il y a *un continuum de solutions*. Le système $x_1 + x_2 = 1$ et $2x_1 + 2x_2 = 2$ correspond à cette situation.

Quand $\mathbf{b}$ ne peut pas être exprimé comme une combinaison linéaire de colonnes de $\mathbf{A}$, les équations sont incompatibles et il n'y a *aucune solution*. Le système $x_1 + x_2 = 1$ et $x_1 + x_2 = 2$ correspond à cette situation. □

D'excellents livres sur les thèmes de ce chapitre et du chapitre 4 (ainsi que sur de nombreux thèmes d'autres chapitres) sont [80], [49] et [6].

## 3.2 Exemples

*Exemple 3.1. Détermination d'un équilibre statique*

Les conditions pour qu'un système dynamique linéaire soit en équilibre statique se traduisent par un système d'équations linéaires. Considérons, par exemple, trois ressorts verticaux en série $s_i$ $(i = 1,2,3)$, dont le premier est attaché au plafond et le dernier à un objet de masse $m$. Négligeons la masse de chaque ressort, et notons $k_i$ le coefficient de raideur du $i$-ème ressort. Nous voulons calculer l'élongation $x_i$ de l'extrémité inférieure du ressort $i$ $(i = 1,2,3)$ qui résulte de l'action de la masse de l'objet quand le système a atteint son équilibre statique. La somme de toutes les forces agissant en un point quelconque est alors nulle. Pourvu que $m$ soit suffisamment petite pour que la loi d'élasticité de Hooke s'applique, les équations linéaires suivantes sont ainsi satisfaites :

$$\begin{aligned} mg &= k_3(x_3 - x_2), && (3.2)\\ k_3(x_2 - x_3) &= k_2(x_1 - x_2), && (3.3)\\ k_2(x_2 - x_1) &= k_1 x_1, && (3.4) \end{aligned}$$

avec $g$ l'accélération due à la gravité. Ce système d'équations peut encore s'écrire

$$\begin{bmatrix} k_1+k_2 & -k_2 & 0 \\ -k_2 & k_2+k_3 & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}. \qquad (3.5)$$

La matrice du membre de gauche de (3.5) est *tridiagonale*, car tous ses éléments non nuls sont sur sa diagonale principale et sur les diagonales situées juste au dessus et juste en dessous. Ceci resterait vrai s'il y avait beaucoup plus de ressorts en série, auquel cas la matrice serait aussi *creuse*, c'est à dire avec une majorité d'éléments nuls. Notons qu'un changement de la masse de l'objet ne modifierait que le membre de droite de (3.5). On peut ainsi être intéressé à la résolution de plusieurs systèmes linéaires de même matrice $\mathbf{A}$. □

*Exemple 3.2. Interpolation polynomiale.*

Supposons la valeur $y_i$ d'une quantité d'intérêt mesurée aux instants $t_i$, pour $i = 1, 2, 3$. Pour interpoler ces données avec le polynôme

$$P(t,\mathbf{x}) = a_0 + a_1 t + a_2 t^2, \qquad (3.6)$$

où $\mathbf{x} = (a_0, a_1, a_2)^{\mathrm{T}}$, il suffit de résoudre (3.1) avec

$$\mathbf{A} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \end{bmatrix} \quad \text{et} \quad \mathbf{b} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{3.7}$$

Voir le chapitre 5 pour plus d'informations sur l'interpolation. □

## 3.3 Conditionnement

La notion de conditionnement joue un rôle central dans l'évaluation de la difficulté intrinsèque de la résolution d'un problème numérique donné *indépendamment de l'algorithme qui sera employé* [193, 48]. Elle peut donc être utilisée pour détecter des problèmes pour lesquels il convient d'être particulièrement prudent. Nous nous limitons ici au problème du calcul du $\mathbf{x}$ solution de (3.1). En général, $\mathbf{A}$ et $\mathbf{b}$ sont imparfaitement connues, pour au moins deux raisons. Premièrement, le simple fait de convertir un nombre réel en sa représentation à virgule flottante ou de conduire des calculs sur des flottants se traduit presque toujours par des approximations. De plus, les éléments de $\mathbf{A}$ et $\mathbf{b}$ résultent souvent de mesures imprécises. Il est donc important de quantifier l'effet que des perturbations sur $\mathbf{A}$ et $\mathbf{b}$ peuvent avoir sur la solution $\mathbf{x}$.

Remplaçons $\mathbf{A}$ par $\mathbf{A}+\delta\mathbf{A}$ et $\mathbf{b}$ par $\mathbf{b}+\delta\mathbf{b}$, et définissons $\widehat{\mathbf{x}}$ comme la solution du système perturbé

$$(\mathbf{A}+\delta\mathbf{A})\widehat{\mathbf{x}} = \mathbf{b}+\delta\mathbf{b}. \tag{3.8}$$

La différence entre les solutions du système perturbé (3.8) et du système original (3.1) est

$$\delta\mathbf{x} = \widehat{\mathbf{x}} - \mathbf{x}. \tag{3.9}$$

Elle satisfait

$$\delta\mathbf{x} = \mathbf{A}^{-1}[\delta\mathbf{b} - (\delta\mathbf{A})\widehat{\mathbf{x}}]. \tag{3.10}$$

Pourvu que des normes compatibles soient utilisées, ceci implique que

$$||\delta\mathbf{x}|| \leqslant ||\mathbf{A}^{-1}|| \cdot (||\delta\mathbf{b}|| + ||\delta\mathbf{A}|| \cdot ||\widehat{\mathbf{x}}||). \tag{3.11}$$

Divisons les deux côtés de (3.11) par $||\widehat{\mathbf{x}}||$, et multiplions le côté droit du résultat par $||\mathbf{A}||/||\mathbf{A}||$ pour obtenir

$$\frac{||\delta\mathbf{x}||}{||\widehat{\mathbf{x}}||} \leqslant ||\mathbf{A}^{-1}|| \cdot ||\mathbf{A}|| \left( \frac{||\delta\mathbf{b}||}{||\mathbf{A}|| \cdot ||\widehat{\mathbf{x}}||} + \frac{||\delta\mathbf{A}||}{||\mathbf{A}||} \right). \tag{3.12}$$

Le coefficient multiplicatif $||\mathbf{A}^{-1}|| \cdot ||\mathbf{A}||$ dans le membre de droite de (3.12) est le *conditionnement* de $\mathbf{A}$

$$\text{cond } \mathbf{A} = ||\mathbf{A}^{-1}|| \cdot ||\mathbf{A}||. \tag{3.13}$$

Il quantifie les conséquences qu'une erreur sur $\mathbf{A}$ ou $\mathbf{b}$ *peut* avoir sur l'erreur sur $\mathbf{x}$. Nous souhaitons qu'il soit aussi petit que possible, pour que la solution soit aussi insensible que possible aux erreurs $\delta\mathbf{A}$ et $\delta\mathbf{b}$.

*Remarque 3.2.* Quand les erreurs sur $\mathbf{b}$ sont négligeables, (3.12) devient

$$\frac{||\delta\mathbf{x}||}{||\widehat{\mathbf{x}}||} \leqslant (\text{cond } \mathbf{A}) \cdot \left( \frac{||\delta\mathbf{A}||}{||\mathbf{A}||} \right). \tag{3.14}$$

□

*Remarque 3.3.* Quand les erreurs sur $\mathbf{A}$ sont négligeables,

$$\delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b}, \tag{3.15}$$

de sorte que

$$\|\delta\mathbf{x}\| \leqslant \|\mathbf{A}^{-1}\| \cdot \|\delta\mathbf{b}\|. \tag{3.16}$$

Or (3.1) implique que

$$\|\mathbf{b}\| \leqslant \|\mathbf{A}\| \cdot \|\mathbf{x}\|, \tag{3.17}$$

et (3.16) et (3.17) impliquent que

$$\|\delta\mathbf{x}\| \cdot \|\mathbf{b}\| \leqslant \|\mathbf{A}^{-1}\| \cdot \|\mathbf{A}\| \cdot \|\delta\mathbf{b}\| \cdot \|\mathbf{x}\|, \tag{3.18}$$

de sorte que

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leqslant (\text{cond } \mathbf{A}) \cdot \left( \frac{||\delta\mathbf{b}||}{||\mathbf{b}||} \right). \tag{3.19}$$

□

Puisque

$$1 = ||\mathbf{I}|| = ||\mathbf{A}^{-1} \cdot \mathbf{A}|| \leqslant ||\mathbf{A}^{-1}|| \cdot ||\mathbf{A}||, \tag{3.20}$$

le conditionnement de $\mathbf{A}$ satisfait

$$\text{cond } \mathbf{A} \geqslant 1. \tag{3.21}$$

Sa valeur dépend de la norme utilisée. Pour la norme spectrale, on a

$$||\mathbf{A}||_2 = \sigma_{\max}(\mathbf{A}), \tag{3.22}$$

où $\sigma_{\max}(\mathbf{A})$ est la plus grande valeur singulière de $\mathbf{A}$. Puisque

$$||\mathbf{A}^{-1}||_2 = \sigma_{\max}(\mathbf{A}^{-1}) = \frac{1}{\sigma_{\min}(\mathbf{A})}, \tag{3.23}$$

avec $\sigma_{\min}(\mathbf{A})$ la plus petite valeur singulière de $\mathbf{A}$, le conditionnement de $\mathbf{A}$ pour la norme spectrale est le rapport de sa plus grande valeur singulière à sa plus petite :

$$\text{cond } \mathbf{A} = \frac{\sigma_{\max}(\mathbf{A})}{\sigma_{\min}(\mathbf{A})}. \tag{3.24}$$

Plus le conditionnement de $\mathbf{A}$ croît, plus (3.1) devient mal conditionnée.

Il est utile de comparer cond $\mathbf{A}$ avec l'inverse de la précision de la représentation à virgule flottante utilisée. Pour une représentation en double précision suivant la norme IEEE 754 (typique des calculs en MATLAB), cette précision est à peu près égale à $10^{-16}$.

> Chercher à évaluer le $\mathbf{x}$ solution de (3.1) demande un soin particulier quand cond $\mathbf{A}$ n'est pas petit devant $10^{16}$.

*Remarque 3.4.* Bien que ce soit sans doute la pire des méthodes pour les évaluer numériquement, les valeurs singulières de $\mathbf{A}$ sont les racines carrées des valeurs propres de $\mathbf{A}^{\mathrm{T}}\mathbf{A}$. (Quand $\mathbf{A}$ est symétrique, ses valeurs singulières sont donc égales aux valeurs absolues de ses valeurs propres.) □

*Remarque 3.5.* $\mathbf{A}$ est singulière si et seulement si son déterminant est nul. On pourrait donc penser à utiliser la valeur de ce déterminant comme mesure du conditionnement, un petit déterminant indiquant que le système d'équations est presque singulier. Il est cependant très difficile de vérifier qu'un nombre à virgule flottante diffère de zéro de façon significative (penser à ce qui arrive au déterminant de $\mathbf{A}$ quand $\mathbf{A}$ et $\mathbf{b}$ sont multipliés par un nombre positif grand ou petit, ce qui n'a aucun effet sur la difficulté du problème). Le conditionnement tel qu'il a été défini a beaucoup plus de sens, car il est invariant par multiplication de $\mathbf{A}$ par un scalaire non nul de magnitude quelconque (une conséquence de la mise à l'échelle positive de la norme). Comparer $\det(10^{-1}\mathbf{I}_n) = 10^{-n}$ avec $\mathrm{cond}(10^{-1}\mathbf{I}_n) = 1$. □

*Remarque 3.6.* La valeur numérique de cond $\mathbf{A}$ dépend de la norme utilisée, mais un problème mal conditionné pour une norme le sera aussi pour les autres, de sorte que le choix de la norme est affaire de commodité. □

*Remarque 3.7.* Bien que l'évaluation du conditionnement d'une matrice pour la norme spectrale ne demande qu'un appel à la fonction MATLAB `cond(·)`, ceci peut en fait demander plus de calculs que la résolution de (3.1). L'évaluation du conditionnement de la même matrice pour la norme 1 (par un appel à la fonction `cond(·,1)`) est moins coûteux que pour la norme spectrale, et il existe des algorithmes pour estimer son ordre de grandeur à moindre coût [49, 99, 105]. □

*Remarque 3.8.* Le concept de conditionnement s'étend aux matrices rectangulaires, le conditionnement pour la norme spectrale restant alors donné par (3.24). On peut également l'étendre à des problèmes non linéaires, voir la section 14.5.2.1. □

## 3.4 Approches à éviter

> Pour résoudre numériquement un système d'équations linéaires, *l'inversion de matrice* est presque toujours à éviter, car elle demande en général des calculs inutiles.

À moins que $\mathbf{A}$ n'ait une structure telle que son inversion soit particulièrement simple, il faut donc y réfléchir à deux fois avant d'inverser $\mathbf{A}$ pour tirer avantage de la solution explicite

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{3.25}$$

La règle de Cramer pour la résolution de systèmes d'équations linéaires, qui requiert le calcul de rapports de déterminants, est la *pire approche possible*. Les déterminants sont connus pour être difficiles à évaluer avec précision, et leur calcul est inutilement coûteux, même s'il existe des méthodes plus économiques que l'expansion selon les cofacteurs.

## 3.5 Questions sur A

La matrice $\mathbf{A}$ a souvent des propriétés spécifiques dont on peut tirer avantage en choisissant une méthode appropriée. Il est donc important de se poser les questions suivantes :

- $\mathbf{A}$ et $\mathbf{b}$ sont-ils à éléments réels (comme nous le supposons ici) ?
- $\mathbf{A}$ est-elle carrée et inversible (comme nous le supposons ici) ?
- $\mathbf{A}$ est-elle symétrique, c'est à dire telle que $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$ ?
- $\mathbf{A}$ est-elle *symétrique définie positive* (ce qu'on note par $\mathbf{A} \succ 0$) ? Ceci veut dire que $\mathbf{A}$ est symétrique et telle que

  $$\forall \mathbf{v} \neq \mathbf{0}, \mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} > 0, \tag{3.26}$$

  ce qui implique que toutes ses valeurs propres sont réelles et strictement positives.
- Si $\mathbf{A}$ est de grande taille, est-elle *creuse*, c'est à dire telle que la plupart de ses éléments sont nuls ?
- $\mathbf{A}$ est-elle *à diagonale dominante*, c'est à dire telle que la valeur absolue de chacun de ses éléments diagonaux est strictement supérieure à la somme des valeurs absolues de tous les autres éléments de la même ligne ?
- $\mathbf{A}$ est-elle *tridiagonale*, c'est à dire telle que seule sa diagonale principale et les diagonales situées juste au dessus et en dessous sont non nulles ? Elle peut alors s'écrire

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix}. \tag{3.27}$$

— **A** est-elle une matrice de *Toeplitz*, c'est à dire telle que tous les éléments de la même diagonale descendante aient la même valeur ? Elle peut alors s'écrire

$$\mathbf{A} = \begin{bmatrix} h_0 & h_{-1} & h_{-2} & \cdots & h_{-n+1} \\ h_1 & h_0 & h_{-1} & & h_{-n+2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & h_{-1} \\ h_{n-1} & h_{n-2} & \cdots & h_1 & h_0 \end{bmatrix}. \tag{3.28}$$

— **A** est-elle *bien conditionnée* ? (Voir la section 3.3.)

## 3.6 Méthodes directes

Les *méthodes directes* évaluent le vecteur **x** solution de (3.1) en un nombre fini de pas. Elles requièrent une quantité de ressources prévisible et peuvent être rendues très robustes, mais deviennent difficiles à appliquer sur les problèmes de très grande taille. Ceci les différencie des *méthodes itératives*, considérées dans la section 3.7, qui visent à générer une suite d'approximations de la solution. Certaines méthodes itératives peuvent traiter des problèmes avec des millions d'inconnues, comme on en rencontre par exemple lors de la résolution d'équations aux dérivées partielles.

*Remarque 3.9.* La distinction entre les méthodes directes et itératives n'est pas aussi nette qu'il pourrait sembler ; des résultats obtenus par des méthodes directes peuvent être améliorés par des techniques itératives (comme en section 3.6.4), et les méthodes itératives les plus sophistiquées (présentées en section 3.7.2) trouveraient la solution exacte en un nombre fini de pas si les calculs étaient conduits de façon exacte. □

### *3.6.1 Substitution arrière ou avant*

La substitution arrière ou avant s'applique quand **A** est triangulaire. C'est un cas moins particulier qu'on pourrait le penser, car plusieurs des méthodes présentées

plus loin et applicables à des systèmes linéaires génériques passent par la résolution de systèmes triangulaires.

La *substitution arrière* s'applique au système triangulaire supérieur

$$\mathbf{U}\mathbf{x} = \mathbf{b}, \tag{3.29}$$

où

$$\mathbf{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & & u_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{n,n} \end{bmatrix}. \tag{3.30}$$

Quand $\mathbf{U}$ est inversible, tous ses éléments diagonaux sont non nuls et (3.29) peut être résolue inconnue par inconnue en commençant par la dernière

$$x_n = b_n/u_{n,n}, \tag{3.31}$$

puis en remontant pour obtenir

$$x_{n-1} = (b_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1,} \tag{3.32}$$

et ainsi de suite, avec finalement

$$x_1 = (b_1 - u_{1,2}x_2 - u_{1,3}x_3 - \cdots - u_{1,n}x_n)/u_{1,1}. \tag{3.33}$$

La *substitution avant*, quant à elle, s'applique au système triangulaire inférieur

$$\mathbf{L}\mathbf{x} = \mathbf{b}, \tag{3.34}$$

où

$$\mathbf{L} = \begin{bmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ l_{n,1} & l_{n,2} & \ldots & l_{n,n} \end{bmatrix}. \tag{3.35}$$

Elle résout aussi (3.34) inconnue par inconnue, mais commence par $x_1$ puis descend pour évaluer $x_2$ et ainsi de suite jusqu'à ce que la valeur de $x_n$ soit obtenue.

La résolution de (3.29) par substitution arrière peut être menée à bien en MATLAB via l'instruction `x=linsolve(U,b,optsUT)`, avec `optsUT.UT=true`, ce qui spécifie que `U` est une matrice triangulaire supérieure. On peut de même résoudre (3.34) par substitution avant via `x=linsolve(L,b,optsLT)`, pourvu que `optsLT.LT=true`, ce qui spécifie que `L` est une matrice triangulaire inférieure.

### 3.6.2 Élimination gaussienne

L'élimination gaussienne [104] transforme le système d'équations initial (3.1) en un système triangulaire supérieur

$$\mathbf{U}\mathbf{x} = \mathbf{v}, \tag{3.36}$$

en remplaçant chaque ligne de $\mathbf{A}\mathbf{x}$ *et* de $\mathbf{b}$ par une combinaison appropriée de telles lignes. Ce système triangulaire est alors résolu par substitution arrière, inconnue par inconnue. Tout ceci est mené à bien par l'instruction MATLAB `x=A\b`. Cette agréable concision cache en réalité le fait que $\mathbf{A}$ a été factorisée, et la factorisation résultante n'est donc pas disponible pour une utilisation ultérieure, par exemple pour résoudre (3.1) avec la même matrice $\mathbf{A}$ mais un autre vecteur $\mathbf{b}$.

Quand (3.1) doit être résolu pour plusieurs membres de droite $\mathbf{b}^i$ ($i = 1, \cdots, m$) *tous connus à l'avance*, le système

$$\mathbf{A}\mathbf{x}^1 \cdots \mathbf{x}^m = \mathbf{b}^1 \cdots \mathbf{b}^m \tag{3.37}$$

est transformé de façon similaire par combinaison de lignes en

$$\mathbf{U}\mathbf{x}^1 \cdots \mathbf{x}^m = \mathbf{v}^1 \cdots \mathbf{v}^m. \tag{3.38}$$

Les solutions sont alors obtenues en résolvant les systèmes triangulaires

$$\mathbf{U}\mathbf{x}^i = \mathbf{v}^i, \qquad i = 1, \cdots, m. \tag{3.39}$$

Cette approche classique pour résoudre (3.1) n'a pas d'avantage par rapport à la factorisation LU présentée ci-dessous. Comme elle travaille simultanément sur $\mathbf{A}$ et $\mathbf{b}$, l'élimination gaussienne pour un second membre $\mathbf{b}$ précédemment inconnu ne peut pas tirer avantage des calculs effectués avec d'autres seconds membres, même si $\mathbf{A}$ reste la même.

### 3.6.3 Factorisation LU

La factorisation LU, une reformulation matricielle de l'élimination gaussienne, est l'outil de base à utiliser quand $\mathbf{A}$ n'a pas de structure particulière exploitable. Commençons par sa version la plus simple.

#### 3.6.3.1 Factorisation LU sans pivotage

$\mathbf{A}$ est factorisée comme

$$\mathbf{A} = \mathbf{L}\mathbf{U}, \tag{3.40}$$

où **L** est triangulaire inférieure (*lower*) et **U** triangulaire supérieure (*upper*). (On parle aussi de *factorisation LR*, avec L triangulaire à gauche (*left*) et R triangulaire à droite (*right*).)

*Quand elle est possible*, cette factorisation n'est pas unique, car il y a $n^2+n$ inconnues dans **L** et **U** pour seulement $n^2$ éléments dans **A**, qui fournissent autant de relations scalaires entre ces inconnues. Il est donc nécessaire d'ajouter $n$ contraintes pour assurer l'unicité. C'est pourquoi nous donnons une valeur unité à chacun des éléments diagonaux de **L**. L'équation (3.40) se traduit alors par

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n,1} & \cdots & l_{n,n-1} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & & u_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{n,n} \end{bmatrix}. \tag{3.41}$$

Quand (3.41) admet une solution en $l_{i,j}$ et $u_{i,j}$, cette solution peut être obtenue très simplement en considérant dans un ordre approprié les équations scalaires résultantes. Chaque inconnue est alors exprimée en fonction d'éléments de **A** et d'éléments déjà évalués de **L** et de **U**. Pour simplifier les notations, et parce que notre but n'est pas de programmer la factorisation LU, bornons-nous à illustrer ceci avec un tout petit exemple.

*Exemple 3.3. Factorisation LU sans pivotage*

Pour le système

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ l_{2,1} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{1,1} & u_{1,2} \\ 0 & u_{2,2} \end{bmatrix}, \tag{3.42}$$

nous obtenons

$$u_{1,1} = a_{1,1}, \quad u_{1,2} = a_{1,2}, \quad l_{2,1}u_{1,1} = a_{2,1} \quad \text{et} \quad l_{2,1}u_{1,2} + u_{2,2} = a_{2,2}. \tag{3.43}$$

De sorte que, pourvu que $a_{11} \neq 0$,

$$l_{2,1} = \frac{a_{2,1}}{u_{1,1}} = \frac{a_{2,1}}{a_{1,1}} \quad \text{et} \quad u_{2,2} = a_{2,2} - l_{2,1}u_{1,2} = a_{2,2} - \frac{a_{2,1}}{a_{1,1}}a_{1,2}. \tag{3.44}$$

□

Les termes qui apparaissent dans des dénominateurs, comme $a_{1,1}$ dans l'exemple 3.3, sont appelés *pivots*. La factorisation LU sans pivotage échoue chaque fois qu'un pivot se révèle nul.

Après factorisation LU, le système à résoudre est

$$\mathbf{LUx} = \mathbf{b}. \tag{3.45}$$

La solution est évaluée en deux étapes. On évalue d'abord **y** solution de

$$\mathbf{Ly} = \mathbf{b}. \tag{3.46}$$

Puisque **L** est triangulaire inférieure, ceci est mené à bien par *substitution avant*, chaque équation fournissant la solution pour une nouvelle inconnue. Comme les éléments diagonaux de **L** sont tous égaux à un, ceci est particulièrement simple.

On évalue ensuite **x** solution de

$$\mathbf{U}\mathbf{x} = \mathbf{y}. \tag{3.47}$$

Comme **U** est triangulaire supérieure, ceci est mené à bien par *substitution arrière*, chaque équation fournissant ici aussi la solution pour une nouvelle inconnue.

*Exemple 3.4. Échec de la factorisation LU sans pivotage*

Pour

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

le pivot $a_{1,1}$ est égal à zéro, de sorte que l'algorithme échoue à moins qu'on effectue un pivotage. Notons qu'il suffit ici d'échanger les lignes de **A** (ainsi que celles de **b**) pour que le problème disparaisse. □

*Remarque 3.10.* Quand aucun pivot n'est nul, le pivotage garde un rôle crucial pour améliorer la qualité de la factorisation LU si certains pivots sont proches de zéro. □

#### 3.6.3.2 Pivotage

Le pivotage est l'action de changer l'ordre des équations (ainsi éventuellement que celui des variables) pour éviter des pivots nuls ou trop petits. Quand on ne réordonne que les équations, on parle de *pivotage partiel*, tandis que le *pivotage total* change aussi l'ordre des variables. (Le pivotage total est peu utilisé, car il conduit rarement à de meilleurs résultats tout en étant plus coûteux ; nous ne l'aborderons pas.)

Changer l'ordre des équations revient à permuter les mêmes lignes dans **A** et dans **b**, ce qu'on peut réaliser en multipliant **A** et **b** à gauche par une *matrice de permutation* appropriée. La matrice de permutation **P** qui échange les $i$-ème et $j$-ème lignes de **A** est obtenue en échangeant les $i$-ème et $j$-ème lignes de la matrice identité. Ainsi, par exemple,

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} b_3 \\ b_1 \\ b_2 \end{bmatrix}. \tag{3.48}$$

Puisque $\det \mathbf{I} = 1$ et que tout échange de deux lignes change le signe du déterminant,

$$\det \mathbf{P} = \pm 1. \tag{3.49}$$

**P** est une *matrice orthonormale* (aussi appelée *matrice unitaire*), c'est à dire telle que

$$\mathbf{P}^{\mathrm{T}}\mathbf{P} = \mathbf{I}. \tag{3.50}$$

L'inverse de **P** est donc particulièrement facile à calculer, puisque

$$\mathbf{P}^{-1} = \mathbf{P}^{\mathrm{T}}. \tag{3.51}$$

Enfin, le produit de matrices de permutation est une matrice de permutation.

#### 3.6.3.3 Factorisation LU avec pivotage partiel

Lors du calcul de la $i$-ème colonne de **L**, les lignes $i$ à $n$ de **A** sont réordonnées pour assurer que l'élément de plus grande valeur absolue de la $i$-ème colonne monte sur la diagonale (s'il n'y est déjà). Ceci garantit que tous les éléments de **L** sont bornés par un en valeur absolue. L'algorithme résultant est décrit dans [49].

Soit **P** la matrice de permutation qui résume les échanges des lignes de **A** et de **b** décidés. Le système à résoudre devient

$$\mathbf{PAx} = \mathbf{Pb}, \tag{3.52}$$

et la factorisation LU est effectuée sur **PA**, de sorte que

$$\mathbf{LUx} = \mathbf{Pb}. \tag{3.53}$$

La résolution est à nouveau en deux étapes. On évalue d'abord **y** solution de

$$\mathbf{Ly} = \mathbf{Pb}, \tag{3.54}$$

puis on évalue **x** solution de

$$\mathbf{Ux} = \mathbf{y}. \tag{3.55}$$

Il est bien sûr inutile de stocker la matrice de permutation **P** (creuse) dans un tableau $n \times n$ ; il suffit de garder trace des échanges de lignes associés.

*Remarque 3.11.* Des algorithmes pour résoudre des systèmes d'équations linéaires via une factorisation avec pivotage partiel ou total sont facilement et gratuitement disponibles sur le WEB avec une documentation détaillée (dans LAPACK, par exemple, voir le chapitre 15). La même remarque s'applique à la plupart des méthodes présentées dans ce livre. En MATLAB, la factorisation LU avec pivotage partiel de `A` correspond à l'instruction `[L,U,P]=lu(A)`. □

*Remarque 3.12.* Bien que la stratégie de pivotage de la factorisation LU n'ait pas pour but de maintenir inchangé le conditionnement du problème, l'augmentation de celui-ci est limitée, ce qui rend la factorisation LU avec pivotage partiel utilisable même pour certains problèmes mal conditionnés, comme en section 3.10.1. □

La factorisation LU est un premier exemple de l'*approche par décomposition* du calcul matriciel [226], où une matrice est exprimée comme un produit de facteurs. D'autres exemples sont la *factorisation QR* (sections 3.6.5 et 9.2.3), la *décomposition en valeurs singulières* (sections 3.6.6 et 9.2.4), la *factorisation de*

*Cholesky* (section 3.8.1), la *décomposition de Schur* et la *décomposition spectrale*, ces deux dernières étant menées à bien par l'algorithme QR (section 4.3.6). En concentrant leurs efforts sur le développement d'algorithmes efficaces et robustes pour quelques factorisations importantes, les numériciens ont permis la production d'ensembles de programmes de calcul matriciel très efficaces, avec des applications étonnamment diverses. Des économies considérables peuvent être réalisées quand de nombreux problèmes partagent la même matrice, factorisée une fois pour toutes. Après la factorisation LU d'une matrice $\mathbf{A}$, par exemple, tous les systèmes (3.1) qui ne diffèrent que par leurs vecteurs $\mathbf{b}$ sont facilement résolus grâce à cette factorisation, même si les valeurs de $\mathbf{b}$ à considérer n'étaient pas encore connues quand elle a été calculée. C'est un net avantage sur l'élimination gaussienne où la factorisation de $\mathbf{A}$ est cachée dans la solution de (3.1) pour un $\mathbf{b}$ pré-spécifié.

### *3.6.4 Amélioration itérative*

Soit $\widehat{\mathbf{x}}$ le résultat obtenu en résolvant numériquement (3.1) par factorisation LU. Le résidu $\mathbf{A}\widehat{\mathbf{x}} - \mathbf{b}$ devrait être petit, mais ceci ne garantit pas que $\widehat{\mathbf{x}}$ soit une bonne approximation de la solution mathématique $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. On peut essayer d'améliorer $\widehat{\mathbf{x}}$ en évaluant un vecteur de correction $\delta\mathbf{x}$ tel que

$$\mathbf{A}(\widehat{\mathbf{x}} + \delta\mathbf{x}) = \mathbf{b}, \tag{3.56}$$

ou, de façon équivalente, que

$$\mathbf{A}\delta\mathbf{x} = \mathbf{b} - \mathbf{A}\widehat{\mathbf{x}}. \tag{3.57}$$

*Remarque 3.13.* Comme $\mathbf{A}$ est identique dans (3.57) et (3.1), sa factorisation LU est déjà disponible. □

Une fois $\delta\mathbf{x}$ obtenu en résolvant (3.57), $\widehat{\mathbf{x}}$ est remplacé par $\widehat{\mathbf{x}} + \delta\mathbf{x}$, et la procédure peut être itérée jusqu'à convergence, avec un critère d'arrêt sur $||\delta\mathbf{x}||$. Puisque le résidu $\mathbf{b} - \mathbf{A}\widehat{\mathbf{x}}$ correspond à la différence entre des nombres à virgule flottante qu'on espère proches, il est conseillé de le calculer en précision étendue.

Des améliorations spectaculaires peuvent être obtenues pour un effort aussi limité.

*Remarque 3.14.* L'amélioration itérative peut trouver bien d'autres applications que la résolution de systèmes linéaires via une factorisation LU. □

### *3.6.5 Factorisation QR*

Toute matrice $\mathbf{A}$ inversible de dimensions $n \times n$ peut être factorisée comme

$$\mathbf{A} = \mathbf{QR}, \tag{3.58}$$

où $\mathbf{Q}$ est une matrice orthonormale $n \times n$, telle que $\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I}_n$, et $\mathbf{R}$ est une matrice triangulaire supérieure inversible $n \times n$ (que la tradition persiste à appeler $\mathbf{R}$ et non pas $\mathbf{U}$...). Cette factorisation QR est unique si l'on impose que les éléments diagonaux de $\mathbf{R}$ soient positifs, ce qui n'a rien d'obligatoire. Elle peut être menée à bien en un nombre fini de pas. En MATLAB, la factorisation QR de `A` est effectuée par l'instruction `[Q,R]=qr(A)`.

Multiplions (3.1) à gauche par $\mathbf{Q}^{\mathrm{T}}$ en tenant compte de (3.58) pour obtenir

$$\mathbf{Rx} = \mathbf{Q}^{\mathrm{T}}\mathbf{b}, \tag{3.59}$$

qui est facile à résoudre par rapport à $\mathbf{x}$ car $\mathbf{R}$ est triangulaire.

Pour la norme spectrale, le conditionnement de $\mathbf{R}$ est le même que celui de $\mathbf{A}$, puisque

$$\mathbf{A}^{\mathrm{T}}\mathbf{A} = (\mathbf{QR})^{\mathrm{T}}\mathbf{QR} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR} = \mathbf{R}^{\mathrm{T}}\mathbf{R}. \tag{3.60}$$

La factorisation QR ne détériore donc pas le conditionnement. C'est un avantage par rapport à la factorisation LU, qu'on paye par plus de calculs.

*Remarque 3.15.* Contrairement à la factorisation LU, la factorisation QR s'applique aussi aux matrices rectangulaires, et se révèle très utile pour la résolution de problèmes de moindres carrés linéaires, voir la section 9.2.3. □

Au moins en principe, l'orthogonalisation de Gram-Schmidt pourrait être utilisée pour effectuer des factorisations QR, mais elle souffre d'instabilité numérique quand les colonnes de $\mathbf{A}$ sont presque linéairement dépendantes. C'est pourquoi l'approche plus robuste présentée dans la section suivante est en général préférée, bien qu'une méthode de Gram-Schmidt *modifiée* puisse également être employée [18].

#### 3.6.5.1 Transformation de Householder

L'outil de base pour la factorisation QR est la *transformation de Householder*, décrite par la matrice éponyme

$$\mathbf{H}(\mathbf{v}) = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^{\mathrm{T}}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}, \tag{3.61}$$

où $\mathbf{v}$ est un vecteur à choisir. Le vecteur $\mathbf{H}(\mathbf{v})\mathbf{x}$ est le symétrique de $\mathbf{x}$ par rapport à l'hyperplan passant par l'origine O et orthogonal à $\mathbf{v}$ (figure 3.1).

La matrice $\mathbf{H}(\mathbf{v})$ est symétrique et orthonormale. Ainsi

$$\mathbf{H}(\mathbf{v}) = \mathbf{H}^{\mathrm{T}}(\mathbf{v}) \quad \text{et} \quad \mathbf{H}^{\mathrm{T}}(\mathbf{v})\mathbf{H}(\mathbf{v}) = \mathbf{I}, \tag{3.62}$$

ce qui implique que

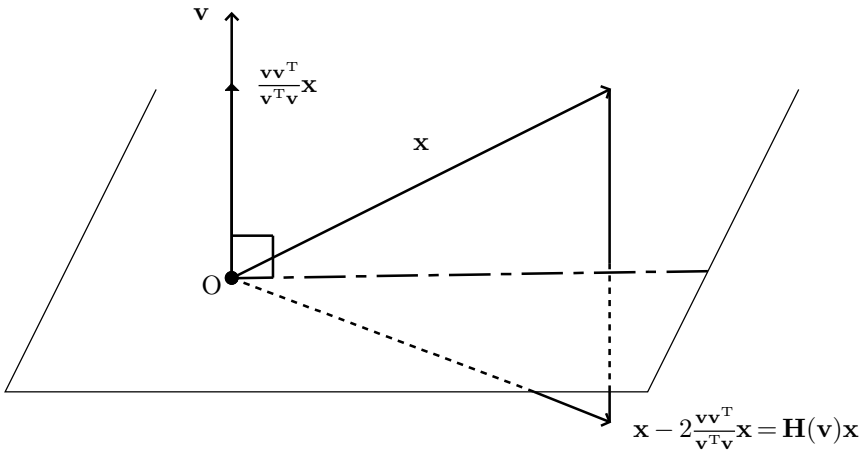


**Fig. 3.1** Transformation de Householder

$$\mathbf{H}^{-1}(\mathbf{v}) = \mathbf{H}(\mathbf{v}). \tag{3.63}$$

De plus, puisque $\mathbf{v}$ est un vecteur propre de $\mathbf{H}(\mathbf{v})$ associé à la valeur propre $-1$ et que tous les autres vecteurs propres de $\mathbf{H}(\mathbf{v})$ sont associés à une valeur propre unité,

$$\det \mathbf{H}(\mathbf{v}) = -1. \tag{3.64}$$

Cette propriété sera utile pour le calcul de déterminants en section 4.2.

Choisissons

$$\mathbf{v} = \mathbf{x} \pm ||\mathbf{x}||_2 \mathbf{e}^1, \tag{3.65}$$

où $\mathbf{e}^1$ est le vecteur correspondant à la première colonne de la matrice identité, et où le symbole $\pm$ indique qu'on peut choisir un signe plus ou moins. La proposition qui suit permet d'utiliser $\mathbf{H}(\mathbf{v})$ pour transformer $\mathbf{x}$ en un vecteur dont tous les éléments sont nuls sauf le premier.

**Proposition 3.1.** *Si*

$$\mathbf{H}_{(+)} = \mathbf{H}(\mathbf{x} + ||\mathbf{x}||_2 \mathbf{e}^1) \tag{3.66}$$

*et*

$$\mathbf{H}_{(-)} = \mathbf{H}(\mathbf{x} - ||\mathbf{x}||_2 \mathbf{e}^1), \tag{3.67}$$

*alors*

$$\mathbf{H}_{(+)}\mathbf{x} = -||\mathbf{x}||_2 \mathbf{e}^1 \tag{3.68}$$

*et*

$$\mathbf{H}_{(-)}\mathbf{x} = +||\mathbf{x}||_2 \mathbf{e}^1. \tag{3.69}$$

□

*Preuve.* Si $\mathbf{v} = \mathbf{x} \pm ||\mathbf{x}||_2 \mathbf{e}^1$, alors

$$\mathbf{v}^{\mathrm{T}}\mathbf{v} = \mathbf{x}^{\mathrm{T}}\mathbf{x} + \|\mathbf{x}\|_2^2(\mathbf{e}^1)^{\mathrm{T}}\mathbf{e}^1 \pm 2\|\mathbf{x}\|_2 x_1 = 2(\|\mathbf{x}\|_2^2 \pm \|\mathbf{x}\|_2 x_1) = 2\mathbf{v}^{\mathrm{T}}\mathbf{x}. \tag{3.70}$$

Ainsi

$$\mathbf{H}(\mathbf{v})\mathbf{x} = \mathbf{x} - 2\mathbf{v}\left(\frac{\mathbf{v}^{\mathrm{T}}\mathbf{x}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}\right) = \mathbf{x} - \mathbf{v} = \mp||\mathbf{x}||_2\mathbf{e}^1. \tag{3.71}$$

□

Entre $\mathbf{H}_{(+)}$ et $\mathbf{H}_{(-)}$, il faut choisir

$$\mathbf{H}_{\text{best}} = \mathbf{H}(\mathbf{x} + \text{signe}\,(x_1)||\mathbf{x}||_2\mathbf{e}^1), \tag{3.72}$$

pour se protéger contre le risque de devoir évaluer la différence de deux nombres à virgule flottante proches. En pratique, la matrice $\mathbf{H}(\mathbf{v})$ n'est pas formée. On calcule à la place le scalaire

$$\rho = 2\frac{\mathbf{v}^{\mathrm{T}}\mathbf{x}}{\mathbf{v}^{\mathrm{T}}\mathbf{v}}, \tag{3.73}$$

et le vecteur

$$\mathbf{H}(\mathbf{v})\mathbf{x} = \mathbf{x} - \rho\mathbf{v}. \tag{3.74}$$

#### 3.6.5.2 Combiner des transformations de Householder

**A** est triangularisée en lui faisant subir une série de transformations de Householder, comme suit.

Commencer avec $\mathbf{A}_0 = \mathbf{A}$.

Calculer $\mathbf{A}_1 = \mathbf{H}_1\mathbf{A}_0$, où $\mathbf{H}_1$ est une matrice de Householder qui transforme la première colonne de $\mathbf{A}_0$ en la première colonne de $\mathbf{A}_1$, dont tous les éléments sont nuls sauf le premier. La proposition 3.1 conduit à prendre

$$\mathbf{H}_1 = \mathbf{H}(\mathbf{a}^1 + \text{signe}(a_1^1)||\mathbf{a}^1||_2\mathbf{e}^1), \tag{3.75}$$

où $\mathbf{a}^1$ est la première colonne de $\mathbf{A}_0$.

Itérer pour obtenir

$$\mathbf{A}_{k+1} = \mathbf{H}_{k+1}\mathbf{A}_k, \quad k = 1, \cdots, n-2. \tag{3.76}$$

$\mathbf{H}_{k+1}$ a pour tâche de mettre en forme la $(k+1)$-ème colonne de $\mathbf{A}_k$ tout en laissant les $k$ colonnes à sa gauche inchangées. Soit $\mathbf{a}^{k+1}$ le vecteur formé par les $n-k$ derniers éléments de la $(k+1)$-ème colonne de $\mathbf{A}_k$. La transformation de Householder doit seulement modifier $\mathbf{a}^{k+1}$, de sorte que

$$\mathbf{H}_{k+1} = \begin{bmatrix} \mathbf{I}_k & \mathbf{0} \\ \mathbf{0} & \mathbf{H}\left(\mathbf{a}^{k+1} + \text{signe}(a_1^{k+1})\left\|\mathbf{a}^{k+1}\right\|_2\mathbf{e}^1\right) \end{bmatrix}. \tag{3.77}$$

Dans la prochaine équation, par exemple, les éléments du haut et du bas de $\mathbf{a}^3$ sont indiqués par le symbole $\times$ :

$$\mathbf{A}_3 = \begin{bmatrix} \bullet & \bullet & \bullet & \cdots & \bullet \\ 0 & \bullet & \bullet & \cdots & \bullet \\ \vdots & 0 & \times & \ddots & \vdots \\ \vdots & \vdots & \vdots & \bullet & \bullet \\ 0 & 0 & \times & \bullet & \bullet \end{bmatrix}. \tag{3.78}$$

Dans (3.77), $\mathbf{e}^1$ a la même dimension que $\mathbf{a}^{k+1}$, et tous ses éléments sont de nouveau égaux à zéro, sauf le premier qui est égal à un. A chaque itération, la matrice $\mathbf{H}_{(+)}$ ou $\mathbf{H}_{(-)}$ qui conduit au calcul le plus stable est sélectionnée, voir (3.72). Finalement

$$\mathbf{R} = \mathbf{H}_{n-1}\mathbf{H}_{n-2}\cdots\mathbf{H}_1\mathbf{A}, \tag{3.79}$$

ou bien, de façon équivalente,

$$\mathbf{A} = (\mathbf{H}_{n-1}\mathbf{H}_{n-2}\cdots\mathbf{H}_1)^{-1}\mathbf{R} = \mathbf{H}_1^{-1}\mathbf{H}_2^{-1}\cdots\mathbf{H}_{n-1}^{-1}\mathbf{R} = \mathbf{Q}\mathbf{R}. \tag{3.80}$$

Compte-tenu de (3.63), on a donc

$$\mathbf{Q} = \mathbf{H}_1\mathbf{H}_2\cdots\mathbf{H}_{n-1}. \tag{3.81}$$

Au lieu d'utiliser des transformations de Householder, on peut mettre en œuvre la factorisation QR via des *rotations de Givens* [49], qui sont aussi des transformations robustes et orthonormales, mais ceci rend les calculs plus complexes sans améliorer les performances.

### *3.6.6 Décomposition en valeurs singulières*

La décomposition en valeurs singulières (ou SVD, pour *Singular Value Decomposition*) [77] s'est révélée être l'une des idées les plus fructueuses de la théorie des matrices [224]. Bien qu'elle soit principalement utilisée sur des matrices rectangulaires (voir la section 9.2.4, où la procédure est expliquée de façon plus détaillée), elle peut aussi être appliquée à la matrice carrée $\mathbf{A}$, qu'elle transforme en un produit de trois matrices carrées

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}}. \tag{3.82}$$

$\mathbf{U}$ et $\mathbf{V}$ sont orthonormales, c'est à dire telles que

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I}, \tag{3.83}$$

ce qui rend leur inversion particulièrement aisée, puisque

$$\mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}} \quad \text{et} \quad \mathbf{V}^{-1} = \mathbf{V}^{\mathrm{T}}. \tag{3.84}$$

$\mathbf{\Sigma}$ est une matrice diagonale, dont les éléments diagonaux sont égaux aux valeurs singulières de $\mathbf{A}$. Le conditionnement de $\mathbf{A}$ pour la norme spectrale est donc trivial à évaluer à partir de la SVD. Dans ce chapitre, $\mathbf{A}$ est supposée inversible, ce qui implique qu'aucune valeur singulière n'est nulle et que $\mathbf{\Sigma}$ est inversible. En MATLAB, la SVD de `A` est évaluée par l'instruction `[U,S,V]=svd(A)`.

L'équation (3.1) se traduit par

$$\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}\mathbf{x} = \mathbf{b}, \tag{3.85}$$

de sorte que

$$\mathbf{x} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{b}, \tag{3.86}$$

avec $\mathbf{\Sigma}^{-1}$ triviale à évaluer puisque $\mathbf{\Sigma}$ est diagonale. Comme le calcul d'une SVD est significativement plus complexe que la factorisation QR, on peut préférer cette dernière.

Quand cond $\mathbf{A}$ est trop grand, il devient risqué de tenter de résoudre (3.1) en utilisant des nombres à virgule flottante, même via la factorisation QR. Une meilleure solution approximative peut alors *parfois* être obtenue en remplaçant (3.86) par

$$\widehat{\mathbf{x}} = \mathbf{V}\widehat{\mathbf{\Sigma}}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{b}, \tag{3.87}$$

où $\widehat{\mathbf{\Sigma}}^{-1}$ est une matrice diagonale telle que

$$\widehat{\mathbf{\Sigma}}^{-1}_{i,i} = \begin{cases} 1/\sigma_{i,i} & \text{si} \quad \sigma_{i,i} > \delta \\ 0 \quad \text{autrement} \end{cases}, \tag{3.88}$$

avec $\delta$ un seuil positif à choisir par l'utilisateur. Ceci revient à remplacer toute valeur singulière de $\mathbf{A}$ plus petite que $\delta$ par zéro, et à faire ainsi comme si (3.1) avait une infinité de solutions, puis à prendre la solution de norme euclidienne minimale. Voir la section 9.2.6 pour plus de détails sur cette approche par *régularisation* dans le contexte des moindres carrés. Une telle approche est cependant à utiliser avec précaution ici, car la qualité de la solution approximative $\widehat{\mathbf{x}}$ fournie par (3.87) dépend beaucoup de la valeur prise par $\mathbf{b}$. Supposons, par exemple, que $\mathbf{A}$ soit symétrique définie positive, et que $\mathbf{b}$ soit un vecteur propre de $\mathbf{A}$ associé à une *très petite* valeur propre $\lambda_{\mathrm{b}}$, telle que $\|\mathbf{b}\|_2 = 1$. La solution mathématique de (3.1)

$$\mathbf{x} = \frac{1}{\lambda_{\mathrm{b}}}\mathbf{b}, \tag{3.89}$$

a alors une norme euclidienne *très grande*, et devrait donc être complètement différente de $\widehat{\mathbf{x}}$, puisque la valeur propre $\lambda_{\mathrm{b}}$ est aussi une (très petite) valeur singulière de $\mathbf{A}$ et $1/\lambda_{\mathrm{b}}$ sera remplacé par zéro lors du calcul de $\widehat{\mathbf{x}}$. Des exemples de problèmes mal posés pour lesquels une régularisation par SVD donne des résultats intéressants sont dans [233].

## 3.7 Méthodes itératives

Dans des problèmes de très grande dimension, comme ceux impliqués dans la résolution d'équations aux dérivées partielles, $\mathbf{A}$ est typiquement creuse, ce qu'il faut exploiter. Les méthodes directes présentées dans la section 3.6 deviennent difficiles à utiliser, car le caractère creux est en général perdu lors de la factorisation de $\mathbf{A}$. On peut alors utiliser des *solveurs directs creux* (pas présentés ici), qui permutent des équations et des inconnues dans le but de minimiser l'apparition de termes non nuls dans les facteurs. Ceci est en soi un problème d'optimisation complexe, de sorte que les méthodes itératives forment une alternative intéressante [49], [201].

### *3.7.1 Méthodes itératives classiques*

Ces méthodes sont lentes et plus guère utilisées, mais simples à comprendre. Elles servent ici d'introduction aux méthodes itératives dans les sous-espaces de Krylov présentées en section 3.7.2.

#### 3.7.1.1 Principe

Pour résoudre (3.1) en $\mathbf{x}$, décomposons $\mathbf{A}$ en une somme de deux matrices

$$\mathbf{A} = \mathbf{A}_1 + \mathbf{A}_2, \tag{3.90}$$

avec $\mathbf{A}_1$ (facilement) inversible, de façon à assurer

$$\mathbf{x} = -\mathbf{A}_1^{-1}\mathbf{A}_2\mathbf{x} + \mathbf{A}_1^{-1}\mathbf{b}. \tag{3.91}$$

Définissons $\mathbf{M} = -\mathbf{A}_1^{-1}\mathbf{A}_2$ et $\mathbf{v} = \mathbf{A}_1^{-1}\mathbf{b}$ pour obtenir

$$\mathbf{x} = \mathbf{M}\mathbf{x} + \mathbf{v}. \tag{3.92}$$

L'idée est de choisir la décomposition (3.90) pour que la récurrence

$$\mathbf{x}^{k+1} = \mathbf{M}\mathbf{x}^k + \mathbf{v} \tag{3.93}$$

converge vers la solution de (3.1) quand $k$ tend vers l'infini. Tel sera le cas si et seulement si toutes les valeurs propres de $\mathbf{M}$ sont strictement à l'intérieur du cercle unité.

Les méthodes considérées ci-dessous diffèrent par leur décomposition de $\mathbf{A}$. Nous supposons que tous les éléments diagonaux de $\mathbf{A}$ sont non nuls, et écrivons

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}, \tag{3.94}$$

où **D** est une matrice diagonale inversible avec les mêmes éléments diagonaux que **A**, **L** est une matrice triangulaire inférieure avec des zéros sur la diagonale principale, et **U** est une matrice triangulaire supérieure, elle aussi avec des zéros sur sa diagonale principale.

#### 3.7.1.2 Itération de Jacobi

Dans l'itération de Jacobi, $\mathbf{A}_1 = \mathbf{D}$ et $\mathbf{A}_2 = \mathbf{L} + \mathbf{U}$, de sorte que

$$\mathbf{M} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) \quad \text{et} \quad \mathbf{v} = \mathbf{D}^{-1}\mathbf{b}. \tag{3.95}$$

L'interprétation scalaire de cette méthode est la suivante. La $j$-ème ligne de (3.1) est

$$\sum_{i=1}^{n} a_{j,i} x_i = b_j. \tag{3.96}$$

Puisque $a_{j,j} \neq 0$ par hypothèse, on peut la réécrire

$$x_j = \frac{b_j - \sum_{i \neq j} a_{j,i} x_i}{a_{j,j}}, \tag{3.97}$$

qui exprime $x_j$ en fonction des autres inconnues. Une itération de Jacobi calcule

$$x_j^{k+1} = \frac{b_j - \sum_{i \neq j} a_{j,i} x_i^k}{a_{j,j}}, \qquad j = 1, \cdots, n. \tag{3.98}$$

Une condition *suffisante* de convergence vers la solution $\mathbf{x}^\star$ de (3.1) (pour tout vecteur initial $\mathbf{x}^0$) est que **A** soit à diagonale dominante. Cette condition n'est pas nécessaire, et il peut y avoir convergence sous des conditions moins restrictives.

#### 3.7.1.3 Itération de Gauss-Seidel

Dans l'itération de Gauss-Seidel, $\mathbf{A}_1 = \mathbf{D} + \mathbf{L}$ et $\mathbf{A}_2 = \mathbf{U}$, de sorte que

$$\mathbf{M} = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U} \quad \text{et} \quad \mathbf{v} = (\mathbf{D} + \mathbf{L})^{-1}\mathbf{b}. \tag{3.99}$$

L'interprétation scalaire devient

$$x_j^{k+1} = \frac{b_j - \sum_{i=1}^{j-1} a_{j,i} x_i^{k+1} - \sum_{i=j+1}^{n} a_{j,i} x_i^k}{a_{j,j}}, \qquad j = 1, \cdots, n. \tag{3.100}$$

Notons la présence de $x_i^{k+1}$ dans le membre de droite de (3.100). Les éléments de $\mathbf{x}^{k+1}$ déjà évalués sont donc utilisés pour le calcul de ceux qui ne l'ont pas encore été. Ceci accélère la convergence et permet d'économiser de la place mémoire.

*Remarque 3.16.* Le comportement de la méthode de Gauss-Seidel dépend de l'ordre des variables dans $\mathbf{x}$, contrairement à ce qui se passe avec la méthode de Jacobi. □

Comme avec la méthode de Jacobi, une condition *suffisante* de convergence vers la solution $\mathbf{x}^\star$ de (3.1) (pour tout vecteur initial $\mathbf{x}^0$) est que $\mathbf{A}$ soit à diagonale dominante. Cette condition n'est pas nécessaire ici non plus, et il peut y avoir convergence sous des conditions moins restrictives.

#### 3.7.1.4 Méthode SOR

La méthode SOR (acronyme de *successive over-relaxation*) a été développée dans le contexte de la résolution d'équations aux dérivées partielles [256]. Elle réécrit (3.1) comme

$$(\mathbf{D}+\omega\mathbf{L})\mathbf{x} = \omega\mathbf{b} - [\omega\mathbf{U}+(\omega-1)\mathbf{D}]\mathbf{x}, \tag{3.101}$$

où $\omega \neq 0$ est le *facteur de relaxation*, et itère le calcul de $\mathbf{x}^{k+1}$ par résolution de

$$(\mathbf{D}+\omega\mathbf{L})\mathbf{x}^{k+1} = \omega\mathbf{b} - [\omega\mathbf{U}+(\omega-1)\mathbf{D}]\mathbf{x}^{k}. \tag{3.102}$$

Comme $\mathbf{D}+\omega\mathbf{L}$ est triangulaire inférieure, ceci est accompli par substitution avant, et revient à écrire

$$x_j^{k+1} = (1-\omega)x_j^k + \omega\frac{b_j - \sum_{i=1}^{j-1} a_{j,i}x_i^{k+1} - \sum_{i=j+1}^{n} a_{j,i}x_i^k}{a_{j,j}}, \quad j = 1,\cdots,n. \tag{3.103}$$

Globalement,

$$\mathbf{x}^{k+1} = (1-\omega)\mathbf{x}^k + \omega\mathbf{x}_{\mathrm{GS}}^{k+1}, \tag{3.104}$$

où $\mathbf{x}_{\mathrm{GS}}^{k+1}$ est l'approximation de la solution $\mathbf{x}^\star$ suggérée par l'itération de Gauss-Seidel.

Une condition nécessaire de convergence est que $\omega \in [0,2]$. Pour $\omega = 1$, on retombe sur la méthode de Gauss-Seidel. Quand $\omega < 1$ la méthode est sous-relaxée, tandis qu'elle est sur-relaxée si $\omega > 1$. La valeur optimale de $\omega$ dépend de $\mathbf{A}$, mais la sur-relaxation est en général préférée. Elle conduit à accroître la longueur des déplacements suggérés par la méthode de Gauss-Seidel. La convergence de la méthode de Gauss-Seidel peut ainsi être accélérée en extrapolant les résultats de ses itérations. Des méthodes sont disponibles pour adapter $\omega$ sur la base du comportement passé. Elles ont cependant largement perdu de leur intérêt avec l'arrivée des méthodes par itération dans les sous-espaces de Krylov.

### *3.7.2 Itération dans les sous-espaces de Krylov*

L'itération dans les sous-espaces de Krylov [89, 240] a donné un coup de vieux aux méthodes itératives classiques, qui peuvent se révéler très lentes ou même ne pas converger. Elle est qualifiée dans [55] de *l'un des dix algorithmes qui ont eu la plus grande influence dans le développement et la pratique de la science et de l'ingénierie au 20-ème siècle*.

#### 3.7.2.1 De Jacobi à Krylov

L'itération de Jacobi peut s'écrire

$$\mathbf{x}^{k+1} = -\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}^k + \mathbf{D}^{-1}\mathbf{b}. \tag{3.105}$$

L'équation (3.94) implique que $\mathbf{L}+\mathbf{U} = \mathbf{A}-\mathbf{D}$, de sorte que

$$\mathbf{x}^{k+1} = (\mathbf{I}-\mathbf{D}^{-1}\mathbf{A})\mathbf{x}^k + \mathbf{D}^{-1}\mathbf{b}. \tag{3.106}$$

Puisque la vraie solution $\mathbf{x}^\star = \mathbf{A}^{-1}\mathbf{b}$ est inconnue, l'erreur

$$\delta\mathbf{x}^k = \mathbf{x}^k - \mathbf{x}^\star \tag{3.107}$$

ne peut pas être calculée. Pour caractériser la qualité de la solution approximative obtenue jusqu'ici, on remplace donc cette erreur par le résidu

$$\mathbf{r}^k = \mathbf{b} - \mathbf{A}\mathbf{x}^k = -\mathbf{A}(\mathbf{x}^k - \mathbf{x}^\star) = -\mathbf{A}\delta\mathbf{x}^k. \tag{3.108}$$

Normalisons le système d'équations à résoudre pour le rendre tel que $\mathbf{D} = \mathbf{I}$. Alors

$$\begin{aligned} \mathbf{x}^{k+1} &= (\mathbf{I}-\mathbf{A})\mathbf{x}^k + \mathbf{b} \\ &= \mathbf{x}^k + \mathbf{r}^k. \end{aligned} \tag{3.109}$$

Soustrayons $\mathbf{x}^\star$ des deux membres de (3.109), et multiplions le résultat par $-\mathbf{A}$ pour obtenir

$$\mathbf{r}^{k+1} = \mathbf{r}^k - \mathbf{A}\mathbf{r}^k. \tag{3.110}$$

L'équation de récurrence (3.110) implique que $\mathbf{r}^k$ est une combinaison linéaire des vecteurs $\mathbf{r}^0, \mathbf{A}\mathbf{r}^0, \cdots, \mathbf{A}^k\mathbf{r}^0$ :

$$\mathbf{r}^k \in \text{Vect}\{\mathbf{r}^0, \mathbf{A}\mathbf{r}^0, \cdots, \mathbf{A}^k\mathbf{r}^0\}, \tag{3.111}$$

et (3.109) implique alors que

$$\mathbf{x}^k - \mathbf{x}^0 = \sum_{i=0}^{k-1} \mathbf{r}^i. \tag{3.112}$$

On a donc

$$\mathbf{x}^k \in \mathbf{x}^0 + \text{Vect}\{\mathbf{r}^0, \mathbf{A}\mathbf{r}^0, \cdots, \mathbf{A}^{k-1}\mathbf{r}^0\}, \tag{3.113}$$

où $\text{Vect}\{\mathbf{r}^0, \mathbf{A}\mathbf{r}^0, \cdots, \mathbf{A}^{k-1}\mathbf{r}^0\}$ est le $k$-ème *sous-espace de Krylov* généré par $\mathbf{A}$ à partir de $\mathbf{r}^0$, noté $\mathscr{K}_k(\mathbf{A}, \mathbf{r}^0)$.

*Remarque 3.17.* La définition des sous-espaces de Krylov implique que

$$\mathscr{K}_{k-1}(\mathbf{A}, \mathbf{r}^0) \subset \mathscr{K}_k(\mathbf{A}, \mathbf{r}^0), \tag{3.114}$$

et que chaque itération augmente la dimension de l'espace de recherche *au plus* d'une unité. Supposons, par exemple, que $\mathbf{x}^0 = \mathbf{0}$, ce qui implique que $\mathbf{r}^0 = \mathbf{b}$. Supposons de plus que $\mathbf{b}$ soit un vecteur propre de $\mathbf{A}$ tel que

$$\mathbf{A}\mathbf{b} = \lambda \mathbf{b}. \tag{3.115}$$

Alors

$$\forall k \geqslant 1, \quad \text{Vect}\{\mathbf{r}^0, \mathbf{A}\mathbf{r}^0, \cdots, \mathbf{A}^{k-1}\mathbf{r}^0\} = \text{Vect}\{\mathbf{b}\}. \tag{3.116}$$

Ceci est bien adapté, puisque la solution est $\mathbf{x} = \lambda^{-1}\mathbf{b}$. □

*Remarque 3.18.* Soit $P_n(\lambda)$ le polynôme caractéristique de $\mathbf{A}$,

$$P_n(\lambda) = \det(\mathbf{A} - \lambda \mathbf{I}_n). \tag{3.117}$$

D'après le théorème de Cayley-Hamilton, $P_n(\mathbf{A})$ est la matrice $n \times n$ dont tous les éléments sont nuls. $\mathbf{A}^n$ est donc une combinaison linéaire de $\mathbf{A}^{n-1}, \mathbf{A}^{n-2}, \cdots, \mathbf{I}_n$, de sorte que

$$\forall k \geqslant n, \quad \mathscr{K}_k(\mathbf{A}, \mathbf{r}^0) = \mathscr{K}_n(\mathbf{A}, \mathbf{r}^0). \tag{3.118}$$

La dimension de l'espace dans lequel la recherche prend place ne peut donc plus croître après les $n$ premières itérations. □

Un point crucial, non prouvé ici, est qu'il existe $\nu \leqslant n$ tel que

$$\mathbf{x}^\star \in \mathbf{x}^0 + \mathscr{K}_\nu(\mathbf{A}, \mathbf{r}^0). \tag{3.119}$$

En principe, on peut donc espérer obtenir la solution en au plus $n = \dim \mathbf{x}$ itérations dans les sous-espaces de Krylov, alors qu'une telle borne n'existe pas pour les itérations de Jacobi, Gauss-Seidel ou SOR. En pratique, avec des calculs à virgule flottante, on peut encore obtenir de meilleurs résultats en itérant jusqu'à ce que la solution soit jugée satisfaisante.

#### 3.7.2.2 A est symétrique définie positive

Quand $\mathbf{A} \succ \mathbf{0}$, la famille des *méthodes de gradients conjugués* [97], [78], [218] reste à ce jour la référence. La solution approximative est recherchée en minimisant

$$J(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x}. \tag{3.120}$$

En utilisant les conditions théoriques d'optimalité présentées en section 9.1, il est facile de montrer que le seul minimiseur de cette fonction de coût est en effet $\widehat{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{b}$. Partant de $\mathbf{x}^k$, l'approximation de $\mathbf{x}^\star$ à l'itération $k$, $\mathbf{x}^{k+1}$ est calculé par recherche à une dimension dans une direction $\mathbf{d}^k$ comme

$$\mathbf{x}^{k+1}(\lambda_k) = \mathbf{x}^k + \lambda_k \mathbf{d}^k. \tag{3.121}$$

Il est là aussi facile de montrer que $J(\mathbf{x}^{k+1}(\lambda_k))$ est minimal si

$$\lambda_k = \frac{(\mathbf{d}^k)^{\mathrm{T}}(\mathbf{b} - \mathbf{A}\mathbf{x}^k)}{(\mathbf{d}^k)^{\mathrm{T}}\mathbf{A}\mathbf{d}^k}. \tag{3.122}$$

La direction de recherche $\mathbf{d}^k$ est choisie pour assurer que

$$(\mathbf{d}^i)^{\mathrm{T}}\mathbf{A}\mathbf{d}^k = 0, \quad i = 0, \cdots, k-1, \tag{3.123}$$

ce qui veut dire qu'elle est conjuguée par rapport à $\mathbf{A}$ (ou $\mathbf{A}$-orthogonale) avec toutes les directions de recherche précédentes. Si les calculs étaient exacts, ceci assurerait la convergence vers $\widehat{\mathbf{x}}$ en $n$ itérations au plus. A cause de l'effet des erreurs d'arrondi, il peut s'avérer utile d'autoriser plus de $n$ itérations, quoique $n$ puisse être si large que $n$ itérations soit en fait plus que ce qui est possible. (On obtient souvent une approximation utile de la solution en moins de $n$ itérations.)

Après $n$ itérations,

$$\mathbf{x}^n = \mathbf{x}^0 + \sum_{i=0}^{n-1} \lambda_i \mathbf{d}^i, \tag{3.124}$$

de sorte que

$$\mathbf{x}^n \in \mathbf{x}^0 + \mathrm{Vect}\{\mathbf{d}^0, \cdots, \mathbf{d}^{n-1}\}. \tag{3.125}$$

Un solveur dans les espaces de Krylov est obtenu si les directions de recherche sont telles que

$$\mathrm{Vect}\{\mathbf{d}^0, \cdots, \mathbf{d}^i\} = \mathscr{K}_{i+1}(\mathbf{A}, \mathbf{r}^0) \qquad i = 0, 1, \cdots \tag{3.126}$$

Ceci peut être accompli avec un algorithme étonnamment simple [97], [218], résumé par le tableau 3.1. Voir aussi la section 9.3.4.6 et l'exemple 9.8.

*Remarque 3.19.* La notation := dans le tableau 3.1 signifie que la variable du membre de gauche se voit assigner la valeur qui résulte de l'évaluation du membre de droite. Elle ne doit pas être confondue avec le signe égal, et on peut écrire $k := k+1$ alors que $k = k+1$ n'aurait aucun sens. Ceci dit, MATLAB, comme d'autres langages de programmation, utilise le signe = au lieu de :=. □

**Tableau 3.1** Solveur dans les espaces de Krylov

$$
\begin{aligned}
\mathbf{r}^0 &:= \mathbf{b} - \mathbf{A}\mathbf{x}^0, \\
\mathbf{d}^0 &:= \mathbf{r}^0, \\
\delta^0 &:= \|\mathbf{r}^0\|_2^2, \\
k &:= 0.
\end{aligned}
$$

Tant que $||\mathbf{r}^k||_2 >$ seuil, calculer

$$
\begin{aligned}
\delta'_k &:= (\mathbf{d}^k)^{\mathrm{T}}\mathbf{A}\mathbf{d}^k, \\
\lambda_k &:= \delta_k/\delta'_k, \\
\mathbf{x}^{k+1} &:= \mathbf{x}^k + \lambda_k\mathbf{d}^k, \\
\mathbf{r}^{k+1} &:= \mathbf{r}^k - \lambda_k\mathbf{A}\mathbf{d}^k, \\
\delta_{k+1} &:= \|\mathbf{r}^{k+1}\|_2^2, \\
\beta_k &:= \delta_{k+1}/\delta_k, \\
\mathbf{d}^{k+1} &:= \mathbf{r}^{k+1} + \beta_k\mathbf{d}^k, \\
k &:= k+1.
\end{aligned}
$$

#### 3.7.2.3 A n'est pas symétrique définie positive

C'est une situation beaucoup plus compliquée et plus coûteuse. Des méthodes spécifiques, non détaillées ici, ont été développées pour les matrices symétriques qui ne sont pas définies positives [175], ainsi que pour des matrices non symétriques [203], [239].

#### 3.7.2.4 Préconditionnement

La vitesse de convergence d'un solveur de Krylov dépend fortement du conditionnement de $\mathbf{A}$. Une accélération spectaculaire peut être obtenue en remplaçant (3.1) par

$$\mathbf{MAx} = \mathbf{Mb}, \tag{3.127}$$

où $\mathbf{M}$ est une matrice de préconditionnement convenablement choisie, et de nombreux travaux de recherche ont été consacrés à ce thème [11], [200]. Les méthodes de Krylov préconditionnées qui en résultent convergent beaucoup plus vite et pour des classes de matrices beaucoup plus générales que les méthodes itératives classiques de la section 3.7.1.

Une des approches possibles pour choisir $\mathbf{M}$ est de rechercher une approximation creuse de l'inverse de $\mathbf{A}$ en résolvant

$$\widehat{\mathbf{M}} = \arg\min_{\mathbf{M}\in\mathbb{S}} \|\mathbf{I}_n - \mathbf{AM}\|_{\mathrm{F}}, \tag{3.128}$$

où $\|\cdot\|_{\mathrm{F}}$ est la norme de Frobenius et $\mathbb{S}$ est un ensemble de matrices creuses à préciser. Puisque

$$\|\mathbf{I}_n - \mathbf{AM}\|_{\mathrm{F}}^2 = \sum_{j=1}^{n} \|\mathbf{e}^j - \mathbf{Am}^j\|_2^2, \tag{3.129}$$

où $\mathbf{e}^j$ est la $j$-ème colonne de $\mathbf{I}_n$ et $\mathbf{m}^j$ la $j$-ème colonne de $\mathbf{M}$, le calcul de $\mathbf{M}$ peut être décomposé en $n$ problèmes de moindres carrés indépendants (un par colonne), sous des contraintes spécifiant le caractère creux de $\mathbf{M}$. Les éléments non nuls de $\mathbf{m}^j$ sont alors obtenus en résolvant un *petit* problème de moindres carrés linéaires *sans contrainte* (voir la section 9.2). Le calcul des colonnes de $\widehat{\mathbf{M}}$ est donc facile à paralléliser. La difficulté principale est le choix d'un bon ensemble $\mathbb{S}$, qui peut être mené avec une stratégie adaptative [86]. On peut commencer avec $\mathbf{M}$ diagonale, ou ayant la même répartition de ses éléments nuls que $\mathbf{A}$.

*Remarque 3.20.* Le préconditionnement peut aussi être utilisé avec des méthodes directes. □

## 3.8 Tirer profit de la structure de A

Cette section décrit des cas particuliers importants où la structure de $\mathbf{A}$ suggère des algorithmes dédiés, comme en section 3.7.2.2.

### *3.8.1* A *est symétrique définie positive*

Quand $\mathbf{A}$ est réelle, symétrique et *définie positive*, c'est à dire quand

$$\forall \mathbf{v} \neq \mathbf{0}, \quad \mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} > 0, \tag{3.130}$$

sa factorisation LU est particulièrement aisée, car il existe une unique matrice triangulaire inférieur $\mathbf{L}$ telle que

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}, \tag{3.131}$$

avec $l_{k,k} > 0$ pour tout $k$ ($l_{k,k}$ n'est plus pris égal à 1). Ainsi $\mathbf{U} = \mathbf{L}^{\mathrm{T}}$, et nous pourrions tout aussi bien écrire

$$\mathbf{A} = \mathbf{U}^{\mathrm{T}}\mathbf{U}. \tag{3.132}$$

Cette factorisation, connue comme la *factorisation de Cholesky* [102], est facile à calculer par identification des deux membres de (3.131). Aucun pivotage n'est nécessaire, car les éléments de $\mathbf{L}$ doivent satisfaire

$$\sum_{i=1}^{k} r_{i,k}^2 = a_{k,k}, \quad k = 1, \cdots, n, \tag{3.133}$$

et sont donc bornés. Comme la factorisation de Cholesky échoue si $\mathbf{A}$ n'est pas définie positive, elle peut aussi être utilisée pour tester si des matrices symétriques le sont, ce qui est préférable au calcul des valeurs propres de $\mathbf{A}$. En MATLAB, on peut utiliser `U=chol(A)` ou `L=chol(A,'lower')`.

Quand $\mathbf{A}$ est aussi grande et creuse, voir la section 3.7.2.2.

### 3.8.2 A *est une matrice de Toeplitz*

Quand tous les éléments d'une même diagonale descendante de **A** ont la même valeur, c'est à dire quand

$$\mathbf{A} = \begin{bmatrix} h_0 & h_{-1} & h_{-2} & \cdots & h_{-n+1} \\ h_1 & h_0 & h_{-1} & & h_{-n+2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ h_{n-2} & & \ddots & h_0 & h_{-1} \\ h_{n-1} & h_{n-2} & \cdots & h_1 & h_0 \end{bmatrix}, \tag{3.134}$$

comme dans les problèmes de déconvolution, **A** est une matrice de Toeplitz. On peut alors utiliser l'algorithme de *Levinson-Durbin* (pas présenté ici) pour obtenir des solutions récursivement sur la dimension $n$ du vecteur solution $\mathbf{x}^n$, où $\mathbf{x}^n$ est exprimé en fonction de $\mathbf{x}^{n-1}$.

### 3.8.3 A *est une matrice de Vandermonde*

Quand

$$\mathbf{A} = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^{n-1} \\ 1 & t_2 & t_2^2 & \cdots & t_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^{n-1} \end{bmatrix}, \tag{3.135}$$

on dit que c'est une matrice de Vandermonde. De telles matrices, qu'on rencontre par exemple en interpolation polynomiale, deviennent vite mal conditionnées quand $n$ croît, ce qui requiert des méthodes numériques robustes ou une reformulation du problème pour éviter tout emploi d'une matrice de Vandermonde.

### 3.8.4 A *est creuse*

**A** est creuse quand la plupart de ses éléments sont nuls. Ceci est particulièrement fréquent quand on discrétise une équation aux dérivées partielles, puisque chaque nœud de discrétisation n'est influencé que par ses proches voisins. Au lieu de stocker tous les éléments de **A**, on peut alors utiliser des descriptions plus économiques, comme une liste de paires {adresse, valeur} ou une liste de vecteurs décrivant la partie non nulle de **A**, comme illustré par l'exemple qui suit.

*Exemple 3.5. Systèmes tridiagonaux*

Quand

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix}, \tag{3.136}$$

les éléments non nuls de $\mathbf{A}$ peuvent être stockés dans trois vecteurs $\mathbf{a}$, $\mathbf{b}$ et $\mathbf{c}$ (un par diagonale descendante non nulle). Ceci permet d'économiser de la mémoire qui aurait été utilisée de façon inutile pour stocker les éléments nuls de $\mathbf{A}$. La factorisation LU devient alors extraordinairement simple avec l'algorithme de Thomas [42]. □

La façon dont MATLAB gère les matrices creuses est expliquée dans [70]. Un point critique quand on résout des systèmes de grande taille est la façon dont les éléments non nuls de $\mathbf{A}$ sont stockés. Des choix malvenus peuvent se traduire par des échanges intenses avec la mémoire disque, ce qui peut ralentir l'exécution de plusieurs ordres de grandeur. Des algorithmes (non présentés ici) sont disponibles pour réordonner les éléments de matrices creuses de façon automatique.

## 3.9 Enjeux de complexité

Une première mesure naturelle de la complexité d'un algorithme est le nombre des opérations requises.

### *3.9.1 Compter les flops*

En général, on se borne à compter les opérations à virgule flottante (ou *flops*). Pour les algorithmes finis, le comptage des flops est juste une question de comptabilité.

*Exemple 3.6.* Multiplier deux matrices $n \times n$ génériques requiert $\mathrm{O}(n^3)$ flops ; multiplier une matrice $n \times n$ générique par un vecteur générique ne requiert que $\mathrm{O}(n^2)$ flops. □

*Exemple 3.7.* Pour résoudre un système triangulaire supérieur avec l'algorithme de la section 3.6.1, il faut un flop pour obtenir $x_n$ par (3.31), trois flops de plus pour obtenir $x_{n-1}$ par (3.32), $\cdots$, et $2n-1$ flops supplémentaires pour obtenir $x_1$ par (3.33). Le nombre total de flops est donc

$$1 + 3 + \cdots + (2n-1) = n^2. \tag{3.137}$$

□

*Exemple 3.8.* Quand **A** est tridiagonale, (3.1) peut être résolue avec l'algorithme de Thomas (un cas particulier de la factorisation LU) en $8n-6$ flops [42]. □

Pour une matrice **A** générique $n \times n$, le nombre de flops requis pour résoudre un système d'équations linéaires se révèle beaucoup plus grand que dans les exemples 3.7 et 3.8 :

— la factorisation LU requiert $2n^3/3$ flops. La résolution de chacun des deux systèmes triangulaires résultants pour obtenir la solution pour un seul membre de droite demande environ $n^2$ flops supplémentaires, de sorte que le nombre total de flops pour $m$ membres de droite est environ $(2n^3/3)+2mn^2$.
— la factorisation QR requiert $4n^3/3$ flops, et le nombre total de flops pour $m$ membres de droite est $(4n^3/3)+3mn^2$.
— la SVD requiert $(20n^3/3)+\mathrm{O}(n^2)$ flops [49].

*Remarque 3.21.* Pour une matrice **A** générique $n \times n$, les factorisations LU, QR et SVD requièrent donc toutes $\mathrm{O}(n^3)$ flops. On peut cependant les classer du point de vue du nombre de flops requis, avec LU $<$ QR $<$ SVD. Pour de petits problèmes, chacune de ces factorisations est de toute façon obtenue très rapidement, de sorte que ces enjeux de complexité ne deviennent significatifs que pour des problèmes de grande taille (ou des problèmes résolus à de nombreuses reprises par des algorithmes itératifs). □

Quand **A** est symétrique définie positive, la factorisation de Cholesky s'applique, et ne requiert que $n^3/3$ flops. Le nombre total de flops pour $m$ membres de droite devient alors $(n^3/3)+2mn^2$.

Le nombre de flops requis par des méthodes itératives dépend du degré de creux de **A**, de la vitesse de convergence de ces méthodes (qui dépend elle-même du problème considéré) et du degré d'approximation qu'on est prêt à tolérer dans la résolution. Pour les solveurs de Krylov, le nombre maximum d'itérations requis pour obtenir une solution exacte en l'absence d'erreurs d'arrondi est connu et égal à $\dim \mathbf{x}$. C'est un avantage considérable sur les méthodes itératives classiques.

### *3.9.2 Faire faire le travail rapidement*

Pour un système linéaire de grande taille, comme on en rencontre dans de vraies applications, le nombre de flops n'est qu'un ingrédient parmi d'autres pour déterminer le temps nécessaire pour arriver à une solution, car faire entrer et sortir les données pertinentes des unités arithmétiques peut prendre plus de temps que l'exécution des flops. Il faut noter que la mémoire de l'ordinateur est intrinsèquement unidimensionnelle, tandis que **A** a deux dimensions. La façon dont les tableaux à deux dimensions sont transformées en objets à une dimension pour tenir compte de ce fait dépend du langage utilisé. FORTRAN, MATLAB, Octave,

R et Scilab, par exemple, stockent les matrices denses par colonnes, tandis que C et Pascal les stockent par lignes. Pour les matrices creuses, la situation est encore plus diverse.

La connaissance et l'exploitation de la façon dont les tableaux sont stockés permettent d'accélérer les algorithmes, car l'accès à des éléments contigus est rendu beaucoup plus rapide par l'utilisation de mémoire cache.

Quand on utilise un langage interprété à base de matrices, comme MATLAB, Octave ou Scilab, il faut éviter, chaque fois que possible, de décomposer des opérations telles que (2.1) sur des matrices génériques en des opérations sur les éléments de ces matrices comme dans (2.2) car ceci ralentit considérablement les calculs.

*Exemple 3.9.* Soient $\mathbf{v}$ et $\mathbf{w}$ deux vecteurs choisis au hasard dans $\mathbb{R}^n$. Le calcul de leur produit scalaire $\mathbf{v}^{\mathrm{T}}\mathbf{w}$ par décomposition en une somme de produit d'éléments, comme dans le script

```
vTw = 0;
for i=1:n,
    vTw = vTw + v(i)*w(i);
end
```

prend plus de temps que son calcul par

```
vTw = v'*w;
```

Sur un MacBook Pro avec un processeur 2.4 GHz Intel Core 2 Duo et 4 Go de RAM, qui sera toujours utilisé pour les mesures de temps de calcul, la première méthode prend environs 8 s pour $n = 10^6$, tandis que la seconde demande environ 0.004 s, de sorte qu'elle est à peu près 2000 fois plus rapide. □

La possibilité de modifier la taille d'une matrice $\mathbf{M}$ à chaque itération s'avère elle aussi coûteuse. Chaque fois que possible, il est beaucoup plus efficace de créer un tableau de taille appropriée une fois pour toute en incluant dans le script MATLAB une instruction comme `M=zeros(nr,nc);`, où `nr` est un nombre fixé de lignes et `nc` un nombre fixé de colonnes.

Quand on tente de réduire les temps de calcul en utilisant des processeurs graphiques (ou GPU, pour *Graphical Processing Units*) comme accélérateurs, il faut se rappeler que le rythme avec lequel le bus transfère des nombres de ou vers un GPU est beaucoup plus lent que le rythme auquel ce GPU peut les traiter, et organiser les transferts de données en conséquence.

Avec les ordinateurs personnels multicœurs, les accélérateurs GPU, les processeurs embarqués à grand nombre de cœurs, les clusters, les grilles et les supercalculateurs massivement parallèles, le paysage du calcul numérique n'a jamais été aussi divers, mais la question de Gene Golub et Charles Van Loan [80] demeure :

> *Pouvons-nous occuper les unités arithmétiques ultrarapides en leur livrant suffisamment de données sur des matrices et pouvons-nous réexpédier les résultats vers la mémoire suffisamment vite pour éviter de prendre du retard ?*

## 3.10 Exemples MATLAB

En présentant des scripts courts et leurs résultats, cette section montre combien il est facile d'expérimenter avec certaines des méthodes décrites. Des sections avec le même titre et le même but suivront dans la plupart des chapitres. Elles ne peuvent remplacer un bon tutoriel sur MATLAB, comme il y en a beaucoup. Les noms donnés aux variables suffisent, espérons-le, à expliquer leur rôle. Par exemple, la variable `A` correspond à la matrice **A**.

### *3.10.1* **A** *est dense*

MATLAB offre plusieurs options pour résoudre (3.1). La plus simple utilise l'élimination Gaussienne, et est mise en œuvre par

```
xGE = A\b;
```

Aucune factorisation de **A** n'est alors disponible pour un usage ultérieur, par exemple pour résoudre (3.1) avec la même matrice **A** et un autre vecteur **b**.

On peut préférer choisir une factorisation et l'utiliser. Pour une factorisation LU avec pivotage partiel, on peut écrire

```
[L,U,P] = lu(A);
% Même échange de lignes dans b et A
Pb = P*b;

% Résoudre Ly = Pb, avec L triangulaire inférieure
opts_LT.LT = true;
y = linsolve(L,Pb,opts_LT);

% Résoudre Ux = y, avec U triangulaire supérieure
opts_UT.UT = true;
xLUP = linsolve(U,y,opts_UT);
```

ce qui donne accès à la factorisation de **A** effectuée. Une version en une ligne avec le même résultat serait

```
xLUP = linsolve(A,b);
```

mais **L**, **U** et **P** ne seraient alors plus disponibles pour un usage ultérieur.

Pour une factorisation QR, on peut écrire

```
[Q,R] = qr(A);
QTb = Q'*b;
opts_UT.UT = true;
x_QR = linsolve(R,QTb,opts_UT);
```

et pour une factorisation SVD

```
[U,S,V] = svd(A);
xSVD = V*inv(S)*U'*b;
```

Pour une solution par itération de Krylov, on peut utiliser la fonction `gmres`, qui n'impose pas à **A** d'être symétrique définie positive [203], et écrire

```
xKRY = gmres(A,b);
```

Bien que l'itération de Krylov soit particulièrement intéressante quand **A** est grande et creuse, rien n'interdit de l'utiliser sur une petite matrice dense, comme dans ce qui suit.

Ces cinq méthodes sont utilisées pour résoudre (3.1) avec

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9+\alpha \end{bmatrix} \tag{3.138}$$

et

$$\mathbf{b} = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix}, \tag{3.139}$$

ce qui se traduit par

```
A = [1, 2, 3
     4, 5, 6
     7, 8, 9 + alpha];
b = [10; 11; 12];
```

**A** est alors singulière pour $\alpha = 0$, et son conditionnement s'améliore quand $\alpha$ augmente. Pour tout $\alpha > 0$, il est facile de vérifier que la solution exacte est unique et donnée par

$$\mathbf{x} = \begin{bmatrix} -28/3 \\ 29/3 \\ 0 \end{bmatrix} \approx \begin{bmatrix} -9.3333333333333333 \\ 9.6666666666666667 \\ 0 \end{bmatrix}. \tag{3.140}$$

Le fait que $x_3 = 0$ explique pourquoi **x** est indépendant de la valeur numérique prise par $\alpha$. Par contre, la difficulté de calculer **x** avec précision dépend bien de cette valeur. Dans tous les résultats présentés dans la suite de ce chapitre, le conditionnement utilisé est celui associé à la norme spectrale.

Pour $\alpha = 10^{-13}$, cond $\mathbf{A} \approx 10^{15}$ et

```
xGE =
    -9.297539149888147e+00
     9.595078299776288e+00
     3.579418344519016e-02

xLUP =
    -9.297539149888147e+00
     9.595078299776288e+00
```

```
      3.579418344519016e-02

xQR =
    -9.553113553113528e+00
     1.010622710622708e+01
    -2.197802197802198e-01

xSVD =
    -9.625000000000000e+00
     1.025000000000000e+01
    -3.125000000000000e-01

gmres converged at iteration 2 to a solution
with relative residual 9.9e-15.

xKRY =
    -4.555555555555692e+00
     1.111111111110619e-01
     4.777777777777883e+00
```

La factorisation LU avec pivotage partiel se révèle avoir produit un meilleur résultat que la factorisation QR ou la SVD sur ce problème mal conditionné, avec moins de calculs. Les conditionnements des matrices impliquées sont comme suit

```
CondA = 1.033684444145846e+15

% Factorisation LU avec pivotage partiel
CondL = 2.055595570492287e+00
CondU = 6.920247514139799e+14

% Factorisation QR
CondP = 1
CondQ = 1.000000000000000e+00
CondR = 1.021209931367105e+15

% SVD
CondU = 1.000000000000001e+00
CondS = 1.033684444145846e+15
CondV = 1.000000000000000e+00
```

Pour $\alpha = 10^{-5}$, cond $\mathbf{A} \approx 10^7$ et

```
xGE =
    -9.333333332978063e+00
     9.666666665956125e+00
     3.552713679092771e-10
```

```
xLUP =
    -9.333333332978063e+00
     9.666666665956125e+00
     3.552713679092771e-10

xQR =
    -9.333333335508891e+00
     9.666666671017813e+00
    -2.175583929062594e-09

xSVD =
    -9.333333335118368e+00
     9.666666669771075e+00
    -1.396983861923218e-09

gmres converged at iteration 3 to a solution
with relative residual 0.

xKRY =
    -9.333333333420251e+00
     9.666666666840491e+00
    -8.690781427844740e-11
```

Les conditionnements des matrices impliquées sont

```
CondA =  1.010884565427633e+07

% Factorisation LU avec pivotage partiel
CondL = 2.055595570492287e+00
CondU = 6.868613692978372e+06

% Factorisation QR
CondP = 1
CondQ = 1.000000000000000e+00
CondR = 1.010884565403081e+07

% SVD
CondU = 1.000000000000000e+00
CondS = 1.010884565427633e+07
CondV = 1.000000000000000e+00
```

Pour $\alpha = 1$, cond $\mathbf{A} \approx 88$ et

```
xGE =
    -9.333333333333330e+00
     9.666666666666661e+00
     3.552713678800503e-15
```

```
xLUP =
    -9.333333333333330e+00
     9.666666666666661e+00
     3.552713678800503e-15

xQR =
    -9.333333333333329e+00
     9.666666666666687e+00
    -2.175583928816833e-14

xSVD =
    -9.333333333333286e+00
     9.666666666666700e+00
    -6.217248937900877e-14

gmres converged at iteration 3 to a solution
with relative residual 0.

xKRY =
    -9.333333333333339e+00
     9.666666666666659e+00
     1.021405182655144e-14
```

Les conditionnements des matrices impliquées sont

```
CondA = 8.844827992069874e+01

% Factorisation LU avec pivotage partiel
CondL = 2.055595570492287e+00
CondU = 6.767412723516705e+01

% Factorisation QR
CondP = 1
CondQ = 1.000000000000000e+00
CondR = 8.844827992069874e+01

% SVD
CondU = 1.000000000000000e+00
CondS = 8.844827992069871e+01
CondV =1.000000000000000e+00
```

Les résultats `xGE` et `xLUP` sont toujours identiques, ce qui rappelle que la factorisation LU avec pivotage partielle n'est qu'une mise en œuvre astucieuse de l'élimination gaussienne. Mieux le problème est conditionné et plus les résultats des cinq méthodes se rapprochent. Bien que le produit des conditionnements de $\mathbf{L}$ et $\mathbf{U}$ soit légèrement plus grand que cond $\mathbf{A}$, la factorisation LU avec pivotage par-

tiel (ou l'élimination gaussienne) se révèlent ici donner de meilleurs résultats que la factorisation QR ou la SVD, pour moins de calculs.

### *3.10.2* $\mathbf{A}$ *est dense et symétrique définie positive*

Remplaçons maintenant $\mathbf{A}$ par $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ et $\mathbf{b}$ par $\mathbf{A}^{\mathrm{T}}\mathbf{b}$, avec $\mathbf{A}$ donnée par (3.138) et $\mathbf{b}$ par (3.139). La solution exacte reste la même que dans la section 3.10.1, mais $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ est symétrique définie positive.

*Remarque 3.22.* Multiplier à gauche (3.1) par $\mathbf{A}^{\mathrm{T}}$, comme ici, pour obtenir une matrice symétrique définie positive n'est pas à recommander. Cela détériore le conditionnement du système à résoudre, puisque cond $(\mathbf{A}^{\mathrm{T}}\mathbf{A}) = (\text{cond } \mathbf{A})^2$. □

`A` et `b` sont maintenant générés comme suit

```
A=[66,78,90+7*alpha
   78,93,108+8*alpha
   90+7*alpha,108+8*alpha,45+(9+alpha)^2];
b=[138; 171; 204+12*alpha];
```

La solution via une factorisation de Cholesky est obtenue par le script

```
L = chol(A,'lower');
opts_LT.LT = true;
y = linsolve(L,b,opts_LT);
opts_UT.UT = true;
xCHOL = linsolve(L',y,opts_UT)
```

Pour $\alpha = 10^{-13}$, l'évaluation de la valeur de cond $(\mathbf{A}^{\mathrm{T}}\mathbf{A})$ donne $3.8 \cdot 10^{16}$. Ceci est *très* optimiste (sa vraie valeur est environ $10^{30}$, ce qui laisse fort peu d'espoir d'une solution précise). On ne doit donc pas être surpris des mauvais résultats :

```
xCHOL =
    -5.777777777777945e+00
     2.555555555555665e+00
     3.555555555555555e+00
```

Pour $\alpha = 10^{-5}$, cond $(\mathbf{A}^{\mathrm{T}}\mathbf{A}) \approx 10^{14}$. Les résultats deviennent

```
xCHOL =
    -9.333013445827577e+00
     9.666026889522668e+00
     3.198891051102285e-04
```

Pour $\alpha = 1$, cond $(\mathbf{A}^{\mathrm{T}}\mathbf{A}) \approx 7823$. On obtient alors

```
xCHOL =
    -9.333333333333131e+00
     9.666666666666218e+00
     2.238209617644460e-13
```

### *3.10.3* **A** *est creuse*

`A` et `sA`, qui jouent le rôle de la matrice creuse (et asymétrique) **A**, sont construites par le script

```
n = 1.e3
A = eye(n); % matrice identité 1000 par 1000
A(1,n) = 1+alpha;
A(n,1) = 1; % maintenant légèrement modifiée
sA = sparse(A);
```

Ainsi, $\dim \mathbf{x} = 1000$, et `sA` est une représentation creuse de **A** dans laquelle les zéros ne sont pas stockés, tandis que `A` est une représentation dense d'une matrice creuse, qui comporte $10^6$ éléments, pour la plupart nuls. Comme en sections 3.10.1 et 3.10.2, **A** est singulière pour $\alpha = 0$, et son conditionnement s'améliore quand $\alpha$ augmente.

Tous les éléments de **b** sont pris égaux à un, et `b` est construit comme

```
b = ones(n,1);
```

Pour tout $\alpha > 0$, il est facile de vérifier que l'unique solution de (3.1) est telle que tous ses éléments sont égaux à un, sauf le dernier qui est égal à zéro. Ce système a été résolu avec le même script que dans la section précédente par élimination gaussienne, factorisation LU avec pivotage partiel, factorisation QR et SVD, sans exploiter le caractère creux de **A**. Pour l'itération de Krylov, `sA` a été utilisée à la place de `A`. Le script qui suit a été employé pour régler certains paramètres optionnels de `gmres` :

```
restart = 10;
tol = 1e-12;
maxit = 15;
xKRY = gmres(sA,b,restart,tol,maxit);
```

(voir la documentation de `gmres` pour plus de détails).

Pour $\alpha = 10^{-7}$, cond $\mathbf{A} \approx 4 \cdot 10^7$ et on obtient les résultats suivants. Le temps mis par chaque méthode est en secondes. Comme $\dim \mathbf{x} = 1000$, seuls les deux derniers éléments de la solution numérique sont indiqués. Rappelons que le premier d'entre eux devrait être égal à un, et le second à zéro.

```
TimeGE = 8.526009399999999e-02
LastofxGE =
     1
     0

TimeLUP = 1.363140280000000e-01
LastofxLUP  =
     1
     0
```

```
TimeQR = 9.576683100000000e-02
LastofxQR =
     1
     0

TimeSVD = 1.395477389000000e+00
LastofxSVD =
     1
     0

gmres(10) converged at outer iteration 1
(inner iteration 4)
to a solution with relative residual 1.1e-21.

TimeKRY = 9.034646100000000e-02
LastofxKRY =
     1.000000000000022e+00
     1.551504706009954e-05
```

### *3.10.4* $\mathbf{A}$ *est creuse et symétrique définie positive*

Considérons le même exemple que dans la section 3.10.3, mais avec $n = 10^6$, $\mathbf{A}$ remplacée par $\mathbf{A}^T\mathbf{A}$ et $\mathbf{b}$ remplacé par $\mathbf{A}^T\mathbf{b}$. `sATA`, la représentation creuse de la matrice $\mathbf{A}^T\mathbf{A}$ (symétrique définie positive), peut être construite par

```
sATA = sparse(1:n,1:n,1); % représentation creuse
% de la matrice identité (n,n)
sATA(1,1) = 2;
sATA(1,n) = 2+alpha;
sATA(n,1) = 2+alpha;
sATA(n,n) = (1+alpha)^2+1;
```

et `ATb`, qui représente $\mathbf{A}^T\mathbf{b}$, peut être construite par

```
ATb = ones(n,1);
ATb(1) = 2;
ATb(n) = 2+alpha;
```

(Une représentation dense de $\mathbf{A}^T\mathbf{A}$ serait ingérable, avec $10^{12}$ éléments.)

La méthode des gradients conjugués (éventuellement préconditionnés) est mise en œuvre dans la fonction `pcg`, qu'on peut appeler comme suit

```
tol = 1e-15; % à régler
xCG = pcg(sATA,ATb,tol);
```

Pour $\alpha = 10^{-3}$, cond $(\mathbf{A}^T\mathbf{A}) \approx 1.6 \cdot 10^7$ et on obtient les résultats suivants. Le temps est en secondes. Comme $\dim \mathbf{x} = 10^6$, seuls les deux derniers éléments de la solution numérique sont fournis. Rappelons que le premier d'entre eux devrait être égal à un, et le second à zéro.

```
pcg converged at iteration 6 to a solution
with relative residual 2.2e-18.

TimePCG = 5.922985430000000e-01
LastofxPCG =
     1
    -5.807653514112821e-09
```

## 3.11 En résumé

— La résolution de systèmes d'équations linéaires joue un rôle crucial dans presque toutes les méthodes considérées dans ce livre, et prend souvent la plus grande part du temps de calcul.
— La méthode de Cramer n'est même pas envisageable.
— L'inversion de matrices est inutilement coûteuse, à moins que **A** n'ait une structure très particulière.
— Plus le conditionnement de **A** devient grand, plus le problème risque de devenir difficile.
— La résolution via une factorisation LU est l'outil de base si **A** n'a pas de structure particulière exploitable. Le pivotage permet de l'appliquer à n'importe quelle matrice **A** non singulière. Bien que la factorisation LU augmente le conditionnement du problème, elle le fait avec mesure et peut marcher tout aussi bien que la factorisation QR ou SVD sur les problèmes mal conditionnés, pour moins de calculs.
— Quand la solution n'est pas satisfaisante, une correction itérative peut conduire rapidement à une amélioration spectaculaire.
— La résolution via une factorisation QR est plus coûteuse que via une factorisation LU mais ne détériore pas le conditionnement. Les transformations orthonormales jouent un rôle central dans cette propriété.
— La résolution via une SVD, aussi à base de transformations orthonormales, est encore plus coûteuse que via une factorisation QR. Elle a l'avantage de fournir en sous-produit le conditionnement de **A** pour la norme spectrale, et de permettre de trouver par régularisation des solutions approchées à des problèmes très mal conditionnés.
— La factorisation de Cholesky est un cas particulier de la factorisation LU, qui s'applique quand **A** est symétrique et définie positive. Elle peut aussi être utilisée pour tester si des matrices symétriques sont définies positives.

— Quand **A** est grande et creuse, les méthodes de Krylov ont avantageusement remplacé les méthodes itératives classiques, car elles convergent plus rapidement et plus souvent grâce à des préconditionnements adaptés.
— Quand **A** est grande, creuse, symétrique et définie positive, la méthode des gradients conjugués, qu'on peut voir comme un cas particulier de méthode de Krylov, est la méthode de référence.
— Pour traiter des matrices grandes et creuses, une réindexation appropriée de leurs éléments non nuls peut accélérer les calculs par plusieurs ordres de grandeur.

# Chapitre 4
# Résoudre d'autres problèmes d'algèbre linéaire

Ce chapitre traite de l'évaluation de l'inverse, du déterminant, des valeurs propres et des vecteurs propres d'une matrice $\mathbf{A}$ de dimensions $n \times n$.

## 4.1 Inverser des matrices

> Avant d'évaluer l'inverse d'une matrice, il faut vérifier que le vrai problème n'est pas plutôt la résolution d'un système d'équations linéaires (si tel est le cas, voir le chapitre 3).

A moins que $\mathbf{A}$ n'ait une structure très particulière, qu'elle ne soit par exemple diagonale, on l'inverse en général en calculant la solution $\mathbf{A}^{-1}$ de

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}_n. \tag{4.1}$$

Ceci revient à résoudre les $n$ systèmes linéaires

$$\mathbf{A}\mathbf{x}^i = \mathbf{e}^i, \qquad i = 1, \cdots, n, \tag{4.2}$$

où $\mathbf{x}^i$ est la $i$-ème colonne de $\mathbf{A}^{-1}$ et $\mathbf{e}^i$ la $i$-ème colonne de $\mathbf{I}_n$.

*Remarque 4.1.* Puisque les $n$ systèmes (4.2) ont la même matrice $\mathbf{A}$, si l'on souhaite utiliser une factorisation LU ou QR, celle-ci peut être calculée une fois pour toutes. Avec la factorisation LU, par exemple, inverser une matrice $n \times n$ dense $\mathbf{A}$ requiert environ $8n^3/3$ flops, alors que la résolution de $\mathbf{A}\mathbf{x} = \mathbf{b}$ coûte seulement environ $(2n^3/3) + 2n^2$ flops. □

Pour la factorisation LU avec pivotage partiel, résoudre (4.2) revient à résoudre les systèmes triangulaires

$$\mathbf{L}\mathbf{y}^i = \mathbf{P}\mathbf{e}^i, \quad i = 1, \cdots, n, \tag{4.3}$$

pour calculer les $\mathbf{y}^i$, et

$$\mathbf{U}\mathbf{x}^i = \mathbf{y}^i, \quad i = 1, \cdots, n, \tag{4.4}$$

pour calculer les $\mathbf{x}^i$.

Pour la factorisation QR, cela revient à résoudre les systèmes triangulaires

$$\mathbf{R}\mathbf{x}^i = \mathbf{Q}^{\mathrm{T}}\mathbf{e}^i, \quad i = 1, \cdots, n, \tag{4.5}$$

pour calculer les $\mathbf{x}^i$.

Pour la SVD, on a directement

$$\mathbf{A}^{-1} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}}, \tag{4.6}$$

et inverser $\boldsymbol{\Sigma}$ est trivial puisque cette matrice est diagonale.

Le classement de ces méthodes en termes de nombre de flops requis est le même que pour la résolution des systèmes linéaires :

$$\mathrm{LU} < \mathrm{QR} < \mathrm{SVD}, \tag{4.7}$$

chacun d'eux requérant $\mathrm{O}(n^3)$ flops. Ce n'est pas si mal, si on se souvient que le simple produit de deux matrices $n \times n$ génériques requiert déjà $\mathrm{O}(n^3)$ flops.

## 4.2 Calculer des déterminants

> Évaluer des déterminants est rarement utile. Pour vérifier, par exemple, qu'une matrice est numériquement inversible, il est préférable d'évaluer son conditionnement (voir la section 3.3).

Sauf peut-être pour de petits exemples académiques, les déterminants ne devraient jamais être calculés par expansion des cofacteurs, car cette méthode est immensément coûteuse (voir l'exemple 1.1) et manque de robustesse. Une fois encore, il est préférable de faire appel à une factorisation.

Avec la factorisation LU avec pivotage partiel,

$$\mathbf{A} = \mathbf{P}^{\mathrm{T}}\mathbf{L}\mathbf{U}, \tag{4.8}$$

de sorte que

$$\det \mathbf{A} = \det\left(\mathbf{P}^{\mathrm{T}}\right) \cdot \det(\mathbf{L}) \cdot \det \mathbf{U}, \tag{4.9}$$

où

$$\det \mathbf{P}^{\mathrm{T}} = (-1)^p, \tag{4.10}$$

avec $p$ le nombre des échanges de lignes dus au pivotage, où

$$\det \mathbf{L} = 1 \tag{4.11}$$

et où $\det \mathbf{U}$ est le produit des éléments diagonaux de $\mathbf{U}$.

Avec la factorisation QR,

$$\mathbf{A} = \mathbf{QR}, \tag{4.12}$$

de sorte que

$$\det \mathbf{A} = \det(\mathbf{Q}) \cdot \det \mathbf{R}. \tag{4.13}$$

L'équation (3.64) implique que

$$\det \mathbf{Q} = (-1)^q, \tag{4.14}$$

où $q$ est le nombre de transformations de Householder, et $\det \mathbf{R}$ est égal au produit des éléments diagonaux de $\mathbf{R}$.

Avec la SVD,

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}}, \tag{4.15}$$

de sorte que

$$\det \mathbf{A} = \det(\mathbf{U}) \cdot \det(\boldsymbol{\Sigma}) \cdot \det \mathbf{V}^{\mathrm{T}}. \tag{4.16}$$

Or $\det \mathbf{U} = \pm 1$, $\det \mathbf{V}^{\mathrm{T}} = \pm 1$ et $\det \boldsymbol{\Sigma} = \prod_{i=1}^{n} \sigma_{i,i}$.

## 4.3 Calculer des valeurs propres et des vecteurs propres

### *4.3.1 Approche à éviter*

Les valeurs propres de la matrice (carrée) $\mathbf{A}$ sont les solutions en $\lambda$ de l'*équation caractéristique*

$$\det(\mathbf{A} - \lambda \mathbf{I}) = 0, \tag{4.17}$$

et le vecteur propre $\mathbf{v}^i$ associé à la valeur propre $\lambda_i$ est tel que

$$\mathbf{A}\mathbf{v}^i = \lambda_i \mathbf{v}^i, \tag{4.18}$$

ce qui le définit (à une constante multiplicative près).

On pourrait donc penser à une procédure en trois étapes, où les coefficients de l'équation polynomiale (4.17) seraient évalués à partir de $\mathbf{A}$, avant d'utiliser un algorithme d'usage général pour calculer les $\lambda_i$ en résolvant cette équation polynomiale puis de calculer les $\mathbf{v}^i$ en résolvant le système linéaire (4.18) pour chacun des $\lambda_i$ ainsi obtenus. Sauf si le problème est très petit, *ceci est une mauvaise idée*, ne serait-ce que parce que les racines d'une équation polynomiale peuvent être très sensibles à des erreurs sur les coefficients de ce polynôme (voir le *polynôme perfide* (4.59)

en section 4.4.3). L'exemple 4.3 montrera qu'on peut, au contraire, transformer la recherche des racines d'un polynôme en celle des valeurs propres d'une matrice.

### *4.3.2 Exemples d'applications*

Les applications du calcul de valeurs propres et de vecteurs propres sont très variés, comme le montrent les exemples qui suivent. Dans le premier de ceux-ci, un seul vecteur propre doit être calculé, qui est associé à une valeur propre connue. La réponse s'est avérée avoir des conséquences économiques majeures.

*Exemple 4.1. PageRank*

PageRank est un algorithme utilisé par Google, entre autres considérations, pour décider dans quel ordre des pointeurs sur les pages pertinentes doivent être présentés en réponse à une question donnée d'un surfeur sur le WEB [137], [33]. Soit $N$ le nombre de pages indexées. PageRank utilise une matrice de connexion $\mathbf{G}$ de dimensions $N \times N$, telle que $g_{i,j} = 1$ s'il existe une lien hypertexte de la page $j$ vers la page $i$, et $g_{i,j} = 0$ dans le cas contraire. $\mathbf{G}$ est donc une matrice énorme mais très creuse.

Soit $\mathbf{x}^k \in \mathbb{R}^N$ un vecteur dont $i$-ème élément est la probabilité que le surfeur soit dans la $i$-ème page après $k$ changements de page. Toutes les pages ont initialement la même probabilité, de sorte que

$$x_i^0 = \frac{1}{N}, \quad i = 1, \cdots, N. \tag{4.19}$$

L'évolution de $\mathbf{x}^k$ lors d'un changement de page est décrite par la chaîne de Markov

$$\mathbf{x}^{k+1} = \mathbf{S}\mathbf{x}^k, \tag{4.20}$$

où la matrice de transition $\mathbf{S}$ correspond à un modèle du comportement des surfeurs. Supposons, dans un premier temps, qu'un surfeur suive au hasard n'importe lequel des hyperliens présents dans la page courante. $\mathbf{S}$ est alors une matrice creuse, facile à déduire de $\mathbf{G}$ en procédant comme suit. Son élément $s_{i,j}$ est la probabilité de sauter de la page $j$ vers la page $i$ via un hyperlien. Comme on ne peut pas rester dans la $j$-ème page, $s_{j,j} = 0$. Chacun des $n_j$ éléments non nuls de la $j$-ème colonne de $\mathbf{S}$ est égal à $1/n_j$, de sorte que la somme de tous les éléments d'une colonne de $\mathbf{S}$ quelconque est égal à un.

Ce modèle manque de réalisme, car certaines pages ne contiennent aucun hyperlien ou ne sont le point de destination d'aucun hyperlien. C'est pourquoi on complique le modèle en supposant que le surfeur peut aléatoirement sauter vers une page arbitraire (avec une probabilité 0.15) ou choisir de suivre l'un quelconque des hyperliens présents dans la page courante (avec une probabilité 0.85). Ceci conduit à remplacer $\mathbf{S}$ dans (4.20) par

$$\mathbf{A} = \alpha\mathbf{S} + (1-\alpha)\frac{\mathbf{1}\cdot\mathbf{1}^{\mathrm{T}}}{N}, \tag{4.21}$$

avec $\alpha = 0.85$ et $\mathbf{1}$ un vecteur colonne de dimension $N$ dont tous les éléments sont égaux à un. Avec ce modèle, la probabilité de rester à la même page n'est plus nulle. Bien que $\mathbf{A}$ ne soit plus creuse, l'évaluation de $\mathbf{A}\mathbf{x}^k$ reste presque aussi simple que si elle l'était.

Après une infinité de sauts de page, la distribution asymptotique des probabilités $\mathbf{x}^\infty$ est telle que

$$\mathbf{A}\mathbf{x}^\infty = \mathbf{x}^\infty, \tag{4.22}$$

de sorte que $\mathbf{x}^\infty$ est un vecteur propre de $\mathbf{A}$, associé à une valeur propre unité. Les vecteurs propres sont définis à une constante multiplicative près, mais le sens de $\mathbf{x}^\infty$ implique que

$$\sum_{i=1}^{N} x_i^\infty = 1. \tag{4.23}$$

Une fois $\mathbf{x}^\infty$ évalué, les pages pertinentes associées aux éléments de plus grandes valeurs de $\mathbf{x}^\infty$ peuvent être présentées en premier. Les matrices de transition des chaînes de Markov sont telles que leur valeur propre la plus grande est égale à un. Classer des pages WEB revient donc ici à calculer le vecteur propre associé à la plus grande valeur propre (connue) d'une matrice gigantesque. □

*Exemple 4.2. Oscillations de ponts*

Le matin du 7 novembre 1940, le pont de Tacoma s'est violemment vrillé sous l'action du vent avant de s'écrouler dans les eaux froides du détroit de Puget. Ce pont s'était vu affublé du surnom de *Galloping Gertie* à cause de son comportement inhabituel, et c'est un coup de chance extraordinaire qu'aucun amateur de sensations fortes n'ait été tué dans ce désastre. La vidéo de l'événement, disponible sur le WEB, est un rappel brutal de l'importance de tenir compte du risque d'oscillations lors de la conception de ponts.

Un modèle linéaire dynamique d'un pont, valide pour de petits déplacements, est fourni par l'équation différentielle vectorielle

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{C}\dot{\mathbf{x}} + \mathbf{K}\mathbf{x} = \mathbf{u}, \tag{4.24}$$

où $\mathbf{M}$ est une matrice de masses, $\mathbf{C}$ une matrice d'amortissements, $\mathbf{K}$ une matrice de coefficients de raideur, $\mathbf{x}$ un vecteur décrivant les déplacements des nœuds d'un maillage par rapport à leurs positions d'équilibre en l'absence de forces extérieures, et $\mathbf{u}$ un vecteur de forces extérieures. $\mathbf{C}$ est souvent négligeable, et c'est pourquoi les oscillations sont si dangereuses. L'équation autonome (c'est à dire en l'absence d'entrée extérieure) devient alors

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{K}\mathbf{x} = \mathbf{0}. \tag{4.25}$$

Toutes les solutions de cette équation sont des combinaisons linéaires des modes propres $\mathbf{x}^k$, avec

$$\mathbf{x}^k(t) = \boldsymbol{\rho}^k \exp[\mathrm{i}(\omega_k t + \varphi_k)], \tag{4.26}$$

où i est l'unité imaginaire, telle que $\mathrm{i}^2 = -1$, $\omega_k$ est une pulsation de résonance et $\boldsymbol{\rho}^k$ est la forme du mode associé. Insérons (4.26) dans (4.25) pour obtenir

$$(\mathbf{K} - \omega_k^2 \mathbf{M})\boldsymbol{\rho}^k = \mathbf{0}. \tag{4.27}$$

Calculer $\omega_k^2$ et $\boldsymbol{\rho}^k$ est un *problème de valeurs propres généralisé* [202]. En général, $\mathbf{M}$ est inversible, de sorte que cette équation peut être transformée en

$$\mathbf{A}\boldsymbol{\rho}^k = \lambda_k \boldsymbol{\rho}^k, \tag{4.28}$$

avec $\lambda_k = \omega_k^2$ et $\mathbf{A} = \mathbf{M}^{-1}\mathbf{K}$. Le calcul des $\omega_k$ et $\boldsymbol{\rho}^k$ peut donc se ramener à celui de valeurs propres et de vecteurs propres. La résolution du problème de valeurs propres généralisé peut cependant se révéler préférable car des propriétés utiles de $\mathbf{M}$ et $\mathbf{K}$ peuvent être perdues lors du calcul de $\mathbf{M}^{-1}\mathbf{K}$. □

*Exemple 4.3. Résoudre une équation polynomiale*

Les racines de l'équation polynomiale

$$x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = 0 \tag{4.29}$$

sont les valeurs propres de sa *matrice compagne*

$$\mathbf{A} = \begin{bmatrix} 0 & \cdots & \cdots & 0 & -a_0 \\ 1 & \ddots & 0 & \vdots & -a_1 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & -a_{n-1} \end{bmatrix}, \tag{4.30}$$

et l'une des méthodes les plus efficaces pour calculer ces racines est de chercher les valeurs propres de $\mathbf{A}$. □

### *4.3.3 Méthode de la puissance itérée*

La méthode de la puissance itérée s'applique quand la valeur propre de $\mathbf{A}$ de plus grand module est réelle et simple. Elle évalue alors cette valeur propre et le vecteur propre correspondant. Elle est surtout utile sur les grandes matrices creuses (ou qui peuvent être traitées comme si elles l'étaient, comme dans PageRank).

Supposons, pour le moment, que la valeur propre $\lambda_{\max}$ de plus grand module soit positive. Pourvu que $\mathbf{v}^0$ ait une composante non nulle dans la direction du vecteur propre correspondant $\mathbf{v}_{\max}$, itérer

$$\mathbf{v}^{k+1} = \mathbf{A}\mathbf{v}^k \tag{4.31}$$

fera alors décroître l'angle entre $\mathbf{v}^k$ et $\mathbf{v}_{\max}$. Pour assurer $\left\|\mathbf{v}^{k+1}\right\|_2 = 1$, (4.31) est remplacée par

$$\mathbf{v}^{k+1} = \frac{1}{\left\|\mathbf{A}\mathbf{v}^k\right\|_2}\mathbf{A}\mathbf{v}^k. \tag{4.32}$$

Après convergence,

$$\mathbf{A}\mathbf{v}^{\infty} = \|\mathbf{A}\mathbf{v}^{\infty}\|_2 \, \mathbf{v}^{\infty}, \tag{4.33}$$

de sorte que $\lambda_{\max} = \|\mathbf{A}\mathbf{v}^{\infty}\|_2$ et $\mathbf{v}_{\max} = \mathbf{v}^{\infty}$. La convergence peut être lente si d'autres valeurs propres ont un module proche de celui de $\lambda_{\max}$.

*Remarque 4.2.* Quand $\lambda_{\max}$ est négative, la récurrence devient

$$\mathbf{v}^{k+1} = -\frac{1}{\left\|\mathbf{A}\mathbf{v}^{k}\right\|_2}\mathbf{A}\mathbf{v}^{k}, \tag{4.34}$$

de sorte qu'après convergence

$$\mathbf{A}\mathbf{v}^{\infty} = -\|\mathbf{A}\mathbf{v}^{\infty}\|_2 \, \mathbf{v}^{\infty}. \tag{4.35}$$

□

*Remarque 4.3.* Si $\mathbf{A}$ est symétrique, alors ses vecteurs propres sont orthogonaux et, pourvu que $\|\mathbf{v}_{\max}\|_2 = 1$, la matrice

$$\mathbf{A}' = \mathbf{A} - \lambda_{\max}\mathbf{v}_{\max}\mathbf{v}_{\max}^{\mathrm{T}} \tag{4.36}$$

a les même valeurs propres et vecteurs propres que $\mathbf{A}$, sauf pour $\mathbf{v}_{\max}$, qui est maintenant associé à $\lambda = 0$. On peut ainsi appliquer la méthode de la puissance itérée pour trouver la valeur propre avec la deuxième plus grande magnitude et le vecteur propre correspondant. Cette *procédure de déflation* est à itérer avec prudence, car les erreurs se cumulent. □

### *4.3.4 Méthode de la puissance inverse*

Quand $\mathbf{A}$ est inversible et que sa valeur propre $\lambda_{\min}$ de plus petit module est réelle et unique, la valeur propre de $\mathbf{A}^{-1}$ de plus grand module est $\frac{1}{\lambda_{\min}}$, de sorte qu'une itération en puissance inverse

$$\mathbf{v}^{k+1} = \frac{1}{\|\mathbf{A}^{-1}\mathbf{v}^{k}\|_2}\mathbf{A}^{-1}\mathbf{v}^{k} \tag{4.37}$$

peut être utilisée pour calculer $\lambda_{\min}$ et le vecteur propre correspondant (pourvu que $\lambda_{\min} > 0$). L'inversion de $\mathbf{A}$ est évitée en calculant $\mathbf{v}^{k+1}$ par résolution du système

$$\mathbf{A}\mathbf{v}^{k+1} = \mathbf{v}^{k}, \tag{4.38}$$

et en normalisant le résultat. Si une factorisation de $\mathbf{A}$ est utilisée à cet effet, elle est calculée une fois pour toutes. Une modification triviale de l'algorithme permet de traiter le cas $\lambda_{\min} < 0$.

### 4.3.5 Méthode de la puissance inverse avec décalage

La méthode de la puissance inverse avec décalage vise à calculer un vecteur propre $\mathbf{x}^i$ associé à une valeur propre *isolée* $\lambda_i$ connue approximativement. Cette valeur propre n'a plus besoin d'être celle de plus grand ou de plus petit module. Cette méthode peut être utilisée sur des matrices réelles ou complexes. Elle est particulièrement efficace sur les matrices $\mathbf{A}$ *normales*, c'est à dire qui commutent avec leur transconjuguée $\mathbf{A}^{\mathrm{H}}$ de sorte que

$$\mathbf{A}\mathbf{A}^{\mathrm{H}} = \mathbf{A}^{\mathrm{H}}\mathbf{A}. \tag{4.39}$$

Pour les matrices réelles, ceci se traduit par

$$\mathbf{A}\mathbf{A}^{\mathrm{T}} = \mathbf{A}^{\mathrm{T}}\mathbf{A}, \tag{4.40}$$

et les matrices symétriques sont donc normales.

Soit $\rho$ une valeur approximative de $\lambda_i$, avec $\rho \neq \lambda_i$. Puisque

$$\mathbf{A}\mathbf{x}^i = \lambda_i \mathbf{x}^i, \tag{4.41}$$

nous avons

$$(\mathbf{A} - \rho\mathbf{I})\mathbf{x}^i = (\lambda_i - \rho)\mathbf{x}^i. \tag{4.42}$$

Multiplions (4.42) à gauche par $(\mathbf{A} - \rho\mathbf{I})^{-1}(\lambda_i - \rho)^{-1}$, pour obtenir

$$(\mathbf{A} - \rho\mathbf{I})^{-1}\mathbf{x}^i = (\lambda_i - \rho)^{-1}\mathbf{x}^i. \tag{4.43}$$

Le vecteur $\mathbf{x}^i$ est donc aussi un vecteur propre de $(\mathbf{A} - \rho\mathbf{I})^{-1}$, associé à la valeur propre $(\lambda_i - \rho)^{-1}$. En choisissant $\rho$ proche de $\lambda_i$, et pourvu que les autres valeurs propres de $\mathbf{A}$ soit suffisamment loin, on peut assurer que, pour tout $j \neq i$,

$$\frac{1}{|\lambda_i - \rho|} \gg \frac{1}{|\lambda_j - \rho|}. \tag{4.44}$$

L'itération en puissance inverse décalée

$$\mathbf{v}^{k+1} = (\mathbf{A} - \rho\mathbf{I})^{-1}\mathbf{v}^k, \tag{4.45}$$

combinée avec une normalisation de $\mathbf{v}^{k+1}$ à chaque itération, converge alors vers un vecteur propre de $\mathbf{A}$ associé à $\lambda_i$. En pratique, on calcule plutôt $\mathbf{v}^{k+1}$ en résolvant le système

$$(\mathbf{A} - \rho\mathbf{I})\mathbf{v}^{k+1} = \mathbf{v}^k \tag{4.46}$$

(en général via une factorisation LU avec pivotage partiel de $\mathbf{A} - \rho\mathbf{I}$, calculée une fois pour toutes). Quand $\rho$ se rapproche de $\lambda_i$, la matrice $\mathbf{A} - \rho\mathbf{I}$ devient presque singulière, ce qui n'empêche pas l'algorithme de fonctionner très bien, au moins quand $\mathbf{A}$ est normale. Ses propriétés, y compris son comportement sur des matrices qui ne le sont pas, sont étudiées dans [113].

### 4.3.6 Itération QR

L'itération QR, à base de factorisation QR, permet d'évaluer toutes les valeurs propres d'une matrice carrée $\mathbf{A}$ à coefficients réels, pourvu qu'elle ne soit pas de trop grande taille. Ces valeurs propres peuvent être réelles ou complexes conjuguées. On suppose seulement que leurs modules diffèrent (sauf, bien sûr, pour une paire de valeurs propres complexes conjuguées). Un récit intéressant de l'histoire de cet algorithme fascinant est dans [177]. Sa convergence est étudiée dans [249].

La méthode de base est comme suit. Partant de $\mathbf{A}_0 = \mathbf{A}$ et $i = 0$, répéter jusqu'à convergence

1. Factoriser $\mathbf{A}_i$ comme $\mathbf{Q}_i\mathbf{R}_i$.
2. Inverser l'ordre des facteurs résultants $\mathbf{Q}_i$ et $\mathbf{R}_i$ pour former $\mathbf{A}_{i+1} = \mathbf{R}_i\mathbf{Q}_i$.
3. Incrémenter $i$ d'une unité et aller au pas 1.

Pour des raisons compliquées à expliquer, ceci transfère de la masse de la partie triangulaire inférieure de $\mathbf{A}_i$ vers la partie triangulaire supérieure de $\mathbf{A}_{i+1}$. Le fait que $\mathbf{R}_i = \mathbf{Q}_i^{-1}\mathbf{A}_i$ implique que $\mathbf{A}_{i+1} = \mathbf{Q}_i^{-1}\mathbf{A}_i\mathbf{Q}_i$. Les matrices $\mathbf{A}_{i+1}$ et $\mathbf{A}_i$ ont donc les mêmes valeurs propres. Après convergence, $\mathbf{A}_\infty$ est une matrice triangulaire supérieure par blocs avec les mêmes valeurs propres que $\mathbf{A}$, dans ce qu'on appelle une *forme de Schur réelle*. Il n'y a que des blocs scalaires et $2 \times 2$ sur la diagonale de $\mathbf{A}_\infty$. Chaque bloc scalaire contient une valeur propre réelle de $\mathbf{A}$, tandis que les valeurs propres des blocs $2 \times 2$ sont des valeurs propres complexes conjuguées de $\mathbf{A}$. Si $\mathbf{B}$ est l'un de ces blocs $2 \times 2$, alors ses valeurs propres sont les racines de l'équation du second degré

$$\lambda^2 - \operatorname{trace}(\mathbf{B})\,\lambda + \det \mathbf{B} = 0. \tag{4.47}$$

La factorisation qui en résulte

$$\mathbf{A} = \mathbf{Q}\mathbf{A}_\infty\mathbf{Q}^{\mathrm{T}} \tag{4.48}$$

est appelée *décomposition de Schur* (réelle). Puisque

$$\mathbf{Q} = \prod_i \mathbf{Q}_i, \tag{4.49}$$

elle est orthonormale, comme produit de matrices orthonormales, et $(4.48)$ implique que

$$\mathbf{A} = \mathbf{Q}\mathbf{A}_\infty\mathbf{Q}^{-1} \tag{4.50}$$

*Remarque 4.4.* Après avoir indiqué que "les bonnes mises en œuvre [de l'itération QR] sont depuis longtemps beaucoup plus largement disponibles que les bonnes explications", [246] montre que cet algorithme n'est qu'une mise en œuvre astucieuse et numériquement robuste de la méthode de la puissance itérée de la section 4.3.3, appliquée à toute une base de $\mathbb{R}^n$ plutôt qu'à un vecteur unique. □

*Remarque 4.5.* Chaque fois que $\mathbf{A}$ n'est pas une matrice de Hessenberg supérieure (c'est à dire une matrice triangulaire supérieure complétée d'une diagonale non

nulle juste en dessous de la diagonale principale), une variante triviale de la factorisation QR est tout d'abord utilisée pour la mettre sous cette forme. Ceci accélère considérablement l'itération QR, car la forme de Hessenberg supérieure est préservée par les itérations. Notons que la matrice compagne de l'exemple 4.3 est déjà sous forme de Hessenberg supérieure. □

Quand $\mathbf{A}$ *est symétrique*, toutes ses valeurs propres $\lambda_i$ $(i = 1, \cdots, n)$ sont réelles, et les vecteurs propres correspondants $\mathbf{v}^i$ sont orthogonaux. L'itération QR produit alors une série de matrices symétriques $\mathbf{A}_k$ qui doit converger vers la matrice diagonale

$$\boldsymbol{\Lambda} = \mathbf{Q}^{-1}\mathbf{A}\mathbf{Q}, \tag{4.51}$$

avec $\mathbf{Q}$ orthonormale et

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_n \end{bmatrix}. \tag{4.52}$$

L'équation (4.51) implique que

$$\mathbf{A}\mathbf{Q} = \mathbf{Q}\boldsymbol{\Lambda}, \tag{4.53}$$

ou encore que

$$\mathbf{A}\mathbf{q}^i = \lambda_i \mathbf{q}^i, \quad i = 1, \cdots, n, \tag{4.54}$$

avec $\mathbf{q}^i$ la $i$-ème colonne de $\mathbf{Q}$. Ainsi, $\mathbf{q}^i$ est le vecteur propre associé à $\lambda_i$, et l'itération QR calcule la *décomposition spectrale* de $\mathbf{A}$

$$\mathbf{A} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^{\mathrm{T}} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^{-1}. \tag{4.55}$$

Quand $\mathbf{A}$ *n'est pas symétrique*, le calcul de ses vecteurs propres à partir de la décomposition de Schur reste possible mais devient significativement plus compliqué [42].

### *4.3.7 Itération QR décalée*

La version de base de l'itération QR échoue s'il y a plusieurs valeurs propres réelles (ou plusieurs paires de valeurs propres complexes conjuguées) de même module, comme illustré par l'exemple qui suit.

*Exemple 4.4. Échec de l'itération QR*

La factorisation QR de

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

est

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

de sorte que

$$\mathbf{RQ} = \mathbf{A}$$

et la méthode est bloquée. Ce n'est pas surprenant car les valeurs propres de $\mathbf{A}$ ont la même valeur absolue ($\lambda_1 = 1$ et $\lambda_2 = -1$). □

Pour contourner cette difficulté et accélérer la convergence, l'itération QR décalée de base procède comme suit. Partant de $\mathbf{A}_0 = \mathbf{A}$ et $i = 0$, elle répète jusqu'à convergence

1. Choisir un décalage $\sigma_i$.
2. Factoriser $\mathbf{A}_i - \sigma_i \mathbf{I}$ comme $\mathbf{Q}_i \mathbf{R}_i$.
3. Inverser l'ordre des facteurs résultants $\mathbf{Q}_i$ et $\mathbf{R}_i$ et compenser le décalage pour obtenir $\mathbf{A}_{i+1} = \mathbf{R}_i \mathbf{Q}_i + \sigma_i \mathbf{I}$.

Une stratégie possible est la suivante. Commencer par fixer $\sigma_i$ à la valeur du dernier élément diagonal de $\mathbf{A}_i$, pour accélérer la convergence de la dernière ligne, puis fixer $\sigma_i$ à la valeur de l'avant-dernier élément diagonal de $\mathbf{A}_i$, pour accélérer la convergence de l'avant-dernière ligne, et ainsi de suite.

Beaucoup de travaux ont été consacrés aux propriétés théoriques et à la mise en œuvre pratique de l'itération QR (décalée), et nous n'avons fait ici qu'effleurer le sujet. L'itération QR, appelée dans [229] (cité dans [42]) *l'un des algorithmes les plus remarquables des mathématiques numériques*, se révèle converger dans des situations plus générales que celles pour lesquelles sa convergence a été prouvée. Elle a cependant deux inconvénients principaux. Tout d'abord, les valeurs propres de petit module peuvent être évaluées avec une précision insuffisante, ce qui peut justifier une amélioration itérative. Ensuite, l'algorithme QR n'est pas adapté aux très grandes matrices creuses, car il en détruit le caractère creux. En ce qui concerne la résolution numérique de grands problèmes aux valeurs propres, le lecteur pourra consulter [202], et y découvrir que les sous-espaces de Krylov jouent, là encore, un rôle crucial.

## 4.4 Exemples MATLAB

### *4.4.1 Inverser une matrice*

Considérons à nouveau la matrice $\mathbf{A}$ définie par (3.138). On peut calculer son inverse grâce à la fonction dédiée `inv`, qui procède par élimination gaussienne, ou par l'une quelconque des méthodes disponibles pour résoudre le système linéaire (4.1). On peut ainsi écrire

```
% Inversion par la fonction dédiée
InvADF = inv(A);

% Inversion par élimination gaussienne
I = eye(3); % Matrice identité
InvAGE = A\I;

% Inversion via une factorisation LU
% avec pivotage partiel
[L,U,P] = lu(A);
opts_LT.LT = true;
Y = linsolve(L,P,opts_LT);
opts_UT.UT = true;
InvALUP = linsolve(U,Y,opts_UT);

% Inversion via une factorisation QR
[Q,R] = qr(A);
QTI = Q';
InvAQR = linsolve(R,QTI,opts_UT);

% Inversion via une SVD
[U,S,V] = svd(A);
InvASVD = V*inv(S)*U';
```

L'erreur commise peut être quantifiée par la norme de Frobenius de la différence entre la matrice identité et le produit de `A` par l'estimée de son inverse, calculée comme

```
% Erreur via la fonction dédiée
EDF = I-A*InvADF;
NormEDF = norm(EDF,'fro')

% Erreur via une élimination gaussienne
EGE = I-A*InvAGE;
NormEGE = norm(EGE,'fro')

% Erreur via une factorisation LU
% avec pivotage partiel
ELUP = I-A*InvALUP;
NormELUP = norm(ELUP,'fro')

% Erreur via une factorisation QR
EQR = I-A*InvAQR;
NormEQR = norm(EQR,'fro')

% Erreur via une SVD
```

```
ESVD = I-A*InvASVD;
NormESVD = norm(ESVD,'fro')
```

Pour $\alpha = 10^{-13}$,

```
NormEDF = 3.685148879709611e-02
NormEGE = 1.353164693413185e-02
NormELUP = 1.353164693413185e-02
NormEQR =  3.601384553630034e-02
NormESVD = 1.732896329126472e-01
```

Pour $\alpha = 10^{-5}$,

```
NormEDF = 4.973264728508383e-10
NormEGE = 2.851581367178794e-10
NormELUP = 2.851581367178794e-10
NormEQR = 7.917097832969996e-10
NormESVD = 1.074873453042201e-09
```

Une fois encore, la factorisation LU avec pivotage partiel se révèle un très bon choix sur cet exemple, car elle atteint la plus petite norme de l'erreur et requiert le moins de flops.

### *4.4.2 Calculer un déterminant*

Nous exploitons ici le fait que le déterminant de la matrice **A** définie par (3.138) est égal à $-3\alpha$. Soit `detX` la valeur numérique du déterminant calculée par la méthode X. Nous calculons l'erreur *relative* de cette méthode comme

```
TrueDet = -3*alpha;
REdetX = (detX-TrueDet)/TrueDet
```

Le déterminant de **A** peut être calculé soit par la fonction dédiée `det`, comme

```
detDF = det(A);
```

ou en évaluant le produit des déterminants des matrices d'une factorisation LUP, QR ou SVD.

Pour $\alpha = 10^{-13}$,

```
TrueDet = -3.000000000000000e-13
REdetDF = -7.460615985110166e-03
REdetLUP = -7.460615985110166e-03
REdetQR = -1.010931238834050e-02
REdetSVD = -2.205532173587620e-02
```

Pour $\alpha = 10^{-5}$,

```
TrueDet = -3.000000000000000e-05
REdetDF = -8.226677621822146e-11
REdetLUP = -8.226677621822146e-11
REdetQR = -1.129626855380858e-10
REdetSVD = -1.372496047658452e-10
```

La fonction dédiée et la factorisation LU avec pivotage partiel donnent donc des résultats légèrement meilleurs que les approches QR et SVD, pourtant plus coûteuses.

### *4.4.3 Calculer des valeurs propres*

Considérons une fois encore la matrice **A** définie par (3.138). Ses valeurs propres peuvent être évaluées par la fonction dédiée `eig`, à base d'itération QR, comme

```
lambdas = eig(A);
```

Pour $\alpha = 10^{-13}$, on obtient ainsi

```
lambdas =
     1.611684396980710e+01
    -1.116843969807017e+00
     1.551410816840699e-14
```

À comparer avec la solution obtenue en arrondissant au nombre à 16 chiffres le plus proche une approximation à 50 chiffres calculée avec Maple :

$$\lambda_1 = 16.11684396980710, \tag{4.56}$$
$$\lambda_2 = -1.116843969807017, \tag{4.57}$$
$$\lambda_3 = 1.666666666666699 \cdot 10^{-14}. \tag{4.58}$$

Considérons maintenant le fameux *polynôme perfide* de Wilkinson [251, 1, 62]

$$P(x) = \prod_{i=1}^{20} (x-i). \tag{4.59}$$

Il semble plutôt inoffensif, avec ses racines simples régulièrement espacées $x_i = i$ $(i = 1, \cdots, 20)$. Faisons comme si ces racines n'étaient pas connues et devaient être évaluées. Nous développons $P(x)$ grâce à `poly` et cherchons ses racines grâce à `roots`, fondé sur l'itération QR appliquée à la matrice compagne du polynôme. Le script

```
r = zeros(20,1);
for i=1:20,
    r(i) = i;
end
% Calcul des coefficients
```

```
% de la forme développée suivant les puissances
pol = poly(r);
% Calcul des racines
PolRoots = roots(pol)
```

produit

```
PolRoots =
     2.000032487811079e+01
     1.899715998849890e+01
     1.801122169150333e+01
     1.697113218821587e+01
     1.604827463749937e+01
     1.493535559714918e+01
     1.406527290606179e+01
     1.294905558246907e+01
     1.203344920920930e+01
     1.098404124617589e+01
     1.000605969450971e+01
     8.998394489161083e+00
     8.000284344046330e+00
     6.999973480924893e+00
     5.999999755878211e+00
     5.000000341909170e+00
     3.999999967630577e+00
     3.000000001049188e+00
     1.999999999997379e+00
     9.999999999998413e-01
```

Ces résultats ne sont pas très précis. Pire, ils se révèlent extrêmement sensibles à de petites perturbations de certains des coefficients de la forme développée (4.29). Si, par exemple, le coefficient de $x^{19}$, égal à $-210$, est perturbé en y ajoutant $10^{-7}$ tout en laissant les autres coefficients inchangés, alors les solutions fournies par `roots` deviennent

```
PertPolRoots =
      2.042198199932168e+01 + 9.992089606340550e-01i
      2.042198199932168e+01 - 9.992089606340550e-01i
      1.815728058818208e+01 + 2.470230493778196e+00i
      1.815728058818208e+01 - 2.470230493778196e+00i
      1.531496040228042e+01 + 2.698760803241636e+00i
      1.531496040228042e+01 - 2.698760803241636e+00i
      1.284657850244477e+01 + 2.062729460900725e+00i
      1.284657850244477e+01 - 2.062729460900725e+00i
      1.092127532120366e+01 + 1.103717474429019e+00i
      1.092127532120366e+01 - 1.103717474429019e+00i
      9.567832870568918e+00
```

```
9.113691369146396e+00
7.994086000823392e+00
7.000237888287540e+00
5.999998537003806e+00
4.999999584089121e+00
4.000000023407260e+00
2.999999999831538e+00
1.999999999976565e+00
1.000000000000385e+00
```

Dix des vingt racines sont maintenant jugées complexes conjuguées, et sont radicalement différentes de ce qu'elles étaient dans le cas non perturbé. Ceci illustre le fait que trouver les racines d'une équation polynomiale à partir des coefficients de sa forme développée peut être un problème *mal conditionné*. Ceci était bien connu pour les racines multiples ou proches les unes des autres, mais la découverte du fait que ce problème pouvait aussi affecter un polynôme tel que (4.59), qui n'a aucune de ces caractéristiques, fut ce que Wilkinson qualifia d'*expérience la plus traumatisante de* (sa) *carrière d'analyste numérique* [251].

### *4.4.4 Calculer des valeurs propres et des vecteurs propres*

Considérons à nouveau la matrice **A** définie par (3.138). La fonction dédiée `eig` peut aussi calculer des vecteurs propres, même quand **A** n'est pas symétrique, comme ici. L'instruction

```
[EigVect,DiagonalizedA] = eig(A);
```

produit deux matrices. Chaque colonne de `EigVect` contient un vecteur propre $\mathbf{v}^i$ de **A**, tandis que l'élément diagonal correspondant de la matrice diagonale `DiagonalizedA` contient la valeur propre associée $\lambda_i$. Pour $\alpha = 10^{-13}$, les colonnes de `EigVect` sont, de gauche à droite,

```
-2.319706872462854e-01
-5.253220933012315e-01
-8.186734993561831e-01

-7.858302387420775e-01
-8.675133925661158e-02
6.123275602287992e-01

4.082482904638510e-01
-8.164965809277283e-01
4.082482904638707e-01
```

Les éléments diagonaux de `DiagonalizedA` sont, dans le même ordre,

```
1.611684396980710e+01
-1.116843969807017e+00
1.551410816840699e-14
```

Ils sont donc identiques aux valeurs propres obtenues précédemment avec l'instruction `eig(A)`.

Une vérification (très partielle) de la qualité de ces résultats peut être menée à bien avec le script

```
Residual = A*EigVect-EigVect*DiagonalizedA;
NormResidual = norm(Residual,'fro')
```

qui produit

```
NormResidual = 1.155747735077462e-14
```

## 4.5 En résumé

— Bien réfléchir avant d'inverser une matrice. Il peut ne s'agir que de résoudre un système d'équations linéaires.
— Quand elle s'avère nécessaire, l'inversion d'une matrice $n \times n$ peut être effectuée en résolvant $n$ systèmes de $n$ équations linéaires en $n$ inconnues. Si une factorisation LU ou QR de $\mathbf{A}$ est utilisée, il suffit de la calculer une fois pour toutes.
— Bien réfléchir avant d'évaluer un déterminant. Il peut être plus pertinent de calculer un conditionnement.
— Calculer le déterminant de $\mathbf{A}$ est facile à partir d'une factorisation LU ou QR. Le résultat à base de factorisation QR demande plus de calculs mais devrait être plus robuste au mauvais conditionnement.
— La méthode de la puissance itérée peut être utilisée pour calculer la valeur propre de $\mathbf{A}$ de plus grand module (pourvu qu'elle soit réelle et unique), et le vecteur propre associé. Elle est particulièrement intéressante quand $\mathbf{A}$ est grande et creuse. Des variantes de cette méthode peuvent être utilisées pour calculer la valeur propre de $\mathbf{A}$ de plus petit module et le vecteur propre associé, ou le vecteur propre associé à une valeur propre isolée quelconque connue approximativement.
— L'itération QR (décalée) est la méthode de référence pour le calcul simultané de toutes les valeurs propres de $\mathbf{A}$. Elle peut aussi être utilisée pour calculer les vecteurs propres correspondants, ce qui est particulièrement facile si $\mathbf{A}$ est symétrique.
— L'itération QR (décalée) permet aussi de calculer simultanément toutes les racines d'une équation polynomiale à une seule indéterminée. Les résultats peuvent être très sensibles aux valeurs des coefficients du polynôme sous forme développée.

# Chapitre 5
# Interpoler et extrapoler

## 5.1 Introduction

Soit une fonction $\mathbf{f}(\cdot)$ telle que

$$\mathbf{y} = \mathbf{f}(\mathbf{x}), \tag{5.1}$$

où $\mathbf{x}$ est un vecteur d'entrées et $\mathbf{y}$ un vecteur de sorties. Supposons que cette fonction soit une *boîte noire*, c'est à dire qu'elle ne puisse être évaluée que numériquement, sans que rien ne soit connu de son expression formelle. Supposons de plus que $\mathbf{f}(\cdot)$ ait été évaluée à $N$ valeurs numériques $\mathbf{x}^i$ de $\mathbf{x}$, de sorte qu'on connaisse les $N$ valeurs numériques correspondantes du vecteur des sorties

$$\mathbf{y}^i = \mathbf{f}(\mathbf{x}^i), \quad i = 1, \cdots, N. \tag{5.2}$$

Soit $\mathbf{g}(\cdot)$ une autre fonction, en général beaucoup plus simple à évaluer que $\mathbf{f}(\cdot)$, et telle que

$$\mathbf{g}(\mathbf{x}^i) = \mathbf{f}(\mathbf{x}^i), \quad i = 1, \cdots, N. \tag{5.3}$$

Calculer $\mathbf{g}(\mathbf{x})$ est appelé *interpolation* si $\mathbf{x}$ appartient à l'*enveloppe convexe* des $\mathbf{x}^i$, c'est à dire au plus petit polytope convexe qui les contient. Autrement, on parle d'*extrapolation* (figure 5.1). Une lecture à recommander sur l'interpolation (et l'approximation) avec des fonctions polynomiales ou rationnelles est [232] ; voir aussi le délicieux [231].

> Bien que les méthodes développées pour l'interpolation restent utilisables pour l'extrapolation, cette dernière est beaucoup plus dangereuse. Chaque fois que possible, elle devrait donc être évitée, en s'arrangeant pour que l'enveloppe convexe des $\mathbf{x}^i$ contienne le domaine d'intérêt.

*Remarque 5.1.* Ce n'est *pas toujours une bonne idée* que d'interpoler, ne serait-ce que parce que les données $\mathbf{y}^i$ sont souvent corrompues par du bruit. Il est parfois

préférable d'obtenir un modèle plus simple tel que

$$\mathbf{g}(\mathbf{x}^i) \approx \mathbf{f}(\mathbf{x}^i), \quad i = 1, \cdots, N. \tag{5.4}$$

Ce modèle peut fournir des *prédictions bien meilleures* de $\mathbf{y}$ en $\mathbf{x} \neq \mathbf{x}^i$ qu'un modèle interpolant les données. Sa construction optimale sera considérée au chapitre 9. □

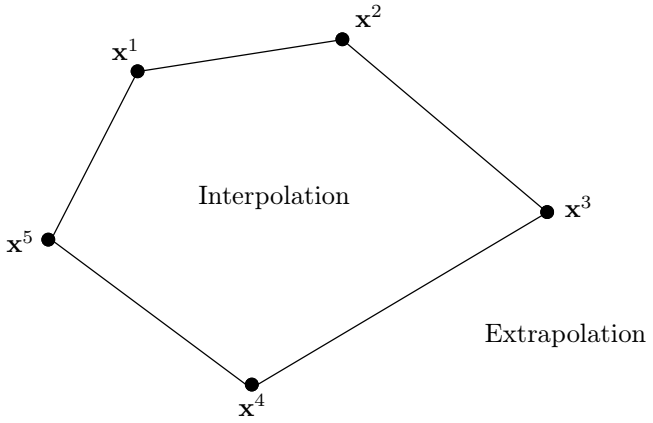


**Fig. 5.1** L'extrapolation prend place à l'extérieur de l'enveloppe convexe des $\mathbf{x}^i$

## 5.2 Exemples

*Exemple 5.1. Expériences sur ordinateur*

Les expériences dans le monde physique sont de plus en plus remplacées par des expériences sur ordinateur, ou *computer experiments*. Lors de la conception de voitures pour satisfaire les normes de sécurité en ce qui concerne les accidents, par exemple, les constructeurs ont partiellement remplacé les *crash tests* de prototypes par des simulations numériques, plus rapides et moins coûteuses mais gourmandes en calcul.

Un code de calcul numérique peut être vu comme une boîte noire qui évalue les valeurs numériques de ses variables de sortie (placées dans $\mathbf{y}$) pour des valeurs numériques données de ses variables d'entrée (placées dans $\mathbf{x}$). Quand le code est déterministe, il définit une fonction

$$\mathbf{y} = \mathbf{f}(\mathbf{x}). \tag{5.5}$$

Sauf dans des cas triviaux, cette fonction ne peut être étudiée qu'à travers des expériences sur ordinateur, où des valeurs numériques potentiellement intéressantes de $\mathbf{x}$ sont utilisées pour calculer les valeurs correspondantes de $\mathbf{y}$ [204].

Pour limiter le nombre d'exécutions d'un code complexe mettant en œuvre une fonction $\mathbf{f}(\cdot)$, on peut souhaiter le remplacer par un code simple mettant en œuvre une fonction $\mathbf{g}(\cdot)$ telle que

$$\mathbf{g}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}) \tag{5.6}$$

pour tout $\mathbf{x}$ dans un domaine d'intérêt $\mathbb{X}$. Exiger que le code simple donne les mêmes sorties que le code complexe pour tous les vecteurs d'entrée $\mathbf{x}^i$ ($i = 1, \cdots, N$) pour lesquels $\mathbf{f}(\cdot)$ a été évaluée revient à exiger que la contrainte d'interpolation (5.3) soit satisfaite. □

*Exemple 5.2. Prototypage*

Supposons maintenant qu'une succession de prototypes soit construite pour diverses valeurs d'un vecteur $\mathbf{x}$ de paramètres de conception, dans le but d'obtenir un produit satisfaisant, comme quantifié par la valeur d'un vecteur $\mathbf{y}$ de caractéristiques de performance mesurées sur ces prototypes. Les données disponibles sont à nouveau sous la forme (5.2), et on peut encore souhaiter disposer d'un code numérique évaluant une fonction $\mathbf{g}(\cdot)$ telle que (5.3) soit satisfaite. Ceci aidera à suggérer de nouvelles valeurs prometteuses de $\mathbf{x}$, pour lesquelles de nouveaux prototypes pourront être construits. Les mêmes outils que ceux employés pour l'expérimentation sur ordinateur peuvent donc aussi être utiles à cette expérimentation dans le monde physique. □

*Exemple 5.3. Prospection minière*

En forant à la latitude $x_1^i$, la longitude $x_2^i$ et la profondeur $x_3^i$ dans une zone aurifère, on recueille un échantillon, avec une concentration $y_i$ en or. Cette concentration dépend de l'endroit où l'échantillon a été collecté, de sorte que $y_i = f(\mathbf{x}^i)$, où $\mathbf{x}^i = (x_1^i, x_2^i, x_3^i)^{\mathrm{T}}$. À partir des mesures de concentration dans ces échantillons très coûteux à obtenir, on souhaite prédire la région la plus prometteuse, par interpolation de $f(\cdot)$. Ceci a motivé le développement du *krigeage*, présenté en section 5.4.3. Bien que le krigeage trouve ses origines en géostatistique, il est de plus en plus utilisé dans l'expérimentation sur ordinateur et le prototypage. □

## 5.3 Cas à une variable

Supposons tout d'abord $\mathbf{x}$ et $\mathbf{y}$ scalaires. L'équation (5.5) se traduit alors par

$$y = f(x). \tag{5.7}$$

La figure 5.2 illustre la non-unicité de la fonction interpolatrice. On recherchera donc cette fonction dans une classe préspécifiée, par exemple celle des polynômes ou des fractions rationnelles (c'est à dire des rapports de polynômes).

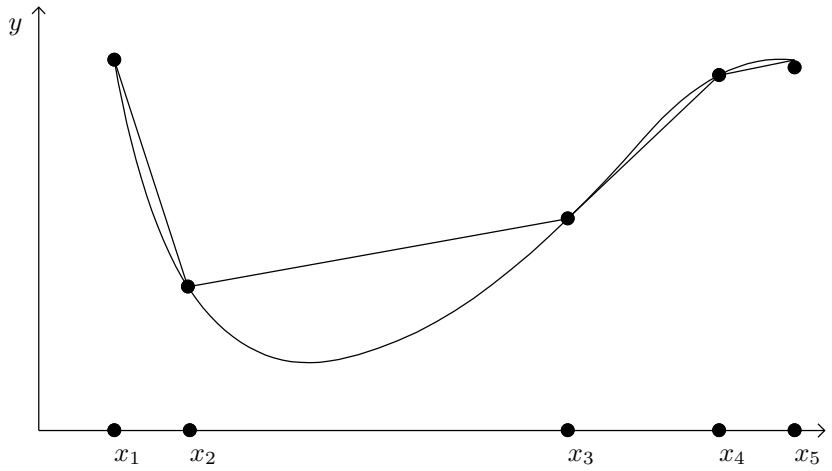


**Fig. 5.2** Interpolateurs

### *5.3.1 Interpolation polynomiale*

L'interpolation polynomiale est utilisée en routine, par exemple, pour l'intégration et la dérivation de fonctions (voir le chapitre 6). Le polynôme de degré $n$

$$P_n(x,\mathbf{p}) = \sum_{i=0}^{n} a_i x^i \tag{5.8}$$

dépend de $n+1$ paramètres $a_i$, qui définissent le vecteur

$$\mathbf{p} = (a_0, a_1, \cdots, a_n)^{\mathrm{T}}. \tag{5.9}$$

$P_n(x,\mathbf{p})$ peut donc interpoler $n+1$ points expérimentaux

$$\{x_j, y_j\}, \quad j = 0, \cdots, n, \tag{5.10}$$

soit autant qu'il y a de paramètres scalaires dans $\mathbf{p}$.

*Remarque 5.2.* Si les données peuvent être décrites exactement par un polynôme de degré moins élevé (par exemple si elles sont alignées), alors un polynôme de degré $n$ peut interpoler plus de $n+1$ données. □

*Remarque 5.3.* Une fois $\mathbf{p}$ calculé, interpoler veut dire évaluer (5.8) pour des valeurs connues de $x$ et des $a_i$. Une mise en œuvre naïve demanderait $n-1$ multiplications par $x$, $n$ multiplications de $a_i$ par une puissance de $x$ et $n$ additions, pour un total de $3n-1$ opérations. Comparons avec l'*algorithme de Horner* :

$$\begin{cases} p_0 = a_n \\ p_i = p_{i-1}x + a_{n-i} \qquad (i = 1, \cdots, n) \\ P(x) = p_n \end{cases}, \tag{5.11}$$

qui ne requiert que $2n$ opérations.

Notons que (5.8) n'est pas nécessairement la représentation la plus adaptée d'un polynôme, car la valeur de $P(x)$ pour une valeur donnée de $x$ peut être très sensible à des erreurs sur les valeurs des $a_i$ [62]. Voir la remarque 5.5. □

Considérons l'interpolation polynomiale pour $x$ dans $[-1,1]$. N'importe quel intervalle non dégénéré $[a,b]$ peut être ramené à $[-1,1]$ par la transformation affine

$$x_{\text{normalisé}} = \frac{1}{b-a}(2x_{\text{initial}} - a - b), \tag{5.12}$$

de sorte que ceci n'est pas restrictif. Un point clé est la distribution des $x_j$ dans $[-1,1]$. Quand ils sont *régulièrement espacés*, l'interpolation devrait se limiter aux petites valeurs de $n$. Elle peut sinon conduire à des résultats inutilisables, avec des oscillations parasites connues sous le nom de *phénomène de Runge*. Ceci peut être évité en utilisant des *points de Tchebychev* [231, 232], par exemple des points de Tchebychev de seconde espèce, donnés par

$$x_j = \cos\frac{j\pi}{n}, \quad j = 0, 1, \cdots, n. \tag{5.13}$$

(L'interpolation par des splines (décrite en section 5.3.2), ou le krigeage (décrit en section 5.4.3) peuvent aussi être envisagés.)

Il existe plusieurs techniques pour calculer le polynôme interpolateur. Comme ce dernier est unique, elles sont mathématiquement équivalentes (mais leurs propriétés numériques diffèrent).

#### 5.3.1.1 Interpolation via la formule de Lagrange

La formule d'interpolation de Lagrange exprime $P_n(x)$ comme

$$P_n(x) = \sum_{j=0}^{n} \left( \prod_{k \neq j} \frac{x - x_k}{x_j - x_k} \right) y_j. \tag{5.14}$$

L'évaluation de $\mathbf{p}$ à partir des données est donc contournée. Il est facile de vérifier que $P_n(x_j) = y_j$ puisque, pour $x = x_j$, tous les produits dans (5.14) sont nuls sauf le $j$-ème, qui est égal à un. Malgré sa simplicité, (5.14) est rarement utilisée en pratique car elle est numériquement instable.

Une reformulation très utile de (5.14) est la formule d'*interpolation de Lagrange barycentrique*

$$P_n(x) = \frac{\sum_{j=0}^{n} \frac{w_j}{x - x_j} y_j}{\sum_{j=0}^{n} \frac{w_j}{x - x_j}}, \tag{5.15}$$

où les poids barycentriques satisfont

$$w_j = \frac{1}{\prod_{k \neq j}(x_j - x_k)}, \quad j = 0, 1, \cdots, n. \tag{5.16}$$

Ces poids ne dépendent que de la localisation des points d'évaluation $x_j$, pas des valeurs des données $y_j$ correspondantes. Ils peuvent donc être calculés une fois pour toutes pour une configuration des $x_j$ donnée. Le résultat est particulièrement simple pour les points de Tchebychev de seconde espèce, puisque

$$w_j = (-1)^j \delta_j, \quad j = 0, 1, \cdots, n, \tag{5.17}$$

avec tous les $\delta_j$ égaux à un, sauf $\delta_0 = \delta_n = 1/2$ [12].

L'interpolation de Lagrange barycentrique est tellement plus stable numériquement que (5.14) qu'elle est considérée comme l'une des meilleures méthodes d'interpolation polynomiale [101].

#### 5.3.1.2 Interpolation via la résolution d'un système linéaire

Quand le polynôme interpolateur est exprimé comme dans (5.8), son vecteur de paramètres **p** est la solution du système linéaire

$$\mathbf{A}\mathbf{p} = \mathbf{y}, \tag{5.18}$$

avec

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \tag{5.19}$$

et

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{5.20}$$

**A** est une matrice de *Vandermonde*, bien connue pour son mauvais conditionnement quand $n$ est grand.

*Remarque 5.4.* Le fait qu'une matrice de Vandermonde soit mal conditionnée ne signifie pas que le problème d'interpolation correspondant est insoluble. Avec des formulations alternatives appropriées, on peut construire des polynômes interpolateurs de degré très élevé. Ceci est spectaculairement illustré dans [231], où une fonction en dents de scie est interpolée avec un polynôme de degré 10 000 en des points de Tchebychev. Le graphe du polynôme interpolant, tracé à l'aide d'une mise

en œuvre astucieuse de la formule de Lagrange barycentrique qui ne demande que O($n$) flops pour évaluer $P_n(x)$, est indiscernable de celui de la fonction interpolée. □

*Remarque 5.5.* Tout polynôme de degré $n$ peut s'écrire

$$P_n(x,\mathbf{p}) = \sum_{i=0}^{n} a_i \phi_i(x), \tag{5.21}$$

où les $\phi_i(x)$ forment une base et où $\mathbf{p} = (a_0, \cdots, a_n)^{\mathrm{T}}$. L'équation (5.8) correspond à la *base en puissances*, où $\phi_i(x) = x^i$, et la représentation polynomiale résultante est appelée la *forme en série de puissances*. Pour n'importe quelle autre base polynomiale, les paramètres du polynôme interpolant sont obtenus en calculant la solution $\mathbf{p}$ de (5.18), avec (5.19) remplacée par

$$\mathbf{A} = \begin{bmatrix} 1 & \phi_1(x_0) & \phi_2(x_0) & \cdots & \phi_n(x_0) \\ 1 & \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_n(x_n) \end{bmatrix}. \tag{5.22}$$

On peut, par exemple, utiliser la *base de Legendre*, telle que

$$\begin{aligned} \phi_0(x) &= 1, \\ \phi_1(x) &= x, \\ (i+1)\phi_{i+1}(x) &= (2i+1)x\phi_i(x) - i\phi_{i-1}(x), \quad i = 1, \cdots, n-1. \end{aligned} \tag{5.23}$$

Comme

$$\int_{-1}^{1} \phi_i(\tau)\phi_j(\tau)\mathrm{d}\tau = 0 \tag{5.24}$$

chaque fois que $i \neq j$, les polynômes de Legendre sont orthogonaux sur $[-1,1]$, ce qui rend le système linéaire à résoudre mieux conditionné qu'avec la base en puissances. □

#### 5.3.1.3 Interpolation via l'algorithme de Neville

L'algorithme de Neville est particulièrement pertinent quand on veut connaître la valeur numérique de $P(x)$ pour une seule valeur numérique de $x$ (au lieu de chercher une expression analytique pour le polynôme). On l'utilise typiquement pour extrapoler en une valeur de $x$ pour laquelle il n'est pas possible d'évaluer directement $y = f(x)$ (voir la section 5.3.4).

Soit $P_{i,j}$ le polynôme de degré $j-i$ qui interpole $\{x_k, y_k\}$ pour $k = i, \cdots, j$. Le schéma de Horner peut être utilisé pour montrer que les polynômes interpolateurs satisfont l'équation de récurrence

$$P_{i,i}(x) = y_i, \quad i = 1, \cdots, n+1,$$

$$P_{i,j}(x) = \frac{1}{x_j - x_i}[(x_j - x)P_{i,j-1}(x) - (x - x_i)P_{i+1,j}(x)], \qquad 1 \leqslant i < j \leqslant n+1, \quad (5.25)$$

où $P_{1,n+1}(x)$ est le polynôme de degré $n$ qui interpole toutes les données.

### 5.3.2 Interpolation par des splines cubiques

Les *splines* sont des fonctions polynomiales par morceaux utilisées, par exemple, dans le contexte de la recherche de solutions approchées d'équations différentielles [23]. Les splines les plus simples et les plus utilisées sont les *splines cubiques* [227, 24], qui approximent la fonction $f(x)$ par un polynôme de degré trois sur chaque sous-intervalle d'un intervalle d'intérêt $[x_0, x_N]$. Ces polynômes sont assemblés de telle façon que leurs valeurs et celles de leurs deux premières dérivées coïncident là où ils se rejoignent. La fonction interpolatrice résultante est ainsi deux fois continûment différentiable.

Considérons $N+1$ données

$$\{x_i, y_i\}, \quad i = 0, \cdots, N, \tag{5.26}$$

et supposons que les coordonnées $x_i$ des *nœuds* (ou *points de rupture*) croissent avec $i$. Sur chaque sous-intervalle $\mathbb{I}_k = [x_k, x_{k+1}]$, on utilise un polynôme de degré trois

$$P_k(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3, \tag{5.27}$$

de sorte qu'il faut quatre contraintes indépendantes par polynôme. Puisque $P_k(x)$ doit être un interpolateur sur $\mathbb{I}_k$, il doit satisfaire

$$P_k(x_k) = y_k \tag{5.28}$$

et

$$P_k(x_{k+1}) = y_{k+1}. \tag{5.29}$$

Les dérivées premières des polynômes interpolateurs doivent prendre la même valeur à chaque extrémité commune de deux sous-intervalles, de sorte que

$$\dot{P}_k(x_k) = \dot{P}_{k-1}(x_k). \tag{5.30}$$

La dérivée seconde de $P_k(x)$ est affine en $x$, comme illustré par la figure 5.3. La formule d'interpolation de Lagrange se traduit par

$$\ddot{P}_k(x) = u_k \frac{x_{k+1} - x}{x_{k+1} - x_k} + u_{k+1} \frac{x - x_k}{x_{k+1} - x_k}, \tag{5.31}$$

ce qui entraîne

$$\ddot{P}_k(x_k) = \ddot{P}_{k-1}(x_k) = u_k. \tag{5.32}$$

Intégrons (5.31) deux fois pour obtenir

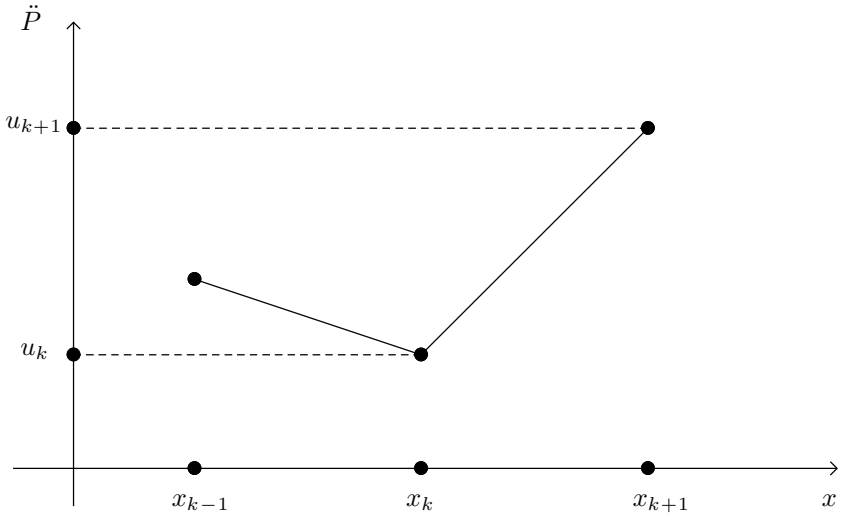


**Fig. 5.3** La dérivée seconde de l'interpolateur est affine par morceaux

$$P_k(x) = u_k \frac{(x_{k+1}-x)^3}{6h_{k+1}} + u_{k+1}\frac{(x-x_k)^3}{6h_{k+1}} + a_k(x-x_k) + b_k, \tag{5.33}$$

où $h_{k+1} = x_{k+1} - x_k$. Tenons compte de (5.28) et (5.29) pour obtenir les constantes d'intégration

$$a_k = \frac{y_{k+1}-y_k}{h_{k+1}} - \frac{h_{k+1}}{6}(u_{k+1}-u_k) \tag{5.34}$$

et

$$b_k = y_k - \frac{1}{6}u_k h_{k+1}^2. \tag{5.35}$$

$P_k(x)$ peut donc s'écrire

$$P_k(x) = \varphi\,(x, \mathbf{u}, \text{données})\,, \tag{5.36}$$

où $\mathbf{u}$ est le vecteur des $u_k$. Cette expression est cubique en $x$ et affine en $\mathbf{u}$.

Il y a $\dim \mathbf{u} = N+1$ inconnues, et $N-1$ conditions de continuité (5.30) (car il y a $N$ sous-intervalles $\mathbb{I}_k$), de sorte que deux contraintes supplémentaires sont nécessaires pour assurer l'unicité de $\mathbf{u}$. Dans les *splines cubiques naturelles*, ces contraintes sont $u_1 = u_N = 0$, ce qui revient à dire que la spline cubique est affine sur $(-\infty, x_0]$ et $[x_N, \infty)$. D'autres choix sont possibles ; on peut par exemple reproduire la dérivée première de $f(\cdot)$ en $x_0$ et en $x_N$, ou supposer que $f(\cdot)$ est périodique et telle que

$$f(x + x_N - x_0) \equiv f(x). \tag{5.37}$$

Les *splines cubiques périodiques* doivent alors satisfaire

$$P_0^{(r)}(x_0) = P_{N-1}^{(r)}(x_N), \quad r = 0,1,2. \tag{5.38}$$

Pour chacun de ces choix, le système d'équations linéaires résultant peut s'écrire

$$\mathbf{T}\bar{\mathbf{u}} = \mathbf{d}, \tag{5.39}$$

avec $\bar{\mathbf{u}}$ le vecteur des $u_i$ qui restent à évaluer et $\mathbf{T}$ tridiagonale, ce qui facilite grandement le calcul de $\bar{\mathbf{u}}$.

Soit $h_{\max}$ le plus grand des sous-intervalles $h_k$ entre nœuds. Quand $f(\cdot)$ est suffisamment douce, l'erreur d'interpolation d'une spline cubique naturelle est en $O(h_{\max}^4)$ pour tout $x$ dans un intervalle fermé qui tend vers $[x_0, x_N]$ quand $h_{\max}$ tend vers zéro [125].

### *5.3.3 Interpolation rationnelle*

L'interpolateur rationnel prend la forme

$$F(x,\mathbf{p}) = \frac{P(x,\mathbf{p})}{Q(x,\mathbf{p})}, \tag{5.40}$$

où $\mathbf{p}$ est un vecteur de paramètres à choisir pour imposer l'interpolation, et où $P(x,\mathbf{p})$ et $Q(x,\mathbf{p})$ sont des polynômes.

Si la représentation en série de puissances des polynômes est utilisée, alors

$$F(x,\mathbf{p}) = \frac{\sum_{i=0}^{p} a_i x^i}{\sum_{j=0}^{q} b_j x^j}, \tag{5.41}$$

avec $\mathbf{p} = (a_0, \cdots, a_p, b_0, \cdots, b_q)^{\mathrm{T}}$. Ceci implique que $F(x,\mathbf{p}) = F(x,\alpha\mathbf{p})$ pour tout $\alpha \neq 0$, et il en sera de même pour toute base polynomiale dans laquelle $P(x,\mathbf{p})$ et $Q(x,\mathbf{p})$ peuvent être exprimés. Il faut donc imposer une contrainte à $\mathbf{p}$ pour le rendre unique pour un interpolateur donné. On peut imposer $b_0 = 1$, par exemple.

Le principal avantage de l'interpolation rationnelle sur l'interpolation polynomiale est une flexibilité accrue puisque la classe des fonctions polynomiales n'est qu'une classe restreinte de fonctions rationnelles, avec un polynôme constant au dénominateur. Les fonctions rationnelles sont, par exemple, beaucoup plus aptes que les fonctions polynomiales à interpoler (ou approximer) des fonctions avec des pôles ou d'autres singularités au voisinage de ces singularités. De plus elles peuvent avoir des asymptotes horizontales ou verticales, contrairement aux fonctions polynomiales.

Bien qu'il y ait autant d'équations que d'inconnues, *il peut n'y avoir aucune solution*. Considérons, par exemple, la fonction rationnelle

$$F(x,a_0,a_1,b_1) = \frac{a_0 + a_1 x}{1 + b_1 x}. \tag{5.42}$$

Elle dépend de trois paramètres, et peut donc en principe être utilisée pour interpoler les valeurs prises par $f(x)$ en trois valeurs de $x$. Supposons que $f(x_0) = f(x_1) \neq f(x_2)$. Alors

$$\frac{a_0 + a_1 x_0}{1 + b_1 x_0} = \frac{a_0 + a_1 x_1}{1 + b_1 x_1}. \tag{5.43}$$

Ceci implique que $a_1 = a_0 b_1$ et que la fonction rationnelle se simplifie en

$$F(x, a_0, a_1, b_1) = a_0 = f(x_0) = f(x_1). \tag{5.44}$$

Elle est donc incapable de reproduire $f(x_2)$. Cette *simplification d'un pôle par un zéro* peut être évitée en rendant $f(x_0)$ légèrement différent de $f(x_1)$, ce qui remplace une interpolation par une approximation, et une simplification par une simplification approchée. Les simplifications approchées sont plutôt courantes quand on interpole des données réelles avec des fonctions rationnelles. Elles rendent le problème mal posé (les valeurs des coefficients de l'interpolateur deviennent très sensibles aux données).

Si l'interpolateur rationnel $F(x, \mathbf{p})$ est linéaire en les paramètres $a_i$ de son numérateur, il est non linéaire en les paramètres $b_j$ de son dénominateur. En général, les contraintes imposant l'interpolation

$$F(x_i, \mathbf{p}) = f(x_i), \quad i = 1, \cdots, n, \tag{5.45}$$

définissent donc un ensemble d'équations non linéaires en $\mathbf{p}$, dont la résolution semble requérir des outils comme ceux décrits au chapitre 7. Ce système peut cependant être transformé en un système linéaire en multipliant la $i$-ème équation de (5.45) par $Q(x_i, \mathbf{p})$ $(i = 1, \cdots, n)$ pour obtenir le système d'équations

$$Q(x_i, \mathbf{p}) f(x_i) = P(x_i, \mathbf{p}), \quad i = 1, \cdots, n, \tag{5.46}$$

qui est lui linéaire en $\mathbf{p}$. Rappelons qu'il faut imposer une contrainte à $\mathbf{p}$ pour rendre la solution génériquement unique, et qu'il faut se protéger contre le risque de simplification d'un pôle par un zéro, souvent en approximant les données plutôt que de les interpoler.

### *5.3.4 Extrapolation de Richardson*

Soit $R(h)$ la valeur approximative fournie par une méthode numérique pour un résultat mathématique $r$, avec $h > 0$ la taille du pas de cette méthode. Supposons que

$$r = \lim_{h \to 0} R(h), \tag{5.47}$$

mais qu'il soit impossible en pratique de faire tendre $h$ vers zéro, comme dans les deux exemples suivants.

*Exemple 5.4. Évaluation de dérivées*

Une des approximations par différences finies de la dérivée du premier ordre d'une fonction $f(\cdot)$ est

$$\dot{f}(x) \approx \frac{1}{h}[f(x+h) - f(x)] \tag{5.48}$$

(cf. chapitre 6). Mathématiquement, plus $h$ est petit et meilleure est l'approximation, mais rendre $h$ trop petit est *désastreux* dans les calculs à virgule flottante, car cela implique le calcul de la différence entre des nombres trop proches. □

*Exemple 5.5. Évaluation d'intégrales*

La méthode des rectangles peut être utilisée pour approximer une intégrale définie

$$\int_a^b f(\tau)\mathrm{d}\tau \approx \sum_i h f(a+ih). \tag{5.49}$$

Mathématiquement, plus $h$ est petit et meilleure est l'approximation, mais quand $h$ devient trop petit l'approximation demande trop de calculs. □

Puisque $h$ ne peut pas tendre vers zéro, l'utilisation de $R(h)$ à la place de $r$ induit une *erreur de méthode*, et l'extrapolation peut être utilisée pour améliorer la précision de l'évaluation de $r$. Supposons que

$$r = R(h) + \mathrm{O}(h^n), \tag{5.50}$$

où l'ordre $n$ de l'erreur de méthode est connu. Le *principe d'extrapolation de Richardson* exploite cette connaissance pour accroître la précision en combinant les résultats obtenus à différents pas. L'équation (5.50) implique que

$$r = R(h) + c_n h^n + c_{n+1} h^{n+1} + \cdots \tag{5.51}$$

Divisons le pas par deux, pour obtenir

$$r = R(\frac{h}{2}) + c_n \left(\frac{h}{2}\right)^n + c_{n+1} \left(\frac{h}{2}\right)^{n+1} + \cdots \tag{5.52}$$

Pour éliminer le terme d'ordre $n$, soustrayons (5.51) de $2^n$ fois (5.52). Il vient

$$(2^n - 1)r = 2^n R(\frac{h}{2}) - R(h) + \mathrm{O}(h^m), \tag{5.53}$$

avec $m > n$, ou de façon équivalente

$$r = \frac{2^n R(\frac{h}{2}) - R(h)}{2^n - 1} + \mathrm{O}(h^m). \tag{5.54}$$

Deux évaluations de $R$ ont donc permis de gagner *au moins* un ordre dans l'approximation. On peut pousser l'idée plus loin en évaluant $R(h_i)$ pour plusieurs valeurs de $h_i$ obtenues par divisions successives par deux d'une taille de pas initiale $h_0$. La valeur pour $h = 0$ du polynôme $P(h)$ extrapolant les données résultantes $(h_i, R(h_i))$

peut alors être calculée avec l'algorithme de Neville (voir la section 5.3.1.3). Dans le contexte de l'évaluation d'intégrales définies, ceci correspond à la méthode de Romberg, voir la section 6.2.2. L'extrapolation de Richardson est aussi utilisée, par exemple, en différentiation numérique (voir la section 6.4.3), ainsi que pour l'intégration d'équations différentielles ordinaires (voir la méthode de Bulirsch-Stoer en section 12.2.4.6).

Au lieu d'accroître la précision, on peut utiliser des idées similaires pour adapter la taille $h$ du pas de façon à maintenir une estimée de l'erreur de méthode à un niveau acceptable (voir la section 12.2.4).

## 5.4 Cas à plusieurs variables

Supposons maintenant qu'il y ait plusieurs variables d'entrée (aussi appelées *facteurs d'entrée*), formant un vecteur $\mathbf{x}$, et plusieurs variables de sortie, formant un vecteur $\mathbf{y}$. Le problème est alors dit MIMO (pour *multi-input multi-output*). Pour simplifier la présentation, nous nous limiterons à une seule sortie notée $y$, de sorte que le problème devient MISO (pour *multi-input single output*). Les problèmes MIMO peuvent toujours être scindés en autant de problèmes MISO qu'il y a de sorties, bien que ce ne soit pas forcément une bonne idée.

### *5.4.1 Interpolation polynomiale*

Dans l'interpolation polynomiale à plusieurs variables, chaque variable d'entrée apparaît comme une indéterminée du polynôme. Si, par exemple, il y a deux variables d'entrée $x_1$ et $x_2$ et si le degré total du polynôme est de deux, alors ce polynôme peut s'écrire

$$P(\mathbf{x},\mathbf{p}) = a_0 + a_1x_1 + a_2x_2 + a_3x_1^2 + a_4x_1x_2 + a_5x_2^2, \tag{5.55}$$

où

$$\mathbf{p} = (a_0, a_1, \cdots, a_5)^{\mathrm{T}}. \tag{5.56}$$

Ce polynôme reste linéaire en le vecteur $\mathbf{p}$ de ses coefficients inconnus, et il en sera ainsi quel que soit le degré du polynôme et le nombre de ses variables d'entrée. Les valeurs de ces coefficients peuvent donc toujours être calculées en résolvant le système d'équations linéaires qui impose l'interpolation, pourvu que ces équations soient en nombre suffisant. Le choix de la structure du polynôme (celui des monômes à y inclure) est loin d'être trivial, par contre.

### 5.4.2 Interpolation par splines

La présentation des splines cubiques dans le cas à une variable fait craindre que les splines multivariables ne soient une affaire compliquée. Les splines cubiques peuvent en fait être considérées comme un cas particulier du krigeage [242, 46], et le traitement du krigeage dans le cas à plusieurs variables est plutôt simple, au moins dans son principe.

### 5.4.3 Krigeage

Le nom de *krigeage* est un hommage au travail séminal de D.G. Krige sur les champs aurifères du Witwatersrand en Afrique du Sud, vers 1950 [131]. Cette technique fut développée et popularisée par G. Matheron, du *Centre de géostatistique* de l'*École des mines de Paris*, l'un des fondateurs de la *géostatistique* où elle joue un rôle central [46, 41, 241]. Initialement appliquée à des problèmes à deux ou trois dimensions, où les facteurs d'entrée correspondaient à des variables d'espace (comme dans la prospection minière), elle s'étend directement à des problèmes avec un nombre de dimensions bien supérieur (comme c'est souvent le cas en statistique industrielle).

Nous montrons ici, sans justification mathématique pour le moment, comment la version la plus simple du krigeage peut être utilisée pour de l'interpolation à plusieurs dimensions. Les équations qui suivent seront établies dans l'exemple 9.2.

Soit $y(\mathbf{x})$ la valeur de la sortie scalaire à prédire sur la base de la valeur prise par le vecteur d'entrée $\mathbf{x}$. Supposons qu'une série d'expériences (qui peuvent être des expériences sur ordinateur ou de vraies mesures physiques) ait fourni les valeurs de sortie

$$y_i = f(\mathbf{x}^i), \quad i = 1, \cdots, N, \tag{5.57}$$

pour $N$ valeurs numériques $\mathbf{x}^i$ du vecteur des entrées, et soit $\mathbf{y}$ le vecteur de ces valeurs de sortie. Notons que le sens de $\mathbf{y}$ ici diffère de celui dans (5.1). La prédiction par krigeage $\widehat{y}(\mathbf{x})$ de la valeur prise par $f(\mathbf{x})$ pour $\mathbf{x} \notin \{\mathbf{x}^i, i = 1, \cdots, N\}$ est linéaire en $\mathbf{y}$, et les poids de la combinaison linéaire dépendent de la valeur de $\mathbf{x}$. On peut donc écrire

$$\widehat{y}(\mathbf{x}) = \mathbf{c}^{\mathrm{T}}(\mathbf{x})\mathbf{y}. \tag{5.58}$$

Il semble naturel de supposer que plus $\mathbf{x}$ est proche de $\mathbf{x}^i$ et plus $f(\mathbf{x})$ ressemble à $f(\mathbf{x}^i)$. Ceci conduit à définir une fonction de corrélation $r(\mathbf{x}, \mathbf{x}^i)$ entre $f(\mathbf{x})$ et $f(\mathbf{x}^i)$ telle que

$$r(\mathbf{x}^i, \mathbf{x}^i) = 1 \tag{5.59}$$

et que $r(\mathbf{x}, \mathbf{x}^i)$ décroisse vers zéro quand la distance entre $\mathbf{x}$ et $\mathbf{x}^i$ augmente. Cette fonction de corrélation dépend souvent d'un vecteur $\mathbf{p}$ de paramètres à régler à partir des données disponibles. Elle sera alors notée $r(\mathbf{x}, \mathbf{x}^i, \mathbf{p})$.

*Exemple 5.6. Fonction de corrélation pour le krigeage*

Une fonction de corrélation paramétrée couramment utilisée est

$$r(\mathbf{x},\mathbf{x}^i,\mathbf{p}) = \prod_{j=1}^{\dim \mathbf{x}} \exp(-p_j|x_j - x_j^i|^2). \tag{5.60}$$

Les *paramètres de portée* $p_j > 0$ spécifient la vitesse avec laquelle l'influence de la mesure $y_i$ décroît quand la distance à $\mathbf{x}^i$ croît. Si $\mathbf{p}$ est trop grand alors l'influence des données décroît très vite et $\widehat{y}(\mathbf{x})$ tend vers zéro dès que $\mathbf{x}$ n'est plus dans le voisinage immédiat d'un $\mathbf{x}^i$. □

Supposons, pour simplifier, que la valeur de $\mathbf{p}$ ait été choisie auparavant, de sorte que $\mathbf{p}$ n'apparaît plus dans les équations. (Des méthodes statistiques sont disponibles pour estimer $\mathbf{p}$ à partir des données, voir la remarque 9.5.)

La prédiction par krigeage est gaussienne, et donc entièrement caractérisée (pour toute valeur donnée du vecteur d'entrée $\mathbf{x}$) par sa moyenne $\widehat{y}(\mathbf{x})$ et sa variance $\widehat{\sigma}^2(\mathbf{x})$. La *moyenne de la prédiction* est

$$\widehat{y}(\mathbf{x}) = \mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{y}, \tag{5.61}$$

où

$$\mathbf{R} = \begin{bmatrix} r(\mathbf{x}^1,\mathbf{x}^1) & r(\mathbf{x}^1,\mathbf{x}^2) & \cdots & r(\mathbf{x}^1,\mathbf{x}^N) \\ r(\mathbf{x}^2,\mathbf{x}^1) & r(\mathbf{x}^2,\mathbf{x}^2) & \cdots & r(\mathbf{x}^2,\mathbf{x}^N) \\ \vdots & \vdots & \ddots & \vdots \\ r(\mathbf{x}^N,\mathbf{x}^1) & r(\mathbf{x}^N,\mathbf{x}^2) & \cdots & r(\mathbf{x}^N,\mathbf{x}^N) \end{bmatrix} \tag{5.62}$$

et

$$\mathbf{r}^{\mathrm{T}}(\mathbf{x}) = \begin{bmatrix} r(\mathbf{x},\mathbf{x}^1) & r(\mathbf{x},\mathbf{x}^2) & \cdots & r(\mathbf{x},\mathbf{x}^N) \end{bmatrix}. \tag{5.63}$$

La *variance de la prédiction* est

$$\widehat{\sigma}^2(\mathbf{x}) = \sigma_y^2 \left[1 - \mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{r}(\mathbf{x})\right], \tag{5.64}$$

où $\sigma_y^2$ est une constante de proportionnalité, qui peut elle aussi être estimée à partir des données, voir la remarque 9.5.

*Remarque 5.6.* L'équation (5.64) permet de fournir des intervalles de confiance pour la prédiction ; sous l'hypothèse que le processus qui génère les données est gaussien de moyenne $\widehat{y}(\mathbf{x})$ et de variance $\widehat{\sigma}^2(\mathbf{x})$, la probabilité que $y(\mathbf{x})$ appartienne à l'intervalle

$$\mathbb{I}(\mathbf{x}) = [\widehat{y}(\mathbf{x}) - 2\widehat{\sigma}(\mathbf{x}), \widehat{y}(\mathbf{x}) + 2\widehat{\sigma}(\mathbf{x})] \tag{5.65}$$

est approximativement égale à 0.95. Le fait que le krigeage puisse accompagner ses prédictions d'un tel label de qualité se révèle très utile, en particulier dans le contexte de l'optimisation (voir la section 9.4.3). □

Dans la figure 5.4, il y a un seul facteur d'entrée $x \in [-1,1]$ par souci de lisibilité. Le graphe de la fonction $f(\cdot)$ à interpoler est tracé avec des tirets, et les points interpolés sont indiqués par des carrés. Le graphe de la prédiction $\widehat{y}(\mathbf{x})$ obtenue par

interpolation est en trait plein et la région de confiance à 95% pour cette prédiction est en gris. Il n'y a pas d'incertitude sur la prédiction aux points interpolés, et plus $x$ s'éloigne d'un point où $f(\cdot)$ a été évaluée plus la prédiction devient incertaine.

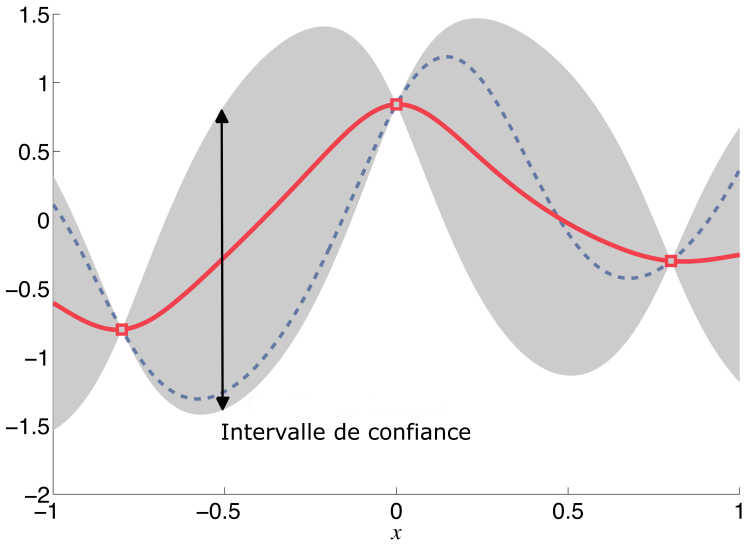


**Fig. 5.4** Interpolateur par krigeage (merci à Emmanuel Vazquez, de Supélec)

Puisque ni **R** ni **y** ne dépendent de **x**, on peut écrire

$$\widehat{y}(\mathbf{x}) = \mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{v}, \tag{5.66}$$

où

$$\mathbf{v} = \mathbf{R}^{-1}\mathbf{y} \tag{5.67}$$

peut être calculé une fois pour toute en résolvant le système d'équations linéaires

$$\mathbf{R}\mathbf{v} = \mathbf{y}. \tag{5.68}$$

Ceci simplifie grandement l'évaluation de $\widehat{y}(\mathbf{x})$ pour toute nouvelle valeur de **x**. Notons que (5.61) garantit qu'il y a interpolation, puisque

$$\mathbf{r}^{\mathrm{T}}(\mathbf{x}^i)\mathbf{R}^{-1}\mathbf{y} = (\mathbf{e}^i)^{\mathrm{T}}\mathbf{y} = y_i, \tag{5.69}$$

où $\mathbf{e}^i$ est la $i$-ème colonne de $\mathbf{I}_N$. Même si ceci est vrai pour n'importe quelle fonction de corrélation et n'importe quelle valeur de **p**, la structure de la fonction de corrélation et la valeur donnée à **p** influent sur la qualité de la prédiction.

La simplicité de (5.61), *valide quel que soit le nombre des facteurs d'entrée*, ne doit pas cacher que la résolution de (5.68) pour calculer **v** peut être un problème

mal conditionné. Une façon d'améliorer le conditionnement est de forcer $r(\mathbf{x},\mathbf{x}^i)$ à zéro quand la distance entre $\mathbf{x}$ et $\mathbf{x}^i$ dépasse un seuil $\delta$ à choisir, ce qui revient à dire que seules les paires $(y_i,\mathbf{x}^i)$ telles que $||\mathbf{x}-\mathbf{x}^i|| \leqslant \delta$ contribuent à $\widehat{y}(\mathbf{x})$. Ceci n'est faisable que s'il y a assez de $\mathbf{x}^i$ dans le voisinage de $\mathbf{x}$, ce que la *malédiction de la dimension* rend impossible quand $\dim \mathbf{x}$ est trop grand (voir l'exemple 8.6).

*Remarque 5.7.* Une légère modification des équations du krigeage transforme l'interpolation des données en leur approximation. Il suffit de remplacer $\mathbf{R}$ par

$$\mathbf{R}' = \mathbf{R} + \sigma_{\mathrm{m}}^2 \mathbf{I}, \tag{5.70}$$

où $\sigma_{\mathrm{m}}^2 > 0$. En théorie, $\sigma_{\mathrm{m}}^2$ devrait être égal à la variance de la prédiction en tout $\mathbf{x}^i$ où des mesures ont été effectuées, mais on peut le voir comme un paramètre de réglage supplémentaire. Cette transformation facilite aussi le calcul de $\mathbf{v}$ quand $\mathbf{R}$ est mal conditionnée, et c'est pourquoi un petit $\sigma_{\mathrm{m}}^2$ peut être utilisé même quand le bruit sur les données est négligeable. □

*Remarque 5.8.* Le krigeage peut aussi être utilisé pour estimer des dérivées et des intégrales [41, 234], ce qui fournit une alternative aux approches présentées au chapitre 6. □

## 5.5 Exemples MATLAB

La fonction

$$f(x) = \frac{1}{1+25x^2} \tag{5.71}$$

fut utilisée par Runge pour étudier les oscillations non désirées qu'on peut observer quand on interpole des données *recueillies en des points régulièrement espacés* avec un polynôme de degré élevé. Des données sont générées en $n+1$ points de ce type par le script

```
for i=1:n+1,
    x(i) = (2*(i-1)/n)-1;
    y(i) = 1/(1+25*x(i)^2);
end
```

Nous commençons par interpoler ces données grâce à `polyfit`, qui procède via la construction d'une matrice de Vandermonde, et à `polyval`, qui calcule la valeur prise par le polynôme interpolateur résultant sur une grille régulière fine spécifiée dans `FineX`, comme suit

```
N = 20*n
FineX = zeros(N,1);
for j=1:N+1,
    FineX(j) = (2*(j-1)/N)-1;
end
```

```
polynomial = polyfit(x,y,n);
fPoly = polyval(polynomial,FineX);
```

La figure 5.5 présente les résultats obtenus avec neuf points d'interpolation, en utilisant donc un polynôme de degré huit. Le graphe de la fonction interpolée est en trait plein, les points d'interpolation sont indiqués par des cercles et le graphe du polynôme interpolateur est en trait mixte. Accroître le degré du polynôme tout en gardant des points d'interpolation $x_i$ régulièrement espacés ne fait qu'aggraver la situation.

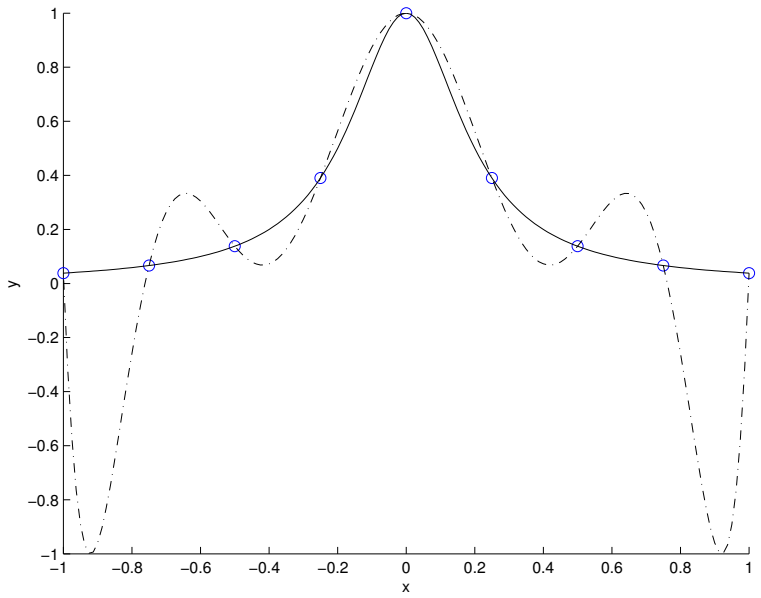


**Fig. 5.5** Interpolation polynomiale pour neuf valeurs régulièrement espacées de $x$ ; le graphe de la fonction interpolée est en trait plein

Une meilleure option consiste à remplacer les $x_i$ régulièrement espacés par des points de Tchebychev satisfaisant (5.13) et à générer les données par le script

```
for i=1:n+1,
    x(i) = cos((i-1)*pi/n);
    y(i) = 1/(1+25*x(i)^2);
end
```

Les résultats avec neuf points d'interpolation montrent encore quelques oscillations, mais nous pouvons maintenant sans danger accroître l'ordre du polynôme pour améliorer la situation. Avec 21 points d'interpolation nous obtenons les résultats de la figure 5.6.

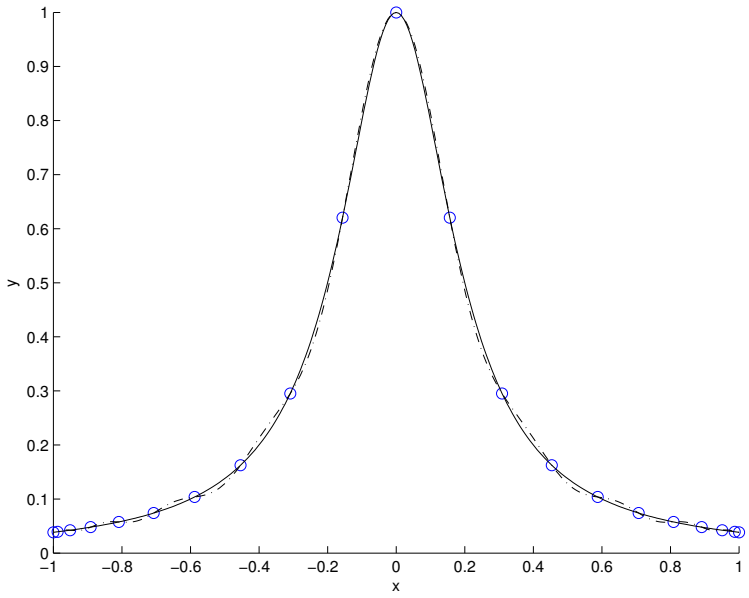


**Fig. 5.6** Interpolation polynomiale pour 21 valeurs de Tchebychev de $x$ ; le graphe de la fonction interpolée est en trait plein

Une option alternative est l'utilisation de splines cubiques. Ceci peut être mené à bien en utilisant les fonctions `spline` (qui calcule le polynôme par morceaux) et `ppval` (qui évalue ce polynôme par morceaux en des points à spécifier). On peut ainsi écrire

```
PieceWisePol = spline(x,y);
fCubicSpline = ppval(PieceWisePol,FineX);
```

Les résultats obtenus avec neuf $x_i$ régulièrement espacés sont présentés en figure 5.7.

## 5.6 En résumé

— Préférer l'interpolation à l'extrapolation, chaque fois que possible.
— L'interpolation peut être une mauvaise réponse à un problème d'approximation ; il ne sert à rien d'interpoler des données bruitées ou incertaines.
— L'interpolation polynomiale sur la base de données recueillies à des valeurs régulièrement espacées de la variable d'entrée doit être limitée à des polynômes de degré bas.

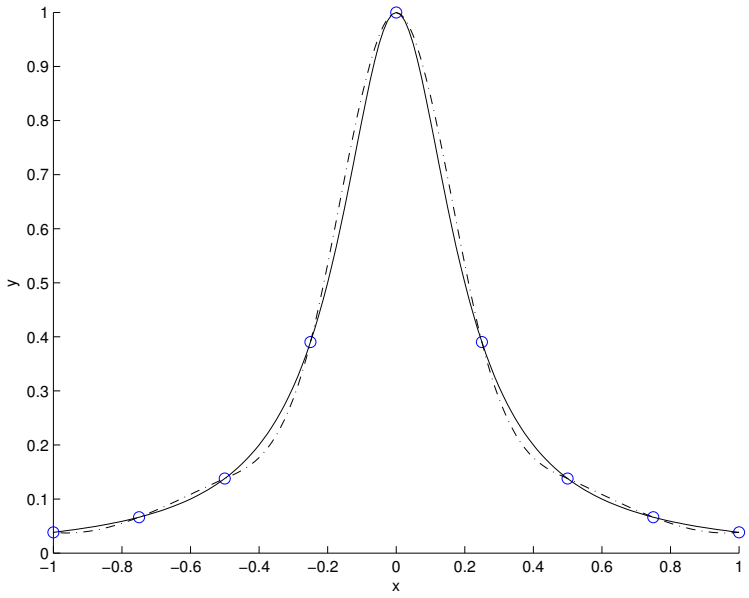


**Fig. 5.7** Interpolation par spline cubique pour neuf valeurs de $x$ régulièrement espacées ; le graphe de la fonction interpolée est en trait plein

— Quand les données sont recueillies en des points de Tchebychev, l'interpolation avec des polynômes de degré très élevé devient viable.
— L'expression explicite d'un polynôme interpolateur peut être obtenue en résolvant un système d'équations linéaires.
— Le conditionnement du système linéaire à résoudre pour obtenir les coefficients du polynôme interpolateur dépend de la base choisie pour le polynôme, et la base en puissances peut ne pas être la plus appropriée, car les matrices de Vandermonde sont mal conditionnées.
— L'interpolation de Lagrange est une des meilleures méthodes disponibles pour l'interpolation polynomiale, pourvu que sa variante barycentrique soit employée.
— L'interpolation par spline cubique requiert la résolution d'un système linéaire tridiagonal, ce qui peut être mené à bien de façon très efficace, même quand le nombre des points à interpoler est très grand.
— L'interpolation par des fonctions rationnelles est plus flexible que l'interpolation par des polynômes, mais plus compliquée (même si les équations qui imposent l'interpolation peuvent être rendues linéaires).
— Le principe d'extrapolation de Richardson est utilisé quand l'extrapolation ne peut être évitée, par exemple pour de l'intégration ou de la différentiation.

— Le krigeage utilise des formules simples pour l'interpolation (ou l'approximation) de fonctions de plusieurs variables, et propose une estimée de la qualité des prédictions qu'il fournit.

# Chapitre 6
# Intégrer et différentier des fonctions

Nous nous intéressons ici à l'intégration et à la différentiation de *fonctions* dans leurs aspects *numériques*. Quand ces fonctions ne sont connues qu'à travers les valeurs numériques qu'elles prennent pour certaines valeurs numériques de leurs arguments, il est hors de question de les intégrer ou de les différentier de façon formelle par calcul symbolique sur ordinateur. La section 6.6.2 montera cependant que quand le source du code évaluant la fonction est disponible, une *différentiation automatique* devient possible, qui implique des traitements formels. L'intégration d'équations différentielles sera considérée aux chapitres 12 et 13.

*Remarque 6.1.* Quand on dispose d'une expression symbolique explicite pour une fonction, on peut penser à l'intégrer ou à la différentier par calcul symbolique sur ordinateur. Les langages de calcul symbolique comme Maple et Mathematica incluent des méthodes d'intégration formelle tellement pénibles à utiliser à la main qu'elles ne sont même pas enseignées dans les cours de calcul avancés. Ces langages facilitent aussi grandement l'évaluation de dérivées.

Le script qui suit, par exemple, utilise la *Symbolic Math Toolbox* de MATLAB pour évaluer les fonctions gradient et hessienne d'une fonction scalaire de plusieurs variables.

```
syms x y
X = [x;y]
F = x^3*y^2-9*x*y+2
G = gradient(F,X)
H = hessian(F,X)
```

Il produit

```
X =
 x
 y

F =
x^3*y^2 - 9*x*y + 2
```

```
G =
 3*x^2*y^2 - 9*y
   2*y*x^3 - 9*x

H =
[      6*x*y^2, 6*y*x^2 - 9]
[ 6*y*x^2 - 9,        2*x^3]
```

Pour des fonctions vectorielles de plusieurs variables, on peut générer de même des fonctions jacobiennes (voir la remarque 7.10).

Nous ne supposons pas ici l'existence de telles expressions symboliques explicites des fonctions à intégrer ou à différentier. □

## 6.1 Exemples

*Exemple 6.1. Navigation inertielle*

Les systèmes de navigation inertielle sont utilisés, entre autres, dans les avions et les sous-marins. Ils comprennent des accéléromètres qui mesurent les accélérations le long de trois axes indépendants (disons, la longitude, la latitude et l'altitude). Pourvu que les conditions initiales soient connues, on obtient les trois composantes de la vitesse en intégrant ces accélérations, puis les trois composantes de la position en intégrant ces vitesses. Des accéléromètres peu coûteux, rendus possibles par les MEMS (pour *micro electromechanical systems*), ont trouvé leur place dans les téléphones, les consoles de jeu et autres dispositifs électroniques personnels. □

*Exemple 6.2. Calcul de puissance*

La puissance $P$ consommée par un appareil électrique (en W) est donnée par

$$P = \frac{1}{T}\int_0^T u(\tau)i(\tau)\mathrm{d}\tau, \tag{6.1}$$

où la tension $u$ aux bornes de l'appareil (en V) est sinusoïdale de période $T$, et où (éventuellement après une phase transitoire) le courant $i$ à travers l'appareil (en A) est lui aussi périodique de période $T$, mais pas nécessairement sinusoïdal. Estimer la valeur de $P$ à partir de mesures de $u(t_k)$ et $i(t_k)$ à des instants $t_k \in [0,T]$, $k = 1,\cdots,N$, revient à évaluer une intégrale. □

*Exemple 6.3. Évaluation de vitesse*

Calculer la vitesse d'un mobile à partir de mesures de sa position revient à différentier un signal dont la valeur est connue à des instants discrets. (Quand un modèle du comportement dynamique du mobile est disponible, on peut en tenir compte en utilisant un *filtre de Kalman* [116, 26], non considéré ici.) □

*Exemple 6.4. Intégration d'équations différentielles*

La méthode des différences finies pour intégrer des équations différentielles ordinaires ou aux dérivées partielles exploite des formules de différentiation numérique. Voir les chapitres 12 et 13. □

## 6.2 Intégrer des fonctions d'une variable

Considérons l'intégrale définie

$$I = \int_a^b f(x)\mathrm{d}x, \tag{6.2}$$

où les limites $a$ et $b$ ont des valeurs numériques connues et où l'intégrande $f(\cdot)$, une fonction réelle supposée intégrable, peut être évaluée numériquement en tout $x$ appartenant à l'intervalle $[a,b]$. Évaluer $I$ est aussi appelé *quadrature*, en souvenir de la méthode qui procède en approximant des surfaces par des unions de petits carrés.

Pour tout $c$ dans $[a,b]$, par exemple en son milieu,

$$\int_a^b f(x)\mathrm{d}x = \int_a^c f(x)\mathrm{d}x + \int_c^b f(x)\mathrm{d}x. \tag{6.3}$$

L'évaluation de $I$ peut donc être scindée récursivement en sous-tâches pour en améliorer la précision, avec une approche *diviser pour régner*. Cette stratégie de *quadrature adaptative* permet de s'adapter aux propriétés locales de l'intégrande $f(\cdot)$ en mettant plus d'évaluations là où $f(\cdot)$ varie vite.

La décision sur l'opportunité de couper $[a,b]$ en deux est en général prise sur la base d'une comparaison entre les résultats numériques $I_+$ et $I_-$ de l'évaluation de $I$ par deux méthodes d'intégration numérique, avec $I_+$ supposée plus précise que $I_-$. Si

$$\frac{|I_+ - I_-|}{|I_+|} < \delta, \tag{6.4}$$

où $\delta$ est une tolérance à choisir sur l'erreur relative, alors le résultat $I_+$ de la meilleure méthode est gardé. Sinon, on peut couper $[a,b]$ en deux et appliquer la même procédure aux deux sous-intervalles résultants. Pour éviter des bissections sans fin, on limite le nombre des récursions et aucune bissection n'est effectuée sur tout sous-intervalle dont la contribution relative à $I$ est considérée comme trop petite. Voir [67] pour une comparaison de stratégies de quadrature adaptative, qui montre qu'aucune d'entre elles ne donne des résultats précis pour toutes les fonctions intégrables.

L'intervalle $[a,b]$ considéré dans ce qui suit peut être l'un des sous-intervalles résultants de l'approche diviser pour régner.

### 6.2.1 Méthodes de Newton-Cotes

Dans les méthodes de Newton-Cotes, $f(\cdot)$ est évaluée en $N+1$ points *régulièrement espacés* $x_i$, $i = 0, \cdots, N$, tels que $x_0 = a$ et $x_N = b$, de sorte que

$$x_i = a + ih, \quad i = 0, 1, \cdots, N, \tag{6.5}$$

avec

$$h = \frac{b-a}{N}. \tag{6.6}$$

L'intervalle $[a,b]$ est partitionné en sous-intervalles de même largeur $kh$, et $k$ doit donc diviser $N$. Chaque sous-intervalle contient $k+1$ points d'évaluation, ce qui permet de remplacer $f(\cdot)$ sur ce sous-intervalle par un polynôme interpolateur de degré $k$. La valeur de l'intégrale définie $I$ est alors approximée par la somme des intégrales des polynômes interpolateurs sur les sous-intervalles pour lesquels ils sont utilisés.

*Remarque 6.2.* Le problème initial a ainsi été remplacé par un problème approché qui peut être résolu de façon exacte (au moins d'un point de vue mathématique). □

*Remarque 6.3.* Espacer les points d'évaluation de façon régulière n'est pas forcément une si bonne idée que cela (voir les sections 6.2.3 et 6.2.4). □

L'intégrale du polynôme interpolateur sur le sous-intervalle $[x_0, x_k]$ peut s'écrire

$$I_{\mathrm{SI}}(k) = h \sum_{j=0}^{k} c_j f(x_j), \tag{6.7}$$

où les coefficients $c_j$ dépendent seulement de l'ordre $k$ du polynôme, et la même formule s'applique à tous les autres sous-intervalles, après incrémentation des indices.

Dans ce qui suit, la méthode de Newton-Cotes fondée sur un polynôme interpolateur d'ordre $k$ est notée NC($k$), et $f(x_j)$ est noté $f_j$. Comme les $x_j$ sont régulièrement répartis, $k$ doit être petit. L'erreur de méthode *locale* commise par NC($k$) sur $[x_0, x_k]$ est

$$e_{\mathrm{NC}}(k) = \int_{x_0}^{x_k} f(x)\mathrm{d}x - I_{\mathrm{SI}}(k), \tag{6.8}$$

et l'erreur de méthode *globale* sur $[a,b]$, notée $E_{\mathrm{NC}}(k)$, est obtenue en sommant les erreurs de méthode locales commises sur tous les sous-intervalles.

Des preuves des résultats concernant les valeurs de $e_{\mathrm{NC}}(k)$ et $E_{\mathrm{NC}}(k)$ présentés ci-dessous peuvent être trouvées dans [64]. Dans ces résultats, $f(\cdot)$ est bien sûr supposée différentiable jusqu'à l'ordre requis.

#### 6.2.1.1 NC(1) : Méthode des trapèzes

Un polynôme interpolateur du premier ordre requiert deux points d'évaluation par sous-intervalle (un à chaque extrémité). L'interpolation est alors affine par morceaux, et l'intégrale de $f(\cdot)$ sur $[x_0, x_1]$ est évaluée par la règle des trapèzes

$$I_{\mathrm{SI}} = \frac{h}{2}(f_0 + f_1) = \frac{b-a}{2N}(f_0 + f_1). \tag{6.9}$$

Toutes les extrémités des sous-intervalles sont utilisées deux fois lors de l'évaluation de $I$, sauf $x_0$ et $x_N$, de sorte que

$$I \approx \frac{b-a}{N}\left(\frac{f_0 + f_N}{2} + \sum_{i=1}^{N-1} f_i\right). \tag{6.10}$$

L'erreur de méthode locale satisfait

$$e_{\mathrm{NC}}(1) = -\frac{1}{12}\ddot{f}(\eta)h^3, \tag{6.11}$$

pour un $\eta \in [x_0, x_1]$, et l'erreur de méthode globale est telle que

$$E_{\mathrm{NC}}(1) = -\frac{b-a}{12}\ddot{f}(\zeta)h^2, \tag{6.12}$$

pour un $\zeta \in [a, b]$. L'erreur de méthode globale sur $I$ est donc en $\mathrm{O}(h^2)$. Si $f(\cdot)$ est un polynôme de degré au plus égal à un, alors $\ddot{f}(\cdot) \equiv 0$ et il n'y a pas d'erreur de méthode, ce qui n'a rien de surprenant.

*Remarque 6.4.* La règle des trapèzes peut aussi être utilisée avec des $x_i$ irrégulièrement espacés, en posant

$$I \approx \frac{1}{2}\sum_{i=0}^{N-1}(x_{i+1} - x_i)\cdot(f_{i+1} + f_i). \tag{6.13}$$

□

#### 6.2.1.2 NC(2) : Méthode de Simpson 1/3

Un polynôme interpolateur du second degré requiert trois points d'évaluation par sous-intervalle. L'interpolation est alors parabolique par morceaux, et

$$I_{\mathrm{SI}} = \frac{1}{3}h(f_0 + 4f_1 + f_2). \tag{6.14}$$

Le 1/3 dans le nom de la méthode vient du coefficient de tête dans (6.14). On peut montrer que

$$e_{\mathrm{NC}}(2) = -\frac{1}{90}f^{(4)}(\eta)h^5, \tag{6.15}$$

pour un $\eta \in [x_0, x_2]$, et que

$$E_{\mathrm{NC}}(2) = -\frac{b-a}{180} f^{(4)}(\zeta) h^4, \tag{6.16}$$

pour un $\zeta \in [a,b]$. L'erreur de méthode globale sur $I$ avec NC(2) est donc en $\mathrm{O}(h^4)$, ce qui est bien mieux qu'avec NC(1). A cause d'une *simplification heureuse*, il n'y a pas d'erreur de méthode si $f(\cdot)$ est un polynôme de degré au plus égal à trois, et pas juste deux comme on aurait pu s'y attendre.

#### 6.2.1.3 NC(3) : Méthode de Simpson 3/8

Un polynôme interpolateur de degré trois conduit à

$$I_{\mathrm{SI}} = \frac{3}{8} h(f_0 + 3f_1 + 3f_2 + f_3). \tag{6.17}$$

Le 3/8 dans le nom de la méthode vient du coefficient de tête dans (6.17). On peut montrer que

$$e_{\mathrm{NC}}(3) = -\frac{3}{80} f^{(4)}(\eta) h^5, \tag{6.18}$$

pour un $\eta \in [x_0, x_3]$, et que

$$E_{\mathrm{NC}}(3) = -\frac{b-a}{80} f^{(4)}(\zeta) h^4, \tag{6.19}$$

pour un $\zeta \in [a,b]$. L'erreur globale de méthode sur $I$ avec NC(3) est donc en $\mathrm{O}(h^4)$, comme avec NC(2), et on ne semble pas avoir gagné quoi que ce soit en augmentant l'ordre du polynôme interpolateur. Comme avec NC(2), il n'y a pas d'erreur de méthode si $f(\cdot)$ est un polynôme de degré au plus égal à trois, mais pour NC(3) ceci n'a plus rien de surprenant.

#### 6.2.1.4 NC(4) : Méthode de Boole

Un polynôme interpolateur d'ordre quatre conduit à

$$I_{\mathrm{SI}} = \frac{2}{45} h(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4). \tag{6.20}$$

On peut montrer que

$$e_{\mathrm{NC}}(4) = -\frac{8}{945} f^{(6)}(\eta) h^7, \tag{6.21}$$

pour un $\eta \in [x_0, x_4]$, et que

$$E_{\mathrm{NC}}(4) = -\frac{2(b-a)}{945} f^{(6)}(\zeta) h^6, \tag{6.22}$$

pour un $\zeta \in [a,b]$. L'erreur de méthode globale sur $I$ avec NC(4) est donc en $O(h^6)$. Grâce à une nouvelle simplification heureuse, il n'y a pas d'erreur de méthode si $f(\cdot)$ est un polynôme de degré au plus égal à cinq.

*Remarque 6.5.* Un examen rapide des formules précédentes pourrait suggérer que

$$E_{NC}(k) = \frac{b-a}{kh} e_{NC}(k), \tag{6.23}$$

ce qui semble naturel puisqu'il y a $(b-a)/kh$ sous-intervalles. Notons cependant que la valeur de $\zeta$ dans l'expression de $E_{NC}(k)$ diffère en général de celle de $\eta$ dans l'expression de $e_{NC}(k)$. □

#### 6.2.1.5 Régler le pas d'une méthode de Newton-Cotes

La taille $h$ du pas doit être suffisamment petite pour que la précision soit acceptable, mais pas trop petite car cela augmenterait inutilement le nombre des opérations. La procédure suivante peut être utilisée pour évaluer l'erreur de méthode et la maintenir à un niveau acceptable en réglant la taille du pas.

Soit $\widehat{I}(h,m)$ la valeur obtenue pour $I$ quand la taille du pas est $h$ et l'erreur de méthode est en $O(h^m)$. Alors,

$$I = \widehat{I}(h,m) + O(h^m). \tag{6.24}$$

En d'autres termes,

$$I = \widehat{I}(h,m) + c_1 h^m + c_2 h^{m+1} + \cdots \tag{6.25}$$

Quand on divise la taille du pas par deux, il vient

$$I = \widehat{I}\left(\frac{h}{2},m\right) + c_1 \left(\frac{h}{2}\right)^m + c_2 \left(\frac{h}{2}\right)^{m+1} + \cdots \tag{6.26}$$

Au lieu de combiner (6.25) et (6.26) pour éliminer le premier terme de l'erreur de méthode, comme le suggérerait l'extrapolation de Richardson, utilisons la différence entre $\widehat{I}(h,m)$ et $\widehat{I}\left(\frac{h}{2},m\right)$ pour obtenir une estimée grossière de l'erreur de méthode globale pour le plus petit pas. Soustrayons (6.26) de (6.25) pour obtenir

$$\widehat{I}(h,m) - \widehat{I}\left(\frac{h}{2},m\right) = c_1 \left(\frac{h}{2}\right)^m (1-2^m) + O(h^k), \tag{6.27}$$

avec $k > m$. Ainsi,

$$c_1 \left(\frac{h}{2}\right)^m = \frac{\widehat{I}(h,m) - \widehat{I}\left(\frac{h}{2},m\right)}{1-2^m} + O(h^k). \tag{6.28}$$

Cette estimée peut être utilisée pour décider s'il est souhaitable de rediviser par deux la taille du pas. Une procédure similaire peut être employée pour adapter la taille du

pas dans le contexte de la résolution d'équations différentielles ordinaires (voir la section 12.2.4.2).

### 6.2.2 Méthode de Romberg

La méthode de Romberg consiste à appliquer l'extrapolation de Richardson de façon répétée à NC(1). Soit $\widehat{I}(h)$ l'approximation de l'intégrale $I$ calculée par NC(1) pour une taille de pas $h$. La méthode de Romberg calcule $\widehat{I}(h_0), \widehat{I}(h_0/2), \widehat{I}(h_0/4)\ldots$, et un polynôme $P(h)$ interpolant ces résultats. $I$ est alors approximée par $P(0)$.

Si $f(\cdot)$ est suffisamment régulière, le terme d'erreur de méthode dans NC(1) ne contient que des termes pairs en $h$, de sorte que

$$E_{\mathrm{NC}}(1) = \sum_{i \geqslant 1} c_{2i} h^{2i}. \tag{6.29}$$

Chaque pas d'extrapolation augmente donc de deux l'ordre de l'erreur de méthode, avec des erreurs en $\mathrm{O}(h^4)$, $\mathrm{O}(h^6)$, $\mathrm{O}(h^8)$... Ceci permet d'obtenir rapidement des résultats très précis.

Soit $R(i,j)$ la valeur de l'intégrale (6.2) comme évaluée par la méthode de Romberg après $j$ pas d'extrapolation de Richardson à base d'une intégration avec un pas constant de taille

$$h_i = \frac{b-a}{2^i}. \tag{6.30}$$

$R(i,0)$ correspond alors à NC(1), et

$$R(i,j) = \frac{1}{4^j - 1}[4^j R(i,j-1) - R(i-1,j-1)]. \tag{6.31}$$

Cette formule est à comparer avec (5.54), où le fait qu'il n'y a pas de terme impair dans l'erreur de méthode n'est pas pris en compte. L'erreur de méthode pour $R(i,j)$ est en $\mathrm{O}(h_i^{2j+2})$. $R(i,1)$ correspond à la méthode de Simpson 1/3 et $R(i,2)$ à la méthode de Boole. Quand $j > 2$, $R(i,j)$ tend à être plus stable que la méthode de Newton-Cotes correspondante.

### 6.2.3 Quadrature gaussienne

Contrairement aux méthodes d'intégration présentées jusqu'ici, la quadrature gaussienne ne requiert pas des points d'évaluation régulièrement espacés sur l'horizon d'intégration $[a,b]$, et tire avantage des degrés de liberté qui en résultent. L'intégrale (6.2) est approximée par

$$I \approx \sum_{i=1}^{N} w_i f(x_i), \tag{6.32}$$

qui a $2N$ paramètres, à savoir les $N$ points d'évaluation $x_i$ et les poids associés $w_i$. Puisqu'un polynôme d'ordre $2N-1$ a $2N$ coefficients, il devient ainsi possible d'imposer que (6.32) n'entraîne aucune erreur de méthode si $f(\cdot)$ est un polynôme de degré au plus égal à $2N-1$. Comparez avec les méthodes de Newton-Cotes.

Gauss a montré que les points d'évaluation $x_i$ dans (6.32) correspondaient aux racines du polynôme de Legendre de degré $N$ [227]. Ces racines ne sont pas triviales à calculer avec précision pour $N$ grand [81], mais elles ont été tabulées. Une fois les points d'évaluation connus, il est beaucoup plus facile d'obtenir les poids correspondants. Le tableau 6.1 donne les valeurs des $x_i$ et $w_i$ pour jusqu'à cinq évaluations de $f(\cdot)$ sur un intervalle normalisé $[-1,1]$. Les résultats pour jusqu'à seize évaluations de $f(\cdot)$ sont dans [146, 147].

**Tableau 6.1** Quadrature gaussienne

| nombre d'évaluations | points d'évaluation $x_i$ | poids $w_i$ |
|---|---|---|
| 1 | 0 | 2 |
| 2 | $\pm 1/\sqrt{3}$ | 1 |
| 3 | 0 | 8/9 |
| | ±0.774596669241483 | 5/9 |
| 4 | ±0.339981043584856 | 0.652145154862546 |
| | ±0.861136311594053 | 0.347854845137454 |
| 5 | 0 | 0.568888888888889 |
| | ±0.538469310105683 | 0.478628670499366 |
| | ±0.906179845938664 | 0.236926885056189 |

Les valeurs de $x_i$ et $w_i$ $(i = 1, \cdots, N)$ dans le tableau 6.1 sont des solutions approchées du système d'équations non linéaires traduisant le fait que

$$\int_{-1}^{1} f(x)\mathrm{d}x = \sum_{i=1}^{N} w_i f(x_i) \tag{6.33}$$

pour $f(x) \equiv 1$, $f(x) \equiv x$, et ainsi de suite jusqu'à $f(x) \equiv x^{2N-1}$. La première de ces équations implique que

$$\sum_{i=1}^{N} w_i = 2. \tag{6.34}$$

*Exemple 6.5.* Pour $N = 1$, $x_1$ et $w_1$ doivent être tels que

$$\int_{-1}^{1} \mathrm{d}x = 2 = w_1, \tag{6.35}$$

et

$$\int_{-1}^{1} x\mathrm{d}x = 0 = w_1 x_1 \Rightarrow x_1 = 0. \tag{6.36}$$

On doit donc évaluer $f(\cdot)$ au centre de l'intervalle normalisé et multiplier le résultat par deux pour obtenir une estimée de l'intégrale. C'est la *formule du point milieu*, exacte pour l'intégration de polynôme jusqu'à l'ordre un. La méthode des trapèzes requiert deux évaluations de $f(\cdot)$ pour atteindre les mêmes performances. □

*Remarque 6.6.* Chaque fois que $a < b$, le changement de variable

$$x = \frac{(b-a)\tau + a + b}{2} \tag{6.37}$$

transforme $\tau \in [-1,1]$ en $x \in [a,b]$. De plus

$$I = \int_a^b f(x)\mathrm{d}x = \int_{-1}^1 f\left(\frac{(b-a)\tau + a + b}{2}\right)\left(\frac{b-a}{2}\right)\mathrm{d}\tau, \tag{6.38}$$

de sorte que

$$I = \left(\frac{b-a}{2}\right)\int_{-1}^1 g(\tau)\mathrm{d}\tau, \tag{6.39}$$

avec

$$g(\tau) = f\left(\frac{(b-a)\tau + a + b}{2}\right). \tag{6.40}$$

L'utilisation d'un intervalle normalisé dans la table 6.1 n'est donc pas restrictive. □

Une variante est la *quadrature de Gauss-Lobatto*, où $x_1 = a$ et $x_N = b$. Le fait d'évaluer l'intégrande aux extrémités de l'intervalle d'intégration facilite les raffinements itératifs où un intervalle d'intégration peut être coupé en sous-intervalles d'une façon telle que des points d'évaluation antérieurs deviennent des extrémités des nouveaux sous-intervalles. La quadrature de Gauss-Lobatto n'introduit pas d'erreur de méthode si l'intégrande est un polynôme de degré au plus $2N-3$ (au lieu de $2N-1$ pour la quadrature de Gauss ; c'est le prix à payer pour la perte de deux degrés de liberté).

### 6.2.4 Intégrer via la résolution d'une EDO

La quadrature gaussienne manque encore de flexibilité quant à où il convient d'évaluer l'intégrande $f(\cdot)$. Une alternative attractive consiste à résoudre l'équation différentielle ordinaire (EDO)

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x), \tag{6.41}$$

avec les conditions initiales $y(a) = 0$, pour obtenir

$$I = y(b). \tag{6.42}$$

Les méthodes d'intégration d'EDO à taille de pas adaptative permettent de faire varier la distance entre points d'évaluation consécutifs pour avoir plus de points là où l'intégrande varie vite. Voir le chapitre 12.

## 6.3 Intégrer des fonctions de plusieurs variables

Considérons maintenant l'intégrale définie

$$I = \int_{\mathbb{D}} f(\mathbf{x}) \mathrm{d}\mathbf{x}, \tag{6.43}$$

où $f(\cdot)$ est une fonction de $\mathbb{D} \subset \mathbb{R}^n$ vers $\mathbb{R}$, et où $\mathbf{x}$ est un vecteur de $\mathbb{R}^n$. L'évaluation de $I$ est beaucoup plus compliquée que pour les fonctions d'une seule variable, car

— elle demande beaucoup plus d'évaluations de $f(\cdot)$ (si l'on utilisait une grille régulière, il faudrait typiquement $m^n$ évaluations au lieu de $m$ dans le cas à une variable),

— la forme de $\mathbb{D}$ peut être beaucoup plus complexe qu'un simple intervalle ($\mathbb{D}$ peut par exemple être une union d'ensembles non convexes disjoints).

Les deux méthodes présentées ci-dessous peuvent être vues comme complémentaires.

### *6.3.1 Intégrations à une dimension imbriquées*

Supposons, pour simplifier, que $n = 2$ et que $\mathbb{D}$ soit comme indiqué sur la figure 6.1. L'intégrale définie $I$ peut alors s'exprimer comme

$$I = \int_{y_1}^{y_2} \int_{x_1(y)}^{x_2(y)} f(x,y) \mathrm{d}x \mathrm{d}y, \tag{6.44}$$

de sorte qu'on peut faire des intégrations internes à une dimension par rapport à $x$ pour un nombre suffisant de valeurs de $y$ suivies d'une intégration externe à une dimension par rapport à $y$. Comme dans le cas à une variable, il devrait y avoir plus d'évaluations numériques de l'intégrande $f(\cdot,\cdot)$ dans les régions où celui-ci varie vite.

### *6.3.2 Intégration de Monte-Carlo*

L'imbrication d'intégrations à une dimension n'est viable que si $f(\cdot)$ est suffisamment douce et si la dimension $n$ de $\mathbf{x}$ est faible. De plus, sa mise en œuvre n'est pas triviale. L'intégration de Monte-Carlo est beaucoup plus simple à mettre en

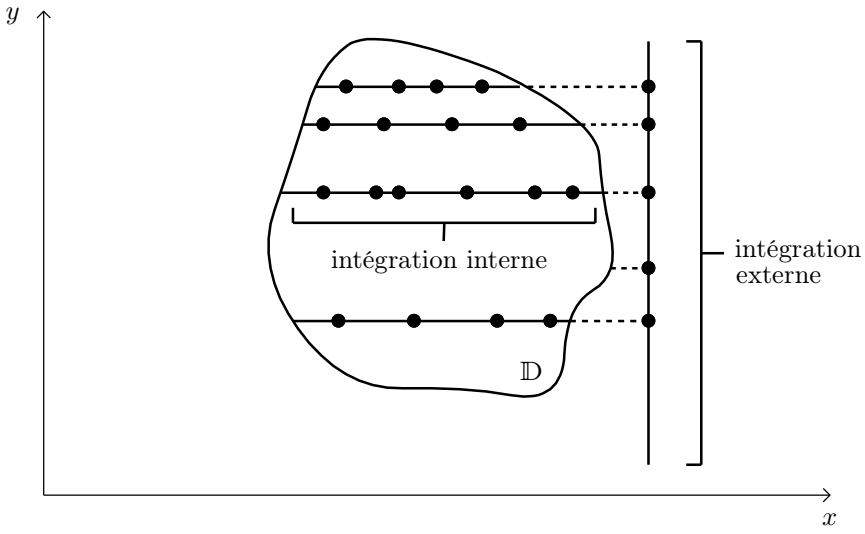


**Fig. 6.1** Intégrations 1D imbriquées

œuvre, s'applique aussi aux fonctions discontinues et est particulièrement efficace quand la dimension de **x** est élevée.

#### 6.3.2.1 La forme du domaine est simple

Supposons, pour commencer, la forme de $\mathbb{D}$ si simple qu'il soit facile de calculer son volume $V_{\mathbb{D}}$ et de tirer des valeurs $\mathbf{x}^i$ ($i = 1, \cdots, N$) de **x** au hasard dans $\mathbb{D}$ avec une distribution uniforme. (La génération de bons nombres pseudo-aléatoires est en fait un problème difficile et important [130, 186] qui ne sera pas abordé ici. Voir le chapitre 9 de [158] et les références qu'il contient pour une description de la façon dont ce problème a été traité dans les versions successives de MATLAB.) Alors

$$I \approx V_{\mathbb{D}} < f >, \tag{6.45}$$

où $< f >$ est la moyenne empirique de $f(\cdot)$ sur les $N$ valeurs de **x** en lesquelles $f(\cdot)$ a été évaluée

$$< f >= \frac{1}{N} \sum_{i=1}^{N} f\left(\mathbf{x}^i\right). \tag{6.46}$$

#### 6.3.2.2 La forme du domaine est complexe

Quand $V_{\mathbb{D}}$ ne peut pas être calculé analytiquement, on peut enfermer $\mathbb{D}$ dans un domaine de forme simple $\mathbb{E}$ et de volume connu $\mathbb{V}_{\mathbb{E}}$, tirer des valeurs $\mathbf{x}^i$ de $\mathbf{x}$ au hasard dans $\mathbb{E}$ avec une distribution uniforme et évaluer $V_{\mathbb{D}}$ comme

$$V_{\mathbb{D}} \approx \left(\text{pourcentage des } \mathbf{x}^i \text{ dans } \mathbb{D}\right) \cdot \mathbb{V}_{\mathbb{E}}. \tag{6.47}$$

Pour évaluer $< f >$, on peut utiliser (6.46), à condition de ne garder que les $\mathbf{x}^i$ dans $\mathbb{D}$ et de définir $N$ comme le nombre de ces $\mathbf{x}^i$.

#### 6.3.2.3 Comment choisir le nombre des échantillons ?

Quand $V_{\mathbb{D}}$ est connu, et pourvu que $f(\cdot)$ soit de carré intégrable sur $\mathbb{D}$, l'écart-type de $I$ comme évalué par la méthode de Monte-Carlo peut être estimé par

$$\sigma_I \approx V_{\mathbb{D}} \cdot \sqrt{\frac{< f^2 > - < f >^2}{N}}, \tag{6.48}$$

où

$$< f^2 > = \frac{1}{N} \sum_{i=1}^{N} f^2\left(\mathbf{x}^i\right). \tag{6.49}$$

La vitesse de convergence est donc en $\mathrm{O}\left(1/\sqrt{N}\right)$ *quelle que soit la dimension de* $\mathbf{x}$, ce qui est remarquable. Pour doubler la précision sur $I$, il faut cependant multiplier $N$ par quatre, et de nombreux échantillons peuvent être nécessaires pour atteindre une précision satisfaisante. Quand $n$ est grand, ceci dit, la situation serait bien pire si l'intégrande devait être évalué sur une grille régulière.

Des méthodes de *réduction de variance* peuvent être utilisées pour améliorer la précision avec laquelle $< f >$ est obtenue pour un $N$ donné [194].

#### 6.3.2.4 Intégration de Quasi-Monte-Carlo

Des réalisations en nombre fini de vecteurs aléatoires indépendants identiquement distribués avec une loi uniforme se révèlent ne pas être régulièrement distribués dans la région d'intérêt. Ceci suggère de les remplacer par des *suites de nombres quasi-aléatoires à faible discrépance* [164, 174], spécifiquement conçues pour l'éviter.

*Remarque 6.7.* L'utilisation d'une grille régulière est hors de question dans des espaces de grande dimension, pour au moins deux raisons : (*i*) le nombre des points nécessaires pour obtenir une grille régulière de pas donné est exponentiel en la dimension $n$ de $\mathbf{x}$ et (*ii*) il est impossible de modifier cette grille de façon incrémentale, car la seule option viable serait de diviser la taille du pas pour chaque dimension par un entier. □

## 6.4 Différentier des fonctions d'une variable

Il est délicat de différentier un signal bruité, car ceci amplifie les composantes du bruit dans les hautes fréquences. Nous supposons ici que le bruit peut être négligé, et nous intéressons à l'évaluation numérique d'une dérivée mathématique. Comme nous le fîmes pour l'intégration, nous supposons pour le moment que $f(\cdot)$ n'est connue qu'à travers son évaluation pour des valeurs numériquement connues de son argument, de sorte qu'une différentiation formelle ne peut être envisagée. (La section 6.6 montrera qu'une différentiation formelle du code évaluant une fonction est en fait possible, pourvu que le code source soit disponible.)

Nous nous limitons ici aux dérivées d'ordre un et deux, mais des approches similaires pourraient être employées pour évaluer des dérivées d'ordre plus élevé. L'hypothèse que l'effet du bruit peut être négligé devient d'autant plus cruciale que l'ordre de la dérivation augmente.

Soit $f(\cdot)$ une fonction dont la valeur numérique est connue en $x_0 < x_1 < \cdots < x_n$. Pour évaluer sa dérivée en $x \in [x_0, x_n]$, nous interpolons $f(\cdot)$ avec un polynôme $P_n(x)$ de degré $n$ puis évaluons analytiquement la dérivée de $P_n(x)$.

*Remarque 6.8.* Comme pour l'intégration en section 6.2, le problème à traiter est ainsi remplacé par un problème approché, qu'il est ensuite possible de résoudre de façon exacte (au moins d'un point de vue mathématique). □

### *6.4.1 Dérivées premières*

Puisque le polynôme interpolateur devra être dérivé une fois, il doit être d'ordre au moins égal à un. Il est trivial de vérifier que le polynôme d'ordre un interpolant les valeurs prises par $f(\cdot)$ en $x_0$ et $x_1$ est donné par

$$P_1(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0), \tag{6.50}$$

où $f(x_i)$ est à nouveau noté $f_i$. Ceci conduit à approximer $\dot{f}(x)$ pour $x \in [x_0, x_1]$ par

$$\dot{P}_1(x) = \frac{f_1 - f_0}{x_1 - x_0}. \tag{6.51}$$

Cette estimée de $\dot{f}(x)$ est donc la même pour tout $x$ dans $[x_0, x_1]$. Si $h = x_1 - x_0$, on peut l'exprimer comme la *différence avant*

$$\dot{f}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}, \tag{6.52}$$

ou comme la *différence arrière*

$$\dot{f}(x_1) \approx \frac{f(x_1) - f(x_1 - h)}{h}. \tag{6.53}$$

Le développement de Taylor au second ordre de $f(\cdot)$ autour de $x_0$ est donné par

$$f(x_0 + h) = f(x_0) + \dot{f}(x_0)h + \frac{\ddot{f}(x_0)}{2}h^2 + \mathrm{o}(h^2), \tag{6.54}$$

ce qui implique que

$$\frac{f(x_0 + h) - f(x_0)}{h} = \dot{f}(x_0) + \frac{\ddot{f}(x_0)}{2}h + \mathrm{o}(h). \tag{6.55}$$

On a donc

$$\dot{f}(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \mathrm{O}(h), \tag{6.56}$$

et l'erreur de méthode commise quand on utilise (6.52) est en $\mathrm{O}(h)$. C'est pourquoi (6.52) est une *différence avant du premier ordre*. De même,

$$\dot{f}(x_1) = \frac{f(x_1) - f(x_1 - h)}{h} + \mathrm{O}(h), \tag{6.57}$$

et (6.53) est une *différence arrière du premier ordre*.

Pour permettre une évaluation plus précise de $\dot{f}(\cdot)$, considérons maintenant un polynôme interpolateur du second ordre $P_2(x)$, interpolant les valeurs prises par $f(\cdot)$ en trois points régulièrement espacés $x_0, x_1$ et $x_2$, tels que

$$x_2 - x_1 = x_1 - x_0 = h. \tag{6.58}$$

La formule de Lagrange (5.14) se traduit par

$$\begin{aligned} P_2(x) &= \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}f_2 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}f_1 + \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}f_0 \\ &= \frac{1}{2h^2}[(x - x_0)(x - x_1)f_2 - 2(x - x_0)(x - x_2)f_1 + (x - x_1)(x - x_2)f_0]. \end{aligned}$$

Dérivons $P_2(x)$ une fois pour obtenir

$$\dot{P}_2(x) = \frac{1}{2h^2}[(x - x_1 + x - x_0)f_2 - 2(x - x_2 + x - x_0)f_1 + (x - x_2 + x - x_1)f_0], \tag{6.59}$$

tel que

$$\dot{P}_2(x_0) = \frac{-f(x_0 + 2h) + 4f(x_0 + h) - 3f(x_0)}{2h}, \tag{6.60}$$

$$\dot{P}_2(x_1) = \frac{f(x_1 + h) - f(x_1 - h)}{2h} \tag{6.61}$$

et

$$\dot{P}_2(x_2) = \frac{3f(x_2) - 4f(x_2 - h) + f(x_2 - 2h)}{2h}. \tag{6.62}$$

Comme

$$f(x_1+h) = f(x_1) + \dot{f}(x_1)h + \frac{\ddot{f}(x_1)}{2}h^2 + \mathrm{O}(h^3) \tag{6.63}$$

et

$$f(x_1-h) = f(x_1) - \dot{f}(x_1)h + \frac{\ddot{f}(x_1)}{2}h^2 + \mathrm{O}(h^3), \tag{6.64}$$

nous avons

$$f(x_1+h) - f(x_1-h) = 2\dot{f}(x_1)h + \mathrm{O}(h^3) \tag{6.65}$$

de sorte que

$$\dot{f}(x_1) = \dot{P}_2(x_1) + \mathrm{O}(h^2). \tag{6.66}$$

Approximer $\dot{f}(x_1)$ par $\dot{P}_2(x_1)$ est ainsi une *différence centrée d'ordre deux*. La même méthode permet de montrer que

$$\begin{aligned} \dot{f}(x_0) &= \dot{P}_2(x_0) + \mathrm{O}(h^2), & (6.67)\\ \dot{f}(x_2) &= \dot{P}_2(x_2) + \mathrm{O}(h^2). & (6.68) \end{aligned}$$

Approximer $\dot{f}(x_0)$ par $\dot{P}_2(x_0)$ revient ainsi à utiliser une *différence avant d'ordre deux*, tandis qu'approximer $\dot{f}(x_2)$ par $\dot{P}_2(x_2)$ revient à utiliser une *différence arrière d'ordre deux*.

Supposons $h$ suffisamment petit pour que les termes d'ordre plus élevé soient négligeables, mais suffisamment grand pour que les erreurs d'arrondi le soient aussi. Diviser $h$ par deux divisera alors approximativement l'erreur par deux pour une différence du premier ordre, et par quatre pour une différence du second ordre.

*Exemple 6.6.* Prenons $f(x) = x^4$, de sorte que $\dot{f}(x) = 4x^3$. Il vient

— pour la différence avant du premier ordre

$$\begin{aligned} \frac{f(x+h)-f(x)}{h} &= 4x^3 + 6hx^2 + 4h^2x + h^3 \\ &= \dot{f}(x) + \mathrm{O}(h), \end{aligned} \tag{6.69}$$

— pour la différence arrière du premier ordre

$$\begin{aligned} \frac{f(x)-f(x-h)}{h} &= 4x^3 - 6hx^2 + 4h^2x - h^3 \\ &= \dot{f}(x) + \mathrm{O}(h), \end{aligned} \tag{6.70}$$

— pour la différence centrée du second ordre

$$\begin{aligned} \frac{f(x+h)-f(x-h)}{2h} &= 4x^3 + 4h^2x \\ &= \dot{f}(x) + \mathrm{O}(h^2), \end{aligned} \tag{6.71}$$

— pour la différence avant du second ordre

$$\begin{aligned}\frac{-f(x+2h)+4f(x+h)-3f(x)}{2h} &= 4x^3-8h^2x-6h^3 \\ &= \dot{f}(x)+\mathrm{O}(h^2), \qquad (6.72)\end{aligned}$$

— et pour la différence arrière du second ordre

$$\begin{aligned}\frac{3f(x)-4f(x-h)+f(x-2h)}{2h} &= 4x^3-8h^2x+6h^3 \\ &= \dot{f}(x)+\mathrm{O}(h^2). \qquad (6.73)\end{aligned}$$

□

### *6.4.2 Dérivées secondes*

Puisque le polynôme interpolateur devra être dérivé deux fois pour évaluer une dérivée seconde, il doit être d'ordre au moins égal à deux. Considérons le polynôme d'ordre deux $P_2(x)$ qui interpole la fonction $f(x)$ en trois points régulièrement espacés $x_0, x_1$ et $x_2$ tels que (6.58) soit satisfaite, et dérivons (6.59) une fois pour obtenir

$$\ddot{P}_2(x) = \frac{f_2-2f_1+f_0}{h^2}. \qquad (6.74)$$

L'approximation de $\ddot{f}(x)$ est donc la même pour tout $x$ dans $[x_0, x_2]$. En version *différence centrée*, il vient

$$\ddot{f}(x_1) \approx \ddot{P}_2(x_1) = \frac{f(x_1+h)-2f(x_1)+f(x_1-h)}{h^2}. \qquad (6.75)$$

Comme

$$f(x_1+h) = \sum_{i=0}^{5} \frac{f^{(i)}(x_1)}{i!}h^i + \mathrm{O}(h^6), \qquad (6.76)$$

$$f(x_1-h) = \sum_{i=0}^{5} \frac{f^{(i)}(x_1)}{i!}(-h)^i + \mathrm{O}(h^6), \qquad (6.77)$$

les termes impairs disparaissent quand on somme (6.76) et (6.77). Ceci implique que

$$\frac{f(x_1+h)-2f(x_1)+f(x_1-h)}{h^2} = \frac{1}{h^2}\left[\ddot{f}(x_1)h^2 + \frac{f^{(4)}(x_1)}{12}h^4 + \mathrm{O}(h^6)\right], \qquad (6.78)$$

et que

$$\ddot{f}(x_1) = \frac{f(x_1+h)-2f(x_1)+f(x_1-h)}{h^2} + \mathrm{O}(h^2). \qquad (6.79)$$

On peut écrire de même des différences avant et arrière, et montrer que

$$\ddot{f}(x_0) = \frac{f(x_0+2h)-2f(x_0+h)+f(x_0)}{h^2}+\mathrm{O}(h), \tag{6.80}$$

$$\ddot{f}(x_2) = \frac{f(x_2)-2f(x_2-h)+f(x_2-2h)}{h^2}+\mathrm{O}(h). \tag{6.81}$$

*Remarque 6.9.* L'erreur de méthode de la différence centrée est donc en $\mathrm{O}(h^2)$, tandis que celles des différences avant et arrière sont seulement en $\mathrm{O}(h)$. C'est pourquoi le schéma de Crank-Nicolson pour la résolution de certaines équations aux dérivées partielles utilise une différence centrée (voir la section 13.3.3). □

*Exemple 6.7.* Reprenons $f(x)=x^4$, de sorte que $\ddot{f}(x)=12x^2$. Il vient

— pour la différence avant du premier ordre

$$\begin{aligned}\frac{f(x+2h)-2f(x+h)+f(x)}{h^2} &= 12x^2+24hx+14h^2\\ &= \ddot{f}(x)+\mathrm{O}(h),\end{aligned} \tag{6.82}$$

— pour la différence arrière du premier ordre

$$\begin{aligned}\frac{f(x)-2f(x-h)+f(x-2h)}{h^2} &= 12x^2-24hx+14h^2\\ &= \ddot{f}(x)+\mathrm{O}(h),\end{aligned} \tag{6.83}$$

— et pour la différence centrée du second ordre

$$\begin{aligned}\frac{f(x+h)-2f(x)+f(x-h)}{h^2} &= 12x^2+2h^2\\ &= \ddot{f}(x)+\mathrm{O}(h^2).\end{aligned} \tag{6.84}$$

□

### *6.4.3 Extrapolation de Richardson*

L'extrapolation de Richardson, présentée en section 5.3.4, s'applique aussi dans le contexte de la dérivation, comme illustré dans l'exemple qui suit.

*Exemple 6.8.* Approximons $r=\dot{f}(x)$ par la différence avant du premier ordre

$$R_1(h)=\frac{f(x+h)-f(x)}{h}, \tag{6.85}$$

telle que

$$\dot{f}(x)=R_1(h)+c_1h+\cdots \tag{6.86}$$

Pour $n = 1$, (5.54) se traduit par

$$\dot{f}(x) = 2R_1(\frac{h}{2}) - R_1(h) + \mathrm{O}(h^m), \tag{6.87}$$

avec $m > 1$. Posons $h' = h/2$ pour obtenir

$$2R_1(h') - R_1(2h') = \frac{-f(x+2h') + 4f(x+h') - 3f(x)}{2h'}, \tag{6.88}$$

qui est la différence avant du second ordre (6.60), de sorte que $m = 2$ et qu'un ordre d'approximation a été gagné. Notons que l'évaluation est ici via le membre de gauche de (6.88). □

L'extrapolation de Richardson peut bénéficier d'une simplification heureuse, comme dans l'exemple qui suit.

*Exemple 6.9.* Approximons maintenant $r = \dot{f}(x)$ par une différence centrée du second ordre

$$R_2(h) = \frac{f(x+h) - f(x-h)}{2h}, \tag{6.89}$$

de sorte que

$$\dot{f}(x) = R_2(h) + c_2 h^2 + \cdots \tag{6.90}$$

Pour $n = 2$, (5.54) se traduit par

$$\dot{f}(x) = \frac{1}{3}\left[4R_2(\frac{h}{2}) - R_2(h)\right] + \mathrm{O}(h^m), \tag{6.91}$$

avec $m > 2$. Posons à nouveau $h' = h/2$ pour obtenir

$$\frac{1}{3}[4R_2(h') - R_2(2h')] = \frac{N(x)}{12h'}, \tag{6.92}$$

avec

$$N(x) = -f(x+2h') + 8f(x+h') - 8f(x-h') + f(x-2h'). \tag{6.93}$$

Un développement de Taylor de $f(\cdot)$ autour de $x$ montre que les termes pairs dans le développement de $N(x)$ se simplifient et que

$$N(x) = 12\dot{f}(x)h' + 0 \cdot f^{(3)}(x)(h')^3 + \mathrm{O}(h'^5). \tag{6.94}$$

L'équation (6.92) implique donc que

$$\frac{1}{3}[4R_2(h') - R_2(2h')] = \dot{f}(x) + \mathrm{O}(h'^4). \tag{6.95}$$

L'extrapolation a permis ici de transformer une approximation du second ordre en approximation du quatrième ordre. □

## 6.5 Différentier des fonctions de plusieurs variables

Rappelons que

- le *gradient* d'une fonction différentiable $J(\cdot)$ de $\mathbb{R}^n$ vers $\mathbb{R}$ évalué en $\mathbf{x}$ est le vecteur colonne de dimension $n$ défini par (2.11),
- la *hessienne* d'une fonction deux fois différentiable $J(\cdot)$ de $\mathbb{R}^n$ vers $\mathbb{R}$ évaluée en $\mathbf{x}$ est la matrice $n \times n$ définie par (2.12),
- La *jacobienne* d'une fonction différentiable $\mathbf{f}(\cdot)$ de $\mathbb{R}^n$ vers $\mathbb{R}^p$ évaluée en $\mathbf{x}$ est la matrice $p \times n$ définie par (2.14),
- le *laplacien* d'une fonction $f(\cdot)$ de $\mathbb{R}^n$ vers $\mathbb{R}$ évalué en $\mathbf{x}$ est le scalaire défini par (2.19).

On rencontre fréquemment des gradients et des hessiennes en optimisation, tandis que les jacobiennes sont souvent impliquées dans la résolution de systèmes d'équations non linéaires et que les laplaciens interviennent dans des équations aux dérivées partielles. Dans tous ces exemples, les éléments à évaluer sont des dérivées *partielles*, ce qui veut dire qu'une seule des composantes de $\mathbf{x}$ est considérée comme variable, tandis que les autres sont maintenues constantes. Les techniques utilisées pour différentier des fonctions d'une seule variable peuvent donc être employées.

*Exemple 6.10.* Évaluons $\frac{\partial^2 f}{\partial x \partial y}$ pour $f(x,y) = x^3 y^3$. Nous approximons tout d'abord

$$g(x,y) = \frac{\partial f}{\partial y}(x,y) \tag{6.96}$$

par une différence centrée du second ordre

$$\begin{aligned}
\frac{\partial f}{\partial y}(x,y) &\approx x^3 \left[ \frac{(y+h_{\mathrm{y}})^3 - (y-h_{\mathrm{y}})^3}{2h_{\mathrm{y}}} \right], \\
x^3 \left[ \frac{(y+h_{\mathrm{y}})^3 - (y-h_{\mathrm{y}})^3}{2h_{\mathrm{y}}} \right] &= x^3 (3y^2 + h_{\mathrm{y}}^2) \\
&= \frac{\partial f}{\partial y}(x,y) + \mathrm{O}(h_{\mathrm{y}}^2).
\end{aligned} \tag{6.97}$$

Puis nous approximons $\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial g}{\partial x}(x,y)$ par une différence centrée du second ordre

$$\begin{aligned}
\frac{\partial^2 f}{\partial x \partial y} &\approx \left[ \frac{(x+h_{\mathrm{x}})^3 - (x-h_{\mathrm{x}})^3}{2h_{\mathrm{x}}} \right] (3y^2 + h_{\mathrm{y}}^2), \\
\left[ \frac{(x+h_{\mathrm{x}})^3 - (x-h_{\mathrm{x}})^3}{2h_{\mathrm{x}}} \right] (3y^2 + h_{\mathrm{y}}^2) &= (3x^2 + h_{\mathrm{x}}^2)(3y^2 + h_{\mathrm{y}}^2) \\
&= 9x^2y^2 + 3x^2 h_{\mathrm{y}}^2 + 3y^2 h_{\mathrm{x}}^2 + h_{\mathrm{x}}^2 h_{\mathrm{y}}^2 \\
&= \frac{\partial^2 f}{\partial x \partial y} + \mathrm{O}(h_{\mathrm{x}}^2) + \mathrm{O}(h_{\mathrm{y}}^2).
\end{aligned} \tag{6.98}$$

Globalement, il vient

$$\frac{\partial^2 f}{\partial x \partial y} \approx \left[\frac{(x+h_{\mathrm{x}})^3-(x-h_{\mathrm{x}})^3}{2h_{\mathrm{x}}}\right]\left[\frac{(y+h_{\mathrm{y}})^3-(y-h_{\mathrm{y}})^3}{2h_{\mathrm{y}}}\right]. \tag{6.99}$$

□

L'évaluation de gradients, au cœur de certaines des méthodes d'optimisation les plus efficaces, est considérée plus en détail dans la section qui suit, dans le cas particulier important où la fonction à différentier est évaluée par un code numérique.

## 6.6 Différentiation automatique

Supposons la valeur numérique de $f(\mathbf{x}_0)$ calculée par un code numérique, dont les variables d'entrée incluent les éléments de $\mathbf{x}_0$. Le premier problème considéré dans cette section est l'évaluation numérique du gradient de $f(\cdot)$ en $\mathbf{x}_0$, c'est à dire de

$$\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}_0) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}_0) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}_0) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}_0) \end{bmatrix}, \tag{6.100}$$

via l'utilisation d'un code numérique déduit de celui évaluant $f(\mathbf{x}_0)$.

Nous commençons par une description des problèmes rencontrés quand on utilise des différences finies, avant de décrire deux approches pour mettre en œuvre la différentiation automatique [71, 84, 222, 93, 190, 173, 85]. Ces approches permettent d'éviter toute erreur de méthode dans l'évaluation de gradients (ce qui n'élimine pas les erreurs d'arrondi, bien sûr). La première de ces approches peut conduire à une diminution drastique du volume de calcul, tandis que la seconde est simple à mettre en œuvre via la surcharge d'opérateurs.

### *6.6.1 Inconvénients de l'approximation par différences finies*

Remplaçons les dérivées partielles dans (6.100) par des différences finies, pour obtenir soit

$$\frac{\partial f}{\partial x_i}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0+\mathbf{e}^i\delta x_i)-f(\mathbf{x}_0)}{\delta x_i}, \quad i=1,\cdots,n, \tag{6.101}$$

où $\mathbf{e}^i$ est la $i$-ème colonne de $\mathbf{I}_n$, soit

$$\frac{\partial f}{\partial x_i}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0 + \mathbf{e}^i \delta x_i) - f(\mathbf{x}_0 - \mathbf{e}^i \delta x_i)}{2\delta x_i}, \quad i = 1, \cdots, n. \tag{6.102}$$

L'erreur de méthode est en $\mathrm{O}(\delta x_i^2)$ pour (6.102), au lieu de $\mathrm{O}(\delta x_i)$ pour (6.101). De plus, (6.102) n'introduit pas de distorsion de phase contrairement à (6.101) (penser au cas où $f(x)$ est une fonction trigonométrique). Par contre, (6.102) requiert plus de calculs que (6.101).

Rappelons qu'on ne peut pas faire tendre $\delta x_i$ vers zéro, car ceci entraînerait le calcul de la différence de nombres réels infinitésimalement proches, un désastre avec des nombres à virgule flottante. Il faut donc trouver un compromis entre les erreurs d'arrondi et de méthode en gardant des $\delta x_i$ finis (et pas nécessairement égaux). Un bon réglage de chacun des $\delta x_i$ est difficile, et peut nécessiter des essais et des erreurs. Même si l'on suppose que des $\delta x_i$ appropriés ont déjà été trouvés, une évaluation *approchée* du gradient de $f(\cdot)$ en $\mathbf{x}_0$ requiert $\dim \mathbf{x} + 1$ évaluations de $f(\cdot)$ avec (6.101) et $2 \cdot \dim \mathbf{x}$ évaluations de $f(\cdot)$ avec (6.102). Ceci peut s'avérer difficile si la dimension de $\mathbf{x}$ est très grande (comme en traitement d'images ou en optimisation de formes) ou si de nombreuses évaluations de gradient doivent être effectuées (comme en optimisation).

La différentiation automatique n'implique, quant à elle, aucune erreur de méthode et peut spectaculairement réduire la charge de calcul.

### 6.6.2 Idée de base de la différentiation automatique

La fonction $f(\cdot)$ est évaluée par un programme (le *code direct*). Nous supposons que $f(\mathbf{x})$ *telle qu'elle est mise en œuvre* dans le code direct est différentiable par rapport à $\mathbf{x}$. Le code direct ne peut donc pas contenir une instruction comme

```
if (x ≠ 1) then f(x):= x, else f(x):= 1.
```  (6.103)

Cette instruction n'a pas grand sens, mais des variantes plus difficiles à détecter peuvent se tapir dans le code direct. Nous distinguerons deux types de variables :

— les *variables indépendantes* (les entrées du code direct), qui incluent les composantes de $\mathbf{x}$,

— the *variables dépendantes* (à calculer par le code direct), qui incluent $f(\mathbf{x})$.

Toutes ces variables sont placées dans un *vecteur d'état* $\mathbf{v}$, une aide conceptuelle qui ne sera pas stockée en tant que telle dans l'ordinateur. Quand $\mathbf{x}$ prend la valeur numérique $\mathbf{x}_0$, l'une des variables dépendantes a pour valeur $f(\mathbf{x}_0)$ à la fin de l'exécution du code direct.

Pour simplifier, supposons tout d'abord que le code direct soit une suite linéaire de $N$ instructions d'affectation, sans boucle ou branchement conditionnel. La $k$-ème instruction d'affectation modifie la $\mu(k)$-ème composante de $\mathbf{v}$ selon

$$v_{\mu(k)} := \phi_k(\mathbf{v}). \tag{6.104}$$

En général, $\phi_k$ ne dépend que de quelques composantes de $\mathbf{v}$. Définissons $\mathbb{I}_k$ comme l'ensemble des indices de ces composantes et remplaçons (6.104) par une version plus détaillée

$$v_{\mu(k)} := \phi_k(\{v_i \mid i \in \mathbb{I}_k\}). \tag{6.105}$$

*Exemple 6.11.* Si la 5ème instruction d'affectation est

$$\mathtt{v}_4 := \mathtt{v}_1 + \mathtt{v}_2\mathtt{v}_3;$$

alors $\mu(5) = 4$, $\phi_5(\mathbf{v}) = v_1 + v_2 v_3$ et $\mathbb{I}_k = \{1,2,3\}$. □

Globalement, la $k$-ème instruction d'affectation se traduit par

$$\mathbf{v} := \boldsymbol{\Phi}_k(\mathbf{v}), \tag{6.106}$$

où $\boldsymbol{\Phi}_k$ ne change aucune des composantes de $\mathbf{v}$, sauf la $\mu(k)$-ème qui est modifiée selon (6.105).

*Remarque 6.10.* L'expression (6.106) ne doit pas être confondue avec une équation à résoudre par rapport à $\mathbf{v}$... □

Soit $\mathbf{v}_k$ l'état du code direct après l'exécution de la $k$-ème instruction d'affectation. Il satisfait

$$\mathbf{v}_k = \boldsymbol{\Phi}_k(\mathbf{v}_{k-1}), \qquad k = 1, \cdots, N. \tag{6.107}$$

C'est l'*équation d'état* d'un système dynamique à temps discret. Les équations d'état trouvent de nombreuses applications, en chimie, en mécanique, en automatique et en traitement du signal, par exemple. (Voir le chapitre 12 pour des exemples d'équations d'état à temps continu.) Le rôle du temps discret est tenu ici par le passage d'une instruction d'affectation à la suivante. L'état final $\mathbf{v}_N$ est obtenu à partir de l'état initial $\mathbf{v}_0$ par composition de fonctions, puisque

$$\mathbf{v}_N = \boldsymbol{\Phi}_N \circ \boldsymbol{\Phi}_{N-1} \circ \cdots \circ \boldsymbol{\Phi}_1(\mathbf{v}_0). \tag{6.108}$$

L'état initial $\mathbf{v}_0$ contient notamment la valeur $\mathbf{x}_0$ de $\mathbf{x}$, et l'état final $\mathbf{v}_N$ la valeur de $f(\mathbf{x}_0)$.

La règle de différentiation en chaîne appliquée à (6.107) et (6.108) se traduit par

$$\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}_0) = \frac{\partial \mathbf{v}_0^{\mathrm{T}}}{\partial \mathbf{x}} \cdot \frac{\partial \boldsymbol{\Phi}_1^{\mathrm{T}}}{\partial \mathbf{v}}(\mathbf{v}_0) \cdot \ldots \cdot \frac{\partial \boldsymbol{\Phi}_N^{\mathrm{T}}}{\partial \mathbf{v}}(\mathbf{v}_{N-1}) \cdot \frac{\partial f}{\partial \mathbf{v}_N}(\mathbf{x}_0). \tag{6.109}$$

Pour aider à mémoriser (6.109), remarquons que, comme $\boldsymbol{\Phi}_k(\mathbf{v}_{k-1}) = \mathbf{v}_k$, l'équation

$$\frac{\partial \mathbf{v}^{\mathrm{T}}}{\partial \mathbf{v}} = \mathbf{I} \tag{6.110}$$

permet de faire disparaître tous les termes intermédiaires du membre de droite de (6.109), ce qui laisse la même expression que dans le membre de gauche.

Posons

$$\frac{\partial \mathbf{v}_0^{\mathrm{T}}}{\partial \mathbf{x}} = \mathbf{C}, \tag{6.111}$$

$$\frac{\partial \boldsymbol{\Phi}_k^{\mathrm{T}}}{\partial \mathbf{v}}(\mathbf{v}_{k-1}) = \mathbf{A}_k \tag{6.112}$$

et

$$\frac{\partial f}{\partial \mathbf{v}_N}(\mathbf{x}_0) = \mathbf{b}. \tag{6.113}$$

L'équation (6.109) devient alors

$$\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x_0}) = \mathbf{C}\mathbf{A}_1 \cdots \mathbf{A}_N \mathbf{b}, \tag{6.114}$$

et l'évaluation du gradient de $f(\cdot)$ en $\mathbf{x}_0$ se résume au calcul de ce produit de matrices et de vecteur. Choisissons, de façon arbitraire, de stocker la valeur de $\mathbf{x}_0$ dans les $n$ premières composantes de $\mathbf{v}_0$, de sorte que

$$\mathbf{C} = \left[ \begin{array}{cc} \mathbf{I} & \mathbf{0} \end{array} \right]. \tag{6.115}$$

De façon tout aussi arbitraire, stockons la valeur de $f(\mathbf{x}_0)$ dans la dernière composante de $\mathbf{v}_N$, de sorte que

$$f(\mathbf{x}_0) = \mathbf{b}^{\mathrm{T}} \mathbf{v}_N, \tag{6.116}$$

avec

$$\mathbf{b} = \left[ \begin{array}{ccccc} 0 & \cdots & 0 & 1 \end{array} \right]^{\mathrm{T}}. \tag{6.117}$$

Reste à voir comment évaluer les matrices $\mathbf{A}_i$ et comment ordonner les calculs dans (6.114).

### *6.6.3 Évaluation rétrograde*

Évaluer (6.114) de façon rétrograde (de la droite vers la gauche) est particulièrement économique en flops, car chaque résultat intermédiaire est un vecteur de même dimension que $\mathbf{b}$ (c'est à dire $\dim \mathbf{v}$), tandis qu'une évaluation progressive (de la gauche vers la droite) produirait des résultats intermédiaires de mêmes dimensions que $\mathbf{C}$ (c'est à dire $\dim \mathbf{x} \times \dim \mathbf{v}$). Plus $\dim \mathbf{x}$ est grand, plus il devient économique de choisir l'évaluation rétrograde. Résoudre (6.114) de façon rétrograde implique de calculer la récurrence

$$\mathbf{d}_{k-1} = \mathbf{A}_k \mathbf{d}_k, \qquad k = N, \cdots, 1, \tag{6.118}$$

qui remonte le "temps", à partir de la condition terminale

$$\mathbf{d}_N = \mathbf{b}. \tag{6.119}$$

La valeur du gradient de $f(\cdot)$ en $\mathbf{x}_0$ est finalement donnée par

$$\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}_0) = \mathbf{C}\mathbf{d}_0, \tag{6.120}$$

ce qui revient à dire qu'elle est contenue dans les $\dim \mathbf{x}$ premiers éléments de $\mathbf{d}_0$. Le vecteur $\mathbf{d}_k$ a la même dimension que $\mathbf{v}_k$ et est appelé son *vecteur adjoint* (ou *vecteur dual*). La récurrence (6.118) est mise en œuvre dans un *code adjoint*, déduit du code direct par *dualisation*, comme expliqué ci-dessous. Voir la section 6.7.2 pour un exemple détaillé.

#### 6.6.3.1 Dualisation d'une instruction d'affectation

Examinons

$$\mathbf{A}_k = \frac{\partial \boldsymbol{\Phi}_k^{\mathrm{T}}}{\partial \mathbf{v}}(\mathbf{v}_{k-1}) \tag{6.121}$$

plus en détail. Rappelons que

$$[\boldsymbol{\Phi}_k(\mathbf{v}_{k-1})]_{\mu(k)} = \phi_k(\mathbf{v}_{k-1}), \tag{6.122}$$

et

$$[\boldsymbol{\Phi}_k(\mathbf{v}_{k-1})]_i = v_i(k-1), \quad \forall i \neq \mu(k), \tag{6.123}$$

où $v_i(k-1)$ est la $i$-ème composante de $\mathbf{v}_{k-1}$. Ceci a pour conséquence que $\mathbf{A}_k$ s'obtient en remplaçant la $\mu(k)$-ème colonne de la matrice identité $\mathbf{I}_{\dim \mathbf{v}}$ par le vecteur $\frac{\partial \phi_k}{\partial \mathbf{v}}(\mathbf{v}_{k-1})$ pour obtenir

$$\mathbf{A}_k = \begin{bmatrix} 1 & 0 & \cdots & \frac{\partial \phi_k}{\partial v_1}(\mathbf{v}_{k-1}) & 0 \\ 0 & \ddots & 0 & \vdots & \vdots \\ \vdots & 0 & 1 & \vdots & \vdots \\ \vdots & \vdots & 0 & \frac{\partial \phi_k}{\partial v_{\mu(k)}}(\mathbf{v}_{k-1}) & 0 \\ 0 & 0 & 0 & \vdots & 1 \end{bmatrix}. \tag{6.124}$$

La structure de $\mathbf{A}_k$ révélée par (6.124) a des conséquences directes sur les instructions d'affectation à inclure dans le code adjoint pour mettre en œuvre (6.118).

La $\mu(k)$-ème composante de la diagonale principale de $\mathbf{A}_k$ est la seule pour laquelle le un de la matrice identité a disparu, ce qui explique pourquoi la $\mu(k)$-ème composante de $\mathbf{d}_{k-1}$ requiert un traitement spécial. Soit $d_i(k-1)$ la $i$-ème composante de $\mathbf{d}_{k-1}$. À cause de (6.124), la récurrence (6.118) équivaut à

$$d_i(k-1) = d_i(k) + \frac{\partial \phi_k}{\partial v_i}(\mathbf{v}_{k-1}) d_{\mu(k)}(k), \quad \forall i \neq \mu(k), \tag{6.125}$$

$$d_{\mu(k)}(k-1) = \frac{\partial \phi_k}{\partial v_{\mu(k)}}(\mathbf{v}_{k-1}) d_{\mu(k)}(k). \tag{6.126}$$

Pour calculer $\mathbf{d}_0$, il n'est pas nécessaire de stocker les valeurs successives prises par le vecteur dual $\mathbf{d}$, et l'indexation de $\mathbf{d}$ par le "temps" peut donc être évitée. Les (pseudo) instructions adjointes pour

$$\mathtt{v}_{\mu(k)} := \phi_k(\{\mathtt{v}_i \mid i \in \mathbb{I}_k\});$$

seront alors, *dans cet ordre*

$$\texttt{for all } i \in \mathbb{I}_k, \quad i \neq \mu(k), \texttt{ do } \quad \mathtt{d}_i := \mathtt{d}_i + \frac{\partial \phi_k}{\partial v_i}(\mathbf{v}_{k-1})\mathtt{d}_{\mu(k)};$$
$$\mathtt{d}_{\mu(k)} := \frac{\partial \phi_k}{\partial v_{\mu(k)}}(\mathbf{v}_{k-1})\mathtt{d}_{\mu(k)};$$

*Remarque 6.11.* Si $\phi_k$ dépend *non linéairement* de certaines variables du code direct, alors le code adjoint fera intervenir les valeurs prises par ces variables, qui devront donc être stockées lors de l'exécution du code direct avant l'exécution du code adjoint. Ces exigences de stockage sont une limitation de l'évaluation rétrograde. □

*Exemple 6.12.* Supposons que le code direct contienne l'instruction d'affectation

`cost := cost+(y-ym)^2;`

de sorte que $\phi_k$ = `cost+(y-ym)^2`.

Soient `dcost`, `dy` et $\mathtt{dy}_\mathrm{m}$ les variables duales de `cost`, `y` et $\mathtt{y}_\mathrm{m}$. La dualisation de cette instruction produit les (pseudo) instructions suivantes pour le code adjoint

$$\mathtt{dy} := \mathtt{dy} + \frac{\partial \phi_k}{\partial y}\ \texttt{dcost} = \mathtt{dy} + 2\,\texttt{(y-ym) dcost;}$$
$$\mathtt{dy}_\mathrm{m} := \mathtt{dy}_\mathrm{m} + \frac{\partial \phi_k}{\partial y_\mathrm{m}}\ \texttt{dcost} = \mathtt{dy}_\mathrm{m} - 2\,\texttt{(y - ym) dcost;}$$
$$\texttt{dcost} := \frac{\partial \phi_k}{\partial \mathrm{cost}}\ \texttt{dcost} = \texttt{dcost;}\quad \texttt{\% inutile}$$

Une seule instruction du code direct s'est donc traduite par plusieurs instructions du code adjoint. □

#### 6.6.3.2 Ordre de dualisation

Rappelons que le rôle du temps est tenu par le passage d'une instruction d'affectation à la suivante. Puisque le code adjoint est exécuté en temps rétrograde, les groupes d'instructions duales associés à chacune des instructions d'affectation du code direct seront exécutés *dans l'ordre inverse* de l'*exécution* des instructions d'affectation correspondantes du code direct.

Quand le code direct comporte des boucles, inverser le sens du temps revient à inverser le sens de variation de leurs compteurs d'itérations ainsi que l'ordre des

instructions dans chacune des boucles. En ce qui concerne les branchements conditionnels, si le code direct contient

```
if (C) then (code A) else (code B);
```

alors le code adjoint doit contenir

```
if (C) then (adjoint de A) else (adjoint de B);
```

et la valeur *vraie* ou *fausse* prise par la condition `C` pendant l'exécution du code direct doit être mémorisée pour que le code adjoint sache quelle branche suivre.

#### 6.6.3.3 Initialisation du code adjoint

La condition terminale (6.119) avec **b** donné par (6.117) signifie que toutes les variables duales doivent être initialisées à zéro, sauf celle associée à la valeur de $f(\mathbf{x}_0)$, qui doit être initialisée à un.

*Remarque 6.12.* **v**, **d** et $\mathbf{A}_k$ ne sont pas mémorisés en tant que tels. Seules les variables directes et duales interviennent. On améliore la lisibilité du code adjoint en utilisant une convention systématique pour nommer les variables duales, par exemple en ajoutant un `d` en tête du nom de la variable dualisée comme dans l'exemple 6.12. □

#### 6.6.3.4 En résumé

La procédure de différentiation automatique via l'usage d'un code adjoint est résumée par la figure 6.2.

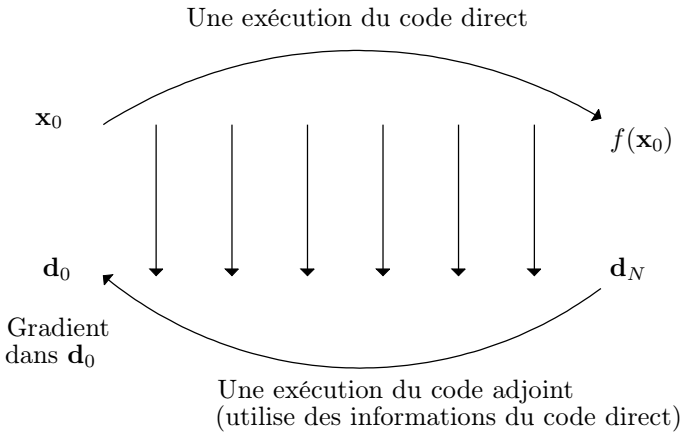


**Fig. 6.2** Procédure à base de code adjoint pour le calcul de gradients

La méthode à base de code adjoint évite les erreurs de méthode dues aux approximations par différences finies. La génération du code adjoint à partir du source du code direct est systématique et peut être automatisée.

Le volume de calcul requis pour évaluer la fonction $f(\cdot)$ et son gradient est typiquement de l'ordre de trois fois celui requis par la seule évaluation de la fonction *quelle que soit la dimension de* $\mathbf{x}$ (à comparer avec l'approche à base de différences finies, pour laquelle l'évaluation de $f(\cdot)$ doit être répétée plus de $\dim \mathbf{x}$ fois). La méthode à base de code adjoint est donc particulièrement appropriée quand

— $\dim \mathbf{x}$ est très grand, comme dans certains problèmes de traitement d'images ou d'optimisation de formes,
— de nombreuses évaluations de gradients sont nécessaires, comme c'est souvent le cas en optimisation,
— l'évaluation de $f(\cdot)$ est longue ou coûteuse.

Par contre, cette méthode ne peut être appliquée que si le source du code direct est disponible et différentiable. Une mise en œuvre à la main demande du soin, car une seule erreur de codage peut rendre le résultat invalide. (Il existe des techniques de vérification partielle, qui exploitent le fait que le produit scalaire du vecteur dual avec la solution des équations d'état linéarisées doit rester constant le long de la trajectoire de l'état.) Finalement, l'exécution du code adjoint requiert la connaissance des valeurs prises par certaines variables lors de l'exécution du code direct (les variables qui interviennent de façon non linéaire dans des instructions d'affectation du code direct). Il faut donc stocker ces valeurs, ce qui peut poser des problèmes de taille de mémoire.

### *6.6.4 Évaluation progressive*

L'évaluation progressive peut être interprétée comme l'évaluation de (6.114) de gauche à droite.

#### 6.6.4.1 Méthode

Soit $\mathbb{P}$ l'ensemble des paires ordonnées $V$ formées d'une variable réelle $v$ et de son gradient par rapport au vecteur $\mathbf{x}$ des variables indépendantes

$$V = (v, \frac{\partial v}{\partial \mathbf{x}}). \tag{6.127}$$

Si $A$ et $B$ appartiennent à $\mathbb{P}$, alors

$$A + B = \left( a + b, \frac{\partial a}{\partial \mathbf{x}} + \frac{\partial b}{\partial \mathbf{x}} \right), \tag{6.128}$$

$$A-B=\left(a-b,\frac{\partial a}{\partial \mathbf{x}}-\frac{\partial b}{\partial \mathbf{x}}\right), \tag{6.129}$$

$$A\cdot B=\left(a\cdot b,\frac{\partial a}{\partial \mathbf{x}}\cdot b+a\cdot\frac{\partial b}{\partial \mathbf{x}}\right), \tag{6.130}$$

$$\frac{A}{B}=\left(\frac{a}{b},\frac{\frac{\partial a}{\partial \mathbf{x}}\cdot b-a\cdot\frac{\partial b}{\partial \mathbf{x}}}{b^2}\right). \tag{6.131}$$

Pour la dernière expression, il est plus efficace d'écrire

$$\frac{A}{B}=\left(c,\frac{\frac{\partial a}{\partial \mathbf{x}}-c\frac{\partial b}{\partial \mathbf{x}}}{b}\right), \tag{6.132}$$

avec $c=a/b$.

La paire ordonnée associée à une constante réelle $d$ est $D=(d,\mathbf{0})$, et celle associée à la $i$-ème variable indépendante $x_i$ est $X_i=(x_i,\mathbf{e}^i)$, où $\mathbf{e}^i$ est comme d'habitude la $i$-ème colonne de la matrice identité. La valeur $g(\mathbf{v})$ prise par une fonction élémentaire $g(\cdot)$ intervenant dans une instruction du code direct est remplacée par la paire

$$G(\mathbf{V})=\left(g(\mathbf{v}),\frac{\partial \mathbf{v}^{\mathrm{T}}}{\partial \mathbf{x}}\cdot\frac{\partial g}{\partial \mathbf{v}}(\mathbf{v})\right), \tag{6.133}$$

où $\mathbf{V}$ est un vecteur de paires $V_i=(v_i,\frac{\partial v_i}{\partial \mathbf{x}})$, qui contient tous les éléments de $\partial\mathbf{v}^{\mathrm{T}}/\partial\mathbf{x}$ et où $\partial g/\partial\mathbf{v}$ est facile à calculer analytiquement.

*Exemple 6.13.* Considérons le code direct de l'exemple qui sera traité en section 6.7.2. Il suffit d'exécuter ce code direct en remplaçant chaque opération sur les réels par l'opération correspondante sur les paires ordonnées, après avoir initialisé les paires comme suit :

$$F=(0,\mathbf{0}), \tag{6.134}$$

$$Y(k)=(y(k),\mathbf{0}),\qquad k=1,\cdots,n_{\mathrm{t}}. \tag{6.135}$$

$$P_1=\left(p_1,\begin{bmatrix}1\\0\end{bmatrix}\right), \tag{6.136}$$

$$P_2=\left(p_2,\begin{bmatrix}0\\1\end{bmatrix}\right). \tag{6.137}$$

Après exécution du code direct, on obtient

$$F=\left(f(\mathbf{x}_0),\frac{\partial f}{\partial \mathbf{x}}(\mathbf{x}_0)\right), \tag{6.138}$$

où $\mathbf{x}_0=(p_1,p_2)^{\mathrm{T}}$ est le vecteur qui contient les valeurs numériques des paramètres pour lesquelles le gradient doit être évalué. □

#### 6.6.4.2 Comparaison avec l'évaluation rétrograde

Contrairement à la méthode à base de code adjoint, la différentiation progressive utilise le même code pour évaluer la fonction et son gradient. Sa mise en œuvre est rendue beaucoup plus simple par l'exploitation de la *surcharge d'opérateurs* qu'autorisent des langages comme C++, ADA, FORTRAN 90 ou MATLAB. Cette surcharge permet de changer le sens donné à des opérateurs en fonction du type des objets sur lesquels ils opèrent. Pourvu que les opérations sur les paires de $\mathbb{P}$ aient été définies, il devient ainsi possible d'utiliser le code direct sans autre modification que de déclarer que les variables appartiennent au type "paire". Le calcul s'adapte alors automatiquement.

Un autre avantage de cette approche est qu'elle fournit le gradient de chacune des variables du code. Ceci veut dire, par exemple, que la sensibilité au premier ordre de la sortie du modèle par rapport aux paramètres

$$\mathbf{s}(k,\mathbf{x}) = \frac{\partial y_{\mathrm{m}}(k,\mathbf{x})}{\partial \mathbf{x}}, \qquad k = 1,\cdots,n_{\mathrm{t}}, \tag{6.139}$$

est facilement disponible, ce qui permet d'utiliser cette information dans une méthode de Gauss-Newton (voir la section 9.3.4.3).

Par contre, le nombre de flops est plus élevé qu'avec l'évaluation rétrograde, et ce d'autant plus que la dimension de $\mathbf{x}$ est grande.

### *6.6.5 Extension à l'évaluation de hessiennes*

Si $f(\cdot)$ est deux fois différentiable par rapport à $\mathbf{x}$, on peut souhaiter évaluer sa hessienne

$$\frac{\partial^2 f}{\partial \mathbf{x}\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\mathbf{x}) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_2^2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}) & \frac{\partial^2 f}{\partial x_n \partial x_2}(\mathbf{x}) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(\mathbf{x}) \end{bmatrix}, \tag{6.140}$$

et la différentiation automatique s'étend facilement à ce cas.

#### 6.6.5.1 Évaluation rétrograde

La hessienne est relié au gradient par

$$\frac{\partial^2 f}{\partial \mathbf{x}\partial \mathbf{x}^{\mathrm{T}}} = \frac{\partial}{\partial \mathbf{x}}\left(\frac{\partial f}{\partial \mathbf{x}^{\mathrm{T}}}\right). \tag{6.141}$$

Si $\mathbf{g}(\mathbf{x})$ est le gradient de $f(\cdot)$ en $\mathbf{x}$, alors

$$\frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \frac{\partial \mathbf{g}^{\mathrm{T}}}{\partial \mathbf{x}}(\mathbf{x}). \tag{6.142}$$

La section 6.6.3 a montré que $\mathbf{g}(\mathbf{x})$ peut être évalué très efficacement en combinant l'emploi d'un code direct évaluant $f(\mathbf{x})$ et du code adjoint correspondant. Cette combinaison peut elle-même être vue comme un second code direct évaluant $\mathbf{g}(\mathbf{x})$. Supposons que la valeur de $\mathbf{g}(\mathbf{x})$ soit dans les $n$ derniers éléments du vecteur d'état $\mathbf{v}$ de ce second code direct à la fin de son exécution. Un second code adjoint peut maintenant être associé à ce second code direct pour calculer la hessienne. Il utilisera une variante de (6.109), où la sortie du second code direct est le vecteur $\mathbf{g}(\mathbf{x})$ au lieu du scalaire $f(\mathbf{x})$ :

$$\frac{\partial \mathbf{g}^{\mathrm{T}}}{\partial \mathbf{x}}(\mathbf{x}) = \frac{\partial \mathbf{v}_0^{\mathrm{T}}}{\partial \mathbf{x}} \cdot \frac{\partial \boldsymbol{\Phi}_1^{\mathrm{T}}}{\partial \mathbf{v}}(\mathbf{v}_0) \cdot \ldots \cdot \frac{\partial \boldsymbol{\Phi}_N^{\mathrm{T}}}{\partial \mathbf{v}}(\mathbf{v}_{N-1}) \cdot \frac{\partial \mathbf{g}^{\mathrm{T}}}{\partial \mathbf{v}_N}(\mathbf{x}). \tag{6.143}$$

Il suffit de remplaçer (6.113) et (6.117) par

$$\frac{\partial \mathbf{g}^{\mathrm{T}}}{\partial \mathbf{v}_N}(\mathbf{x}) = \mathbf{B} = \begin{bmatrix} \mathbf{0} \\ \mathbf{I}_n \end{bmatrix}, \tag{6.144}$$

et (6.114) par

$$\frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \mathbf{C}\mathbf{A}_1 \cdots \mathbf{A}_N \mathbf{B}, \tag{6.145}$$

pour que le calcul de la hessienne se résume à l'évaluation du produit de ces matrices. Tout le reste demeure formellement inchangé, mais la charge de calcul augmente, puisque le vecteur $\mathbf{b}$ a été remplacé par une matrice $\mathbf{B}$ à $n$ colonnes.

#### 6.6.5.2 Évaluation progressive

Au moins dans son principe, l'extension de la différentiation progressive à l'évaluation de dérivées secondes est là aussi plus simple que pour la méthode à base de code adjoint, puisqu'il suffit de remplacer le calcul sur des paires ordonnées par un calcul sur des triplets ordonnés

$$V = (v, \frac{\partial v}{\partial \mathbf{x}}, \frac{\partial^2 v}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}). \tag{6.146}$$

On peut exploiter le fait que les hessiennes sont symétriques.

## 6.7 Exemples MATLAB

### *6.7.1 Intégration*

La fonction densité de probabilité d'une variable gaussienne $x$ de moyenne $\mu$ et d'écart-type $\sigma$ est

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]. \tag{6.147}$$

La probabilité que $x$ appartienne à l'intervalle $[\mu - 2\sigma, \mu + 2\sigma]$ est donnée par

$$I = \int_{\mu-2\sigma}^{\mu+2\sigma} f(x)\mathrm{d}x. \tag{6.148}$$

Elle ne dépend pas des valeurs prises par $\mu$ et $\sigma$, et vaut

$$\mathrm{erf}(\sqrt{2}) \approx 0.9544997361036416. \tag{6.149}$$

Évaluons la par intégration numérique pour $\mu = 0$ et $\sigma = 1$. Nous devons donc calculer

$$I = \int_{-2}^{2} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)\mathrm{d}x. \tag{6.150}$$

Une des fonctions disponibles à cet effet est `quad` [67], qui combine des idées de la méthode 1/3 de Simpson et de la méthode de Romberg, et coupe en deux récursivement l'intervalle d'intégration où et quand c'est utile pour maintenir une estimée de l'erreur de méthode en dessous d'une tolérance absolue, fixée par défaut à $10^{-6}$. Le script

```
f = @(x) exp(-x.^2/2)/sqrt(2*pi);
Integral = quad(f,-2,2)
```

produit

```
Integral = 9.544997948576686e-01
```

de sorte que l'erreur absolue est bien inférieure à $10^{-6}$. Notons le point dans la définition de la fonction anonyme `f`, requis car `x` est considéré comme un argument vectoriel. Voir la documentation MATLAB pour plus de détails.

$I$ peut aussi être évalué avec une méthode de Monte-Carlo, comme dans le script

```
f = @(x) exp(-x.^2/2)/sqrt(2*pi);
IntMC = zeros(20,1);
N=1;
for i=1:20,
    X = 4*rand(N,1)-2;
    % X est uniforme entre -2 et 2
```

```
    % la longueur de [-2,2] est de 4
    F = f(X);
    IntMC(i) = 4*mean(F)
    N = 2*N;
    % le nombre d'évaluations de la fonction
    % double à chaque itération
end
ErrorOnInt = IntMC - 0.9545;
plot(ErrorOnInt,'o','MarkerEdgeColor',...
    'k','MarkerSize',7)
xlabel('log_2(N)')
ylabel('Absolute error on I')
```

Cette approche est incapable de concurrencer `quad`, et la figure 6.3 confirme que la convergence vers zéro de l'erreur absolue sur l'intégrale est lente.

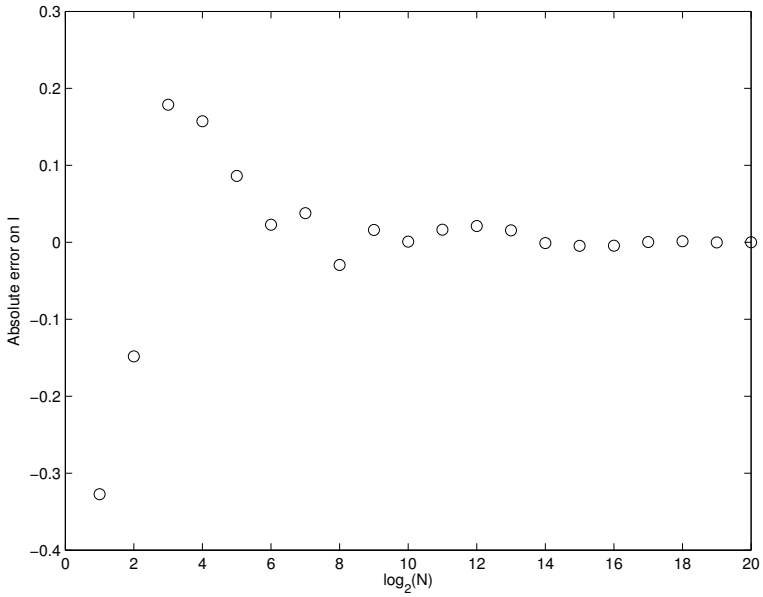


**Fig. 6.3** Erreur absolue sur $I$ en fonction du logarithme du nombre $N$ des évaluations de l'intégrande

Ce qui sauve l'approche de Monte-Carlo, c'est sa capacité à évaluer des intégrales de dimension plus élevée. Illustrons ceci en évaluant

$$V_n = \int_{\mathbb{B}^n} \mathrm{d}\mathbf{x}, \tag{6.151}$$

où $\mathbb{B}^n$ est la boule euclidienne unitaire dans $\mathbb{R}^n$,

$$\mathbb{B}^n = \{\mathbf{x} \in \mathbb{R}^n \text{ tels que } \|\mathbf{x}\|_2 \leqslant 1\}. \tag{6.152}$$

Ceci peut être mené à bien avec le script qui suit, où `n` est la dimension de l'espace euclidien et `V(i)` est la valeur de $V_n$ comme estimée à partir de $2^i$ valeurs de $\mathbf{x}$ tirées de façon pseudo-aléatoire dans $[-1,1]^{\times n}$.

```
clear all
V = zeros(20,1);
N = 1;
%%%
for i=1:20,
    F = zeros(N,1);
    X = 2*rand(n,N)-1;
    % X uniforme entre -1 et 1
    for j=1:N,
        x = X(:,j);
        if (norm(x,2)<=1)
            F(j) = 1;
        end
    end
    V(i) = mean(F)*2^n;
    N = 2*N;
    % le nombre d'évaluations de la fonction
    % double à chaque itération
end
```

$V_n$ est l'(hyper) volume de $\mathbb{B}^n$, qui peut être calculé exactement. La récurrence

$$V_n = \frac{2\pi}{n} V_{n-2} \tag{6.153}$$

peut par exemple être utilisée pour le calculer quand $n$ est pair, en partant de $V_2 = \pi$. Elle implique que $V_6 = \pi^3/6$. En exécutant le script de Monte-Carlo qui précède avec `n = 6;` et en y ajoutant

```
TrueV6 = (pi^3)/6;
RelErrOnV6 = 100*(V - TrueV6)/TrueV6;
plot(RelErrOnV6,'o','MarkerEdgeColor',...
    'k','MarkerSize',7)
xlabel('log_2(N)')
ylabel('Relative error on V_6 (in %)')
```

nous obtenons la figure 6.4, qui montre l'évolution de l'erreur relative sur $V_6$ en fonction de $\log_2 N$.

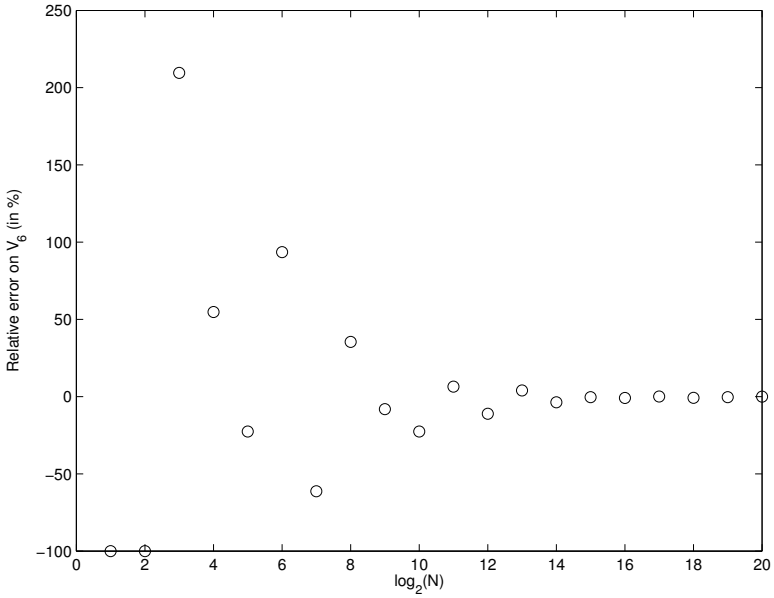


**Fig. 6.4** Erreur relative sur le volume d'une boule euclidienne en dimension six en fonction du logarithme du nombre $N$ d'évaluations de l'intégrande

### *6.7.2 Différentiation*

Considérons le modèle à plusieurs exponentielles

$$y_{\mathrm{m}}(k,\mathbf{p}) = \sum_{i=1}^{n_{\mathrm{exp}}} p_i \cdot \exp(p_{n_{\mathrm{exp}}+i} \cdot t_k), \tag{6.154}$$

où les éléments de $\mathbf{p}$ sont les paramètres inconnus $p_i$, $i = 1,\cdots,2n_{\mathrm{exp}}$, à estimer à partir des données

$$[y(k),t(k)], \quad k = 1,\cdots,n_{\mathrm{times}} \tag{6.155}$$

en minimisant

$$J(\mathbf{p}) = \sum_{k=1}^{n_{\mathrm{times}}} [y(k) - y_{\mathrm{m}}(k,\mathbf{p})]^2. \tag{6.156}$$

Un script évaluant le coût $J(\mathbf{p})$ (code direct) est

```
cost = 0;
for k=1:ntimes,      % boucle progressive
    ym(k) = 0;
    for i=1:nexp,    % boucle progressive
```

```
        ym(k) = ym(k)+p(i)*exp(p(nexp+i)*t(k));
    end
    cost = cost+(y(k)-ym(k))^2;
end
```

L'utilisation des règles systématiques décrites dans la section 6.6.2 permet d'en déduire le script suivant (code adjoint),

```
dcost=1;
dy=zeros(ntimes,1);
dym=zeros(ntimes,1);
dp=zeros(2*nexp,1);
dt=zeros(ntimes,1);
for k=ntimes:-1:1,          % boucle rétrograde
    dy(k) = dy(k)+2*(y(k)-ym(k))*dcost;
    dym(k) = dym(k)-2*(y(k)-ym(k))*dcost;
    dcost = dcost;
    for i=nexp:-1:1,        % boucle rétrograde
        dp(i) = dp(i)+exp(p(nexp+i)*t(k))*dym(k);
        dp(nexp+i) = dp(nexp+i)...
            +p(i)*t(k)*exp(p(nexp+i)*t(k))*dym(k);
        dt(k) = dt(k)+p(i)*p(nexp+i)...
            *exp(p(nexp+i)*t(k))*dym(k);
        dym(k) = dym(k);
    end
    dym(k) = 0;
end
dcost=0;
dp % contient le gradient
```

*Remarque 6.13.* On pourrait bien sûr rendre ce code plus concis en éliminant des instructions inutiles. On pourrait également l'écrire de façon à minimiser le nombre des opérations sur les éléments de vecteurs, qui sont inefficaces dans un langage orienté vers les matrices. □

Supposons les données générées par le script

```
ntimes = 100; % nombre de temps de mesure
nexp = 2;     % nombre de termes exponentiels
% valeur de p utilisée pour générer les données :
pstar = [1; -1; -.3; -1];
h = 0.2; % taille du pas de temps
t(1) = 0;
for k=2:ntimes,
    t(k)=t(k-1)+h;
```

```
end
for k=1:ntimes,
    y(k) = 0;
    for i=1:nexp,
        y(k) = y(k)+pstar(i)*exp(pstar(nexp+i)*t(k));
    end
end
```

Avec ces données, la valeur du gradient en $\mathbf{p} = (1.1, -0.9, -0.2, -0.9)^{\mathrm{T}}$ calculée par le code adjoint est

```
 dp =
     7.847859612874749e+00
     2.139461455801426e+00
     3.086120784615719e+01
    -1.918927727244027e+00
```

Dans cet exemple simple, il est facile de calculer la valeur du gradient du coût analytiquement puisque

$$\frac{\partial J}{\partial \mathbf{p}} = -2 \sum_{k=1}^{n_{\text{times}}} [y(k) - y_{\text{m}}(k, \mathbf{p})] \cdot \frac{\partial y_{\text{m}}}{\partial \mathbf{p}}(k), \tag{6.157}$$

avec, pour $i = 1, \cdots, n_{\text{exp}}$,

$$\frac{\partial y_{\text{m}}}{\partial p_i}(k, \mathbf{p}) = \exp(p_{n_{\text{exp}}+i} \cdot t_k), \tag{6.158}$$

$$\frac{\partial y_{\text{m}}}{\partial p_{n_{\text{exp}}+i}}(k, \mathbf{p}) = t_k \cdot p_i \cdot \exp(p_{n_{\text{exp}}+i} \cdot t_k). \tag{6.159}$$

Les résultats du code adjoint peuvent donc être vérifiés en exécutant le script

```
for i=1:nexp,
    for k=1:ntimes,
        s(i,k) = exp(p(nexp+i)*t(k));
        s(nexp+i,k) = t(k)*p(i)*exp(p(nexp+i)*t(k));
    end
end
for i=1:2*nexp,
    g(i) = 0;
    for k=1:ntimes,
        g(i) = g(i)-2*(y(k)-ym(k))*s(i,k);
    end
end
g % contient le gradient
```

Pour les mêmes données et la même valeur de $\mathbf{p}$, nous obtenons

```
g =
     7.847859612874746e+00
     2.139461455801424e+00
     3.086120784615717e+01
    -1.918927727244027e+00
```

qui correspond bien aux résultats du code adjoint.

## 6.8 En résumé

— Les méthodes traditionnelles pour évaluer des intégrales définies, comme celles de Simpson et de Boole, requièrent que les points où l'intégrande est évalué soient régulièrement espacés. Pour cette raison, elles perdent des degrés de liberté et leurs erreurs de méthode sont plus grandes que ce qu'elles auraient pu être.

— La méthode de Romberg applique le principe de Richardson à la méthode des trapèzes, et peut rapidement produire des résultats extrêmement précis grâce à des simplifications heureuses si l'intégrande est suffisamment lisse.

— La quadrature gaussienne échappe à la contrainte d'un espacement régulier des points d'évaluation, ce qui permet d'améliorer les performances, mais elle se tient aussi à des règles fixées pour décider où évaluer l'intégrande.

— Pour toutes ces méthodes, une approche *diviser pour régner* peut être utilisée pour couper l'horizon d'intégration en sous-intervalles de façon à s'adapter à des changements dans la vitesse de variation de l'intégrande.

— La transformation de l'intégration d'une fonction en celle d'une équation différentielle ordinaire permet aussi d'adapter la taille du pas au comportement local de l'intégrande.

— L'évaluation d'intégrales définies pour des fonctions de plusieurs variables est beaucoup plus compliquée que dans le cas à une seule variable. Pour les problèmes de petite dimension, et pourvu que l'intégrande soit suffisamment lisse, on peut utiliser des intégrations à une dimension imbriquées. L'approche de Monte-Carlo est plus simple à mettre en œuvre (si l'on dispose d'un bon générateur de nombres aléatoires) et peut supporter des discontinuités de l'intégrande. Pour diviser l'écart-type sur l'erreur par deux, il faut alors multiplier le nombre des évaluations de l'intégrande par quatre. Ceci est vrai quelle que soit la dimension de $\mathbf{x}$, ce qui rend l'intégration de Monte-Carlo particulièrement appropriée pour les problèmes en grande dimension.

— La différentiation numérique s'appuie typiquement sur l'interpolation polynomiale. L'ordre de l'approximation peut être calculé et utilisé dans une extrapolation de Richardson pour augmenter l'ordre de l'erreur de méthode. Ceci peut permettre d'éviter les pas trop petits qui conduisent à une explosion de l'erreur d'arrondi.

— Comme les éléments des gradients, des hessiennes et des jacobiennes sont des dérivées partielles, ils peuvent être évalués en utilisant les techniques disponibles pour les fonctions d'une seule variable.
— La différentiation automatique permet d'évaluer le gradient d'une fonction définie par un programme. Contrairement à l'approche par différences finies, elle n'induit pas d'erreur de méthode. La différentiation rétrograde requiert moins de flops que la différentiation progressive, tout particulièrement quand $\dim \mathbf{x}$ est grand, mais elle est plus compliquée à mettre en œuvre et peut demander beaucoup de mémoire. Ces deux techniques s'étendent à l'évaluation numérique de dérivées d'ordres plus élevés.

# Chapitre 7
# Résoudre des systèmes d'équations non linéaires

## 7.1 Quelles différences avec le cas linéaire ?

Comme au chapitre 3, le nombre d'équations scalaires est supposé ici égal au nombre d'inconnues scalaires. Rappelons que dans le cas linéaire il n'y a alors que trois possibilités :

— il n'y a pas de solution, car les équations sont incompatibles,
— la solution est unique,
— l'ensemble des solutions est un continuum, car une au moins des équations peut être déduite des autres par combinaison linéaire, de sorte qu'il n'y a pas assez d'équations indépendantes.

Pour des équations non linéaires, il y a des possibilités nouvelles :

— une équation scalaire à une inconnue peut ne pas avoir de solution (figure 7.1),
— il peut y avoir plusieurs solutions isolées (figure 7.2).

Nous supposons qu'il existe au moins une solution et que l'ensemble des solutions est fini (ce peut être un singleton, mais nous n'en savons rien).

> Les méthodes présentées ici tentent de trouver des solutions dans cet ensemble fini, sans garantie de succès ou d'exhaustivité. La section 14.5.2.3 mentionne brièvement des méthodes numériques garanties qui recherchent toutes les solutions [171], [115].

## 7.2 Exemples

*Exemple 7.1. Points d'équilibre d'équations différentielles non linéaires*

Les réactions chimiques à température constante dans un réacteur parfaitement agité peuvent être décrites par un système d'équations différentielles ordinaires

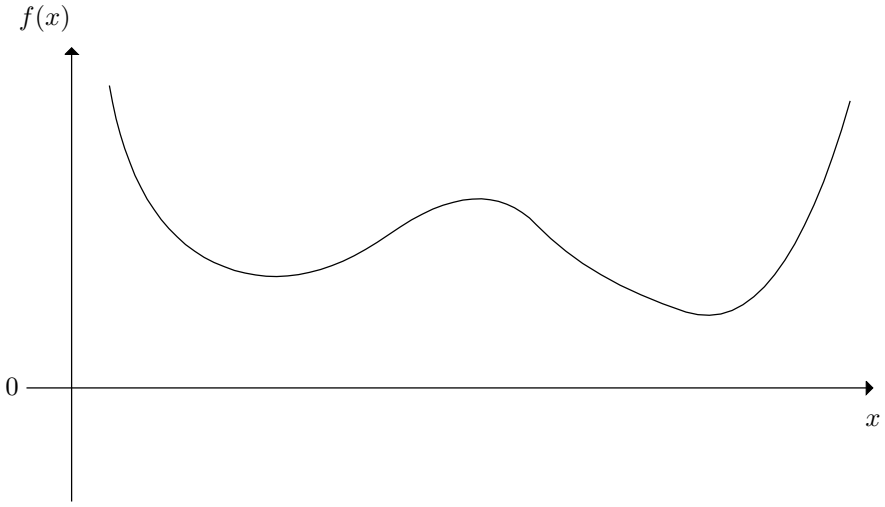


**Fig. 7.1** Une équation non linéaire $f(x) = 0$ sans solution

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \tag{7.1}$$

où $\mathbf{x}$ est un vecteur de concentrations. Les points d'équilibre du réacteur satisfont

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}. \tag{7.2}$$

□

*Exemple 7.2. Plateformes de Stewart-Gough.*

Si vous avez vu un simulateur de vol utilisé pour l'entraînement des pilotes d'avions commerciaux ou militaires, alors vous avez vu une *plateforme de Stewart-Gough*. Les parcs d'attractions utilisent aussi de telles structures. Elles comportent deux plaques rigides, connectées par six vérins hydrauliques dont les longueurs sont contrôlées pour changer la position de l'une des plaques par rapport à l'autre. Dans un simulateur de vol, la plaque de base reste au sol tandis que le siège du pilote est fixé sur la plaque mobile. Les plateformes de Steward-Gough sont des exemples de robots parallèles, puisque les six vérins agissent en parallèle pour déplacer la plaque mobile, contrairement aux effecteurs des bras humanoïdes qui agissent en série. Les robots parallèles forment une alternative attractive aux robots de type série pour les tâches qui demandent de la précision et de la puissance, mais leur commande est plus complexe. Le problème considéré ici est le calcul de toutes les positions possibles de la plaque mobile par rapport à la plaque de base pour une géométrie de la plateforme et des longueurs de vérins donnés. Ces longueurs sont supposées constantes, de sorte que ce problème est statique. Il se traduit par un système de six équations non linéaires à six inconnues (trois angles d'Euler et les trois coordonnées

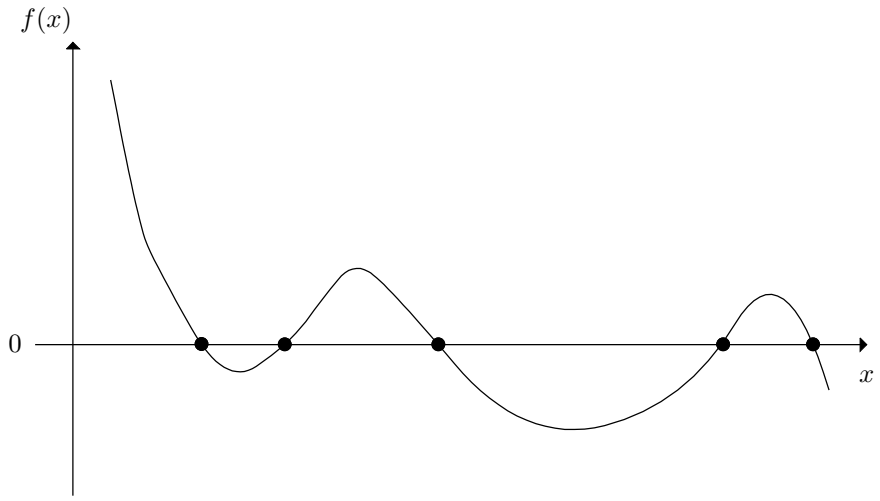


**Fig. 7.2** Une équation non linéaire $f(x) = 0$ avec plusieurs solutions isolées

de la position d'un point donné de la plaque mobile dans le référentiel de la plaque de base). Ces équations impliquent des sinus et des cosinus, et l'on peut préférer considérer un système de neuf équations polynomiales en neuf inconnues. Les inconnues sont alors les sinus et les cosinus des angles d'Euler et les trois coordonnées de la position d'un point donné de la plaque mobile dans le référentiel de la plaque de base, tandis que les équations additionnelles sont

$$\sin^2(\theta_i) + \cos^2(\theta_i) = 1, \quad i = 1,2,3, \tag{7.3}$$

avec $\theta_i$ le $i$-ème angle d'Euler. Le calcul de *toutes* les solutions d'un tel système d'équations est difficile, surtout si l'on ne s'intéresse qu'aux solutions réelles. C'est pourquoi ce problème est devenu un test de référence en calcul formel [83], qui peut aussi être résolu numériquement d'une façon approchée mais garantie grâce à l'analyse par intervalles [52]. Les méthodes décrites dans ce chapitre tentent, plus modestement, de trouver certaines des solutions. □

## 7.3 Une équation à une inconnue

La plupart des méthodes pour résoudre des systèmes d'équations non linéaires à plusieurs inconnues (ou *systèmes multivariés*) sont des extensions de méthodes pour une équation à une inconnue, de sorte que cette section peut servir d'introduction au cas plus général considéré en section 7.4.

Nous voulons trouver la valeur (ou des valeurs) de la variable scalaire $x$ telle(s) que

$$f(x) = 0. \tag{7.4}$$

*Remarque 7.1.* Quand (7.4) est une équation polynomiale, l'itération QR (présentée en section 4.3.6) peut être utilisée pour évaluer toutes ses solutions. □

### *7.3.1 Méthode de bissection*

La méthode de bissection, aussi connue comme sous le nom de *dichotomie*, est la seule méthode présentée en section 7.3 qui n'ait pas de contrepartie pour les systèmes d'équations à plusieurs inconnues en section 7.4. (Les contreparties multivariées de la dichotomie sont fondées sur l'analyse par intervalles ; voir la section 14.5.2.3.) Supposons un intervalle $[a_k, b_k]$ disponible, tel que $f(\cdot)$ soit continue sur $[a_k, b_k]$ et que $f(a_k) \cdot f(b_k) < 0$. L'intervalle $[a_k, b_k]$ contient alors au moins une solution de (7.4). Soit $c_k$ le milieu de $[a_k, b_k]$, donné par

$$c_k = \frac{a_k + b_k}{2}. \tag{7.5}$$

L'intervalle est mis à jour comme suit

$$\text{si} \quad f(a_k) \cdot f(c_k) < 0, \quad \text{alors} \quad [a_{k+1}, b_{k+1}] = [a_k, c_k], \tag{7.6}$$

$$\text{si} \quad f(a_k) \cdot f(c_k) > 0, \quad \text{alors} \quad [a_{k+1}, b_{k+1}] = [c_k, b_k], \tag{7.7}$$

$$\text{si} \quad f(c_k) = 0, \quad \text{alors} \quad [a_{k+1}, b_{k+1}] = [c_k, c_k]. \tag{7.8}$$

L'intervalle résultant $[a_{k+1}, b_{k+1}]$ contient lui aussi au moins une solution de (7.4). Sauf si une solution exacte a été obtenue au milieu du dernier intervalle considéré, la longueur de l'intervalle dans lequel au moins une solution $x^\star$ est enfermée est divisée par deux à chaque itération (figure 7.3).

La méthode ne fournit pas en général d'estimée ponctuelle $x_k$ de $x^\star$, mais une légère modification d'une définition de la section 2.5.3 permet de dire qu'elle converge *linéairement* avec un taux égal à 0.5, puisque

$$\max_{x \in [a_{k+1}, b_{k+1}]} |x - x^\star| = 0.5 \cdot \max_{x \in [a_k, b_k]} |x - x^\star|. \tag{7.9}$$

Tant que l'effet des erreurs d'arrondi peut être négligé, chaque itération augmente ainsi d'une unité le nombre de bits corrects dans la mantisse. Quand on calcule avec des flottants en double précision, il n'y a donc aucune raison de faire plus de 51 itérations après l'obtention d'un premier digit correct, et des précautions particulières doivent être prises pour que les résultats restent garantis, voir la section 14.5.2.3.

*Remarque 7.2.* Quand il y a plusieurs solutions de (7.4) dans $[a_k, b_k]$, la méthode de bissection convergera vers l'une d'entre elles. □

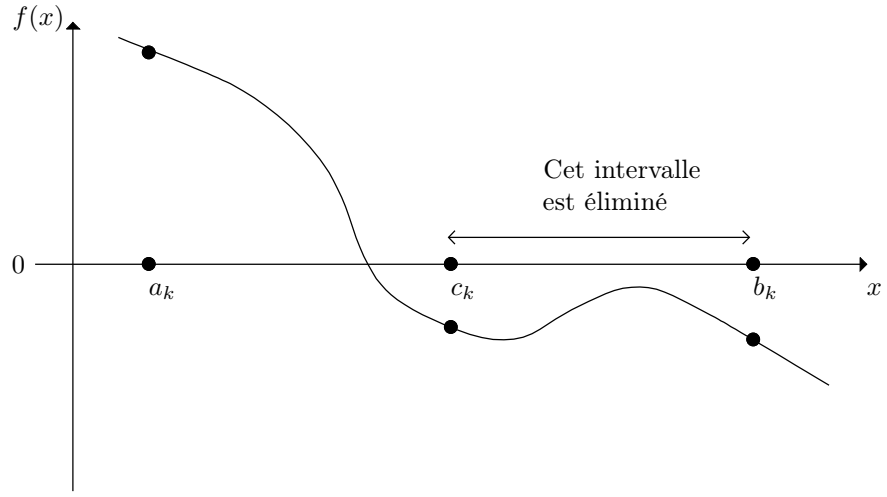


**Fig. 7.3** Méthode de bissection ; $[a_k, c_k]$ contient à coup sûr une solution

### *7.3.2 Méthode de point fixe*

Il est toujours possible de transformer (7.4) en

$$x = \varphi(x), \tag{7.10}$$

par exemple en choisissant

$$\varphi(x) = x + \lambda f(x), \tag{7.11}$$

avec $\lambda \neq 0$ un paramètre à choisir par l'utilisateur. Si elle existe, la limite de l'itération de point fixe

$$x_{k+1} = \varphi(x_k), \quad k = 0, 1, \cdots \tag{7.12}$$

est une solution de (7.4).

La figure 7.4 illustre une situation où l'itération de point fixe converge vers la solution du problème. Une analyse des conditions et de la vitesse de convergence de cette méthode peut être trouvée dans la section 7.4.1.

### *7.3.3 Méthode de la sécante*

Comme avec la méthode de bissection, la $k$-ème itération de la méthode de la sécante utilise la valeur de la fonction en deux points $x_{k-1}$ et $x_k$. Elle ne requiert par

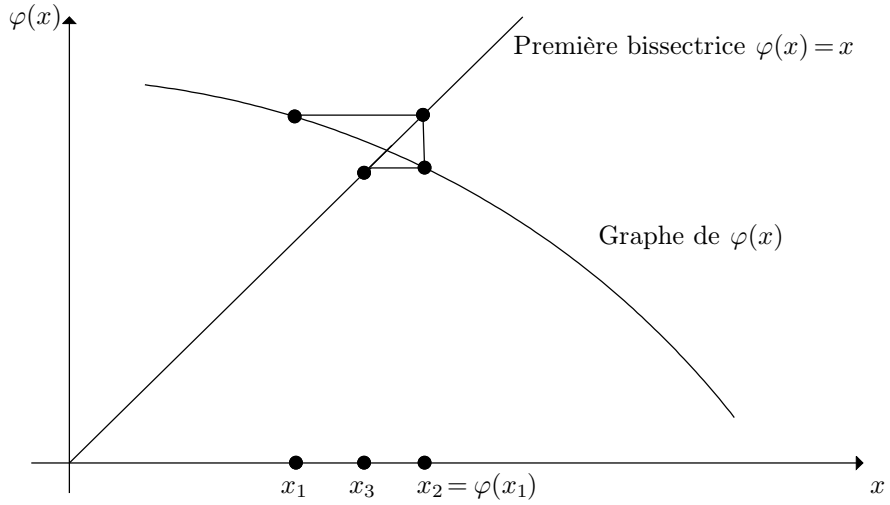


**Fig. 7.4** Itérations de point fixe réussies

contre pas un changement de signe entre $f(x_{k-1})$ et $f(x_k)$. La méthode de la sécante approxime $f(\cdot)$ en interpolant $(x_{k-1}, f(x_{k-1}))$ et $(x_k, f(x_k))$ avec le polynôme du premier ordre

$$P_1(x) = f_k + \frac{f_k - f_{k-1}}{x_k - x_{k-1}}(x - x_k), \tag{7.13}$$

où $f_k$ est une notation condensée pour $f(x_k)$. Le prochain point d'évaluation $x_{k+1}$ est choisi pour annuler $P_1(x_{k+1})$. Une itération calcule donc

$$x_{k+1} = x_k - \frac{(x_k - x_{k-1})}{f_k - f_{k-1}} f_k. \tag{7.14}$$

Comme le montre la figure 7.5, cette procédure peut ne pas converger vers une solution, et le choix des deux points d'évaluation initiaux $x_0$ et $x_1$ est critique.

### *7.3.4 Méthode de Newton*

La méthode de Newton [257] suppose $f(\cdot)$ différentiable et remplace le polynôme interpolateur de la méthode de la sécante par un développement de Taylor au premier ordre de $f(\cdot)$ au voisinage de $x_k$ :

$$f(x) \approx P_1(x) = f(x_k) + \dot{f}(x_k)(x - x_k). \tag{7.15}$$

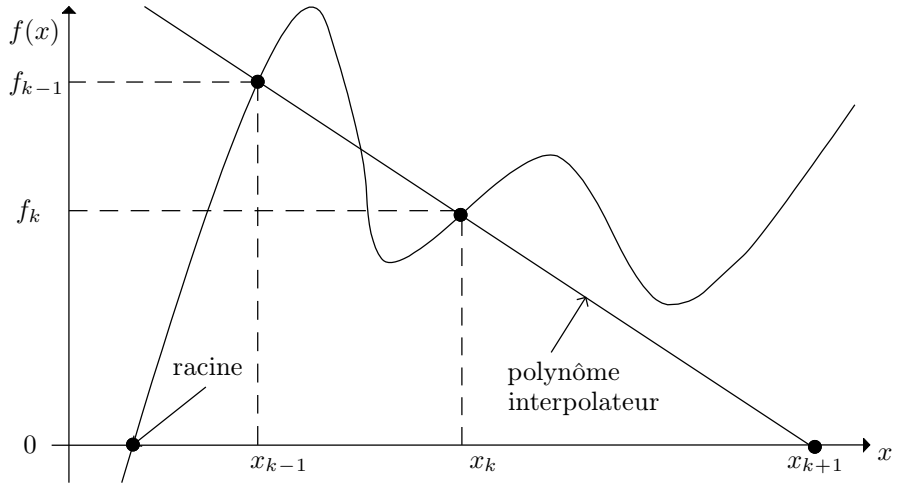


**Fig. 7.5** Échec de la méthode de la sécante

Le prochain point d'évaluation $x_{k+1}$ est choisi pour annuler $P_1(x_{k+1})$. Une itération calcule donc

$$x_{k+1} = x_k - \frac{f(x_k)}{\dot{f}(x_k)}. \tag{7.16}$$

Pour analyser la vitesse de convergence asymptotique de cette méthode, notons $x^\star$ une solution, de sorte que $f(x^\star) = 0$, et développons $f(\cdot)$ au voisinage de $x_k$. Le théorème du reste de Taylor permet d'affirmer qu'il existe $c_k$ entre $x^\star$ et $x_k$ tel que

$$f(x^\star) = f(x_k) + \dot{f}(x_k)(x^\star - x_k) + \frac{\ddot{f}(c_k)}{2}(x^\star - x_k)^2 = 0. \tag{7.17}$$

Quand $\dot{f}(x_k) \neq 0$, ceci implique que

$$\frac{f(x_k)}{\dot{f}(x_k)} + x^\star - x_k + \frac{\ddot{f}(c_k)}{2\dot{f}(x_k)}(x^\star - x_k)^2 = 0. \tag{7.18}$$

Tenons compte de (7.16) pour obtenir

$$x_{k+1} - x^\star = \frac{\ddot{f}(c_k)}{2\dot{f}(x_k)}(x_k - x^\star)^2. \tag{7.19}$$

Quand $x_k$ et $x^\star$ sont suffisamment proches, on en déduit que

$$|x_{k+1} - x^\star| \approx \left| \frac{\ddot{f}(x^\star)}{2\dot{f}(x^\star)} \right| (x_k - x^\star)^2, \tag{7.20}$$

pourvu que les dérivées première et seconde de $f(\cdot)$ soient continues et bornées dans le voisinage de $x^\star$, avec $\dot{f}(x^\star) \neq 0$. La convergence de $x_k$ vers $x^\star$ est alors *quadratique*. Le nombre de digits correct dans la solution devrait approximativement doubler à chaque itération jusqu'à ce que les erreurs d'arrondi prédominent. Ceci est *bien* mieux que la convergence linéaire de la méthode de bissection, mais il y a des inconvénients :

— la méthode de Newton peut ne pas converger vers une solution (voir la figure 7.6),
— il faut évaluer $\dot{f}(x_k)$,
— le choix du point d'évaluation initial $x_0$ est critique.

Réécrivons (7.20) comme

$$|x_{k+1} - x^\star| \approx \rho (x_k - x^\star)^2, \tag{7.21}$$

avec

$$\rho = \left| \frac{\ddot{f}(x^\star)}{2\dot{f}(x^\star)} \right|. \tag{7.22}$$

L'équation (7.21) implique que

$$|\rho(x_{k+1} - x^\star)| \approx [\rho(x_k - x^\star)]^2. \tag{7.23}$$

Ceci suggère de souhaiter que $|\rho(x_0 - x^\star)| < 1$, c'est à dire que

$$|x_0 - x^\star| < \frac{1}{\rho} = \left| \frac{2\dot{f}(x^\star)}{\ddot{f}(x^\star)} \right|, \tag{7.24}$$

bien que la méthode puisse converger vers une solution quand cette condition n'est pas satisfaite.

*Remarque 7.3.* La méthode de Newton rencontre des difficultés quand $\dot{f}(x^\star) = 0$, ce qui arrive quand la racine $x^\star$ est multiple, c'est à dire quand

$$f(x) = (x - x^\star)^m g(x), \tag{7.25}$$

avec $g(x^\star) \neq 0$ et $m > 1$. Sa vitesse de convergence (asymptotique) n'est plus alors que linéaire. Quand le degré de multiplicité $m$ est connu, une vitesse de convergence quadratique peut être restaurée en remplaçant (7.16) par

$$x_{k+1} = x_k - m \frac{f(x_k)}{\dot{f}(x_k)}. \tag{7.26}$$

Quand $m$ n'est pas connu, ou quand $f(\cdot)$ a plusieurs racines multiples, on peut remplacer $f(\cdot)$ dans (7.16) par $h(\cdot)$, avec

$$h(x) = \frac{f(x)}{\dot{f}(x)}, \tag{7.27}$$

car toutes les racines de $h(\cdot)$ sont simples. □

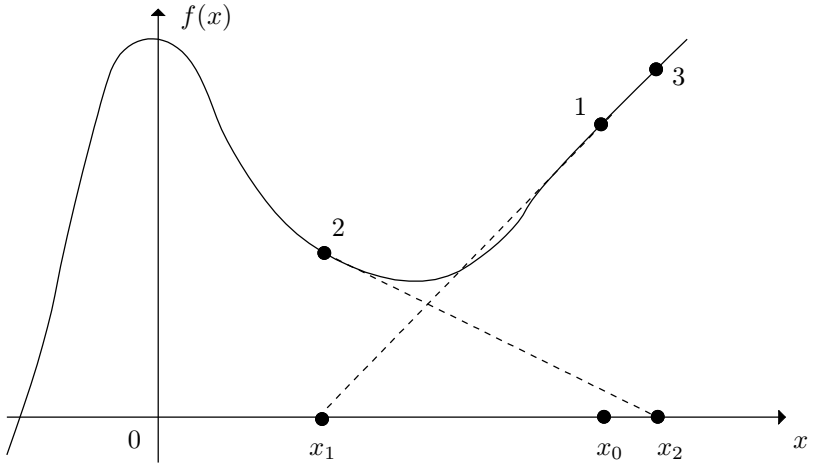


**Fig. 7.6** Échec de la méthode de Newton

Une façon d'échapper à (certains) problèmes de divergence est d'utiliser une méthode de Newton *amortie*

$$x_{k+1} = x_k - \lambda_k \frac{f(x_k)}{\dot{f}(x_k)}, \tag{7.28}$$

où le facteur d'amortissement positif $\lambda_k$ est normalement égal à un, mais décroît quand la valeur absolue de $f(x_{k+1})$ se révèle plus grande que celle de $f(x_k)$, un indice clair du fait que le déplacement $\Delta x = x_{k+1} - x_k$ était trop grand en valeur absolue pour que la fonction soit approximée de façon valide par son développement de Taylor au premier ordre. Si tel est le cas, il faut repartir de $x_k$ et diminuer $\lambda_k$. Ceci assure, au moins mathématiquement, une décroissance monotone de $|f(x_k)|$ au fil des itérations, mais ne garantit toujours pas une convergence vers zéro.

*Remarque 7.4.* Un pas de la méthode de la sécante (7.14) peut être vu comme un pas de la méthode de Newton (7.16) où $\dot{f}(x_k)$ est approximée par une différence finie arrière du premier ordre. Sous les mêmes hypothèses que pour la méthode de Newton, une analyse d'erreur de la méthode de la sécante [225] montre que

$$|x_{k+1} - x^\star| \approx \rho^{\frac{\sqrt{5}-1}{2}} |x_k - x^\star|^{\frac{1+\sqrt{5}}{2}}. \tag{7.29}$$

La vitesse de convergence asymptotique de la méthode de la sécante vers une racine simple $x^\star$ n'est donc pas quadratique, mais reste *superlinéaire*, puisque le *nombre d'or* $(1+\sqrt{5})/2$ est tel que

$$1 < \frac{1+\sqrt{5}}{2} \approx 1.618 < 2. \tag{7.30}$$

Comme pour la méthode de Newton, la vitesse de convergence asymptotique devient linéaire si la racine $x^\star$ est multiple [53].

Rappelons que la méthode de la sécante ne requiert pas l'évaluation de $\dot{f}(x_k)$, de sorte que chaque itération est moins coûteuse qu'avec la méthode de Newton. □

## 7.4 Systèmes d'équations à plusieurs inconnues

Considérons maintenant un ensemble de $n$ équations scalaires en $n$ inconnues scalaires, avec $n > 1$. Il peut être écrit de façon plus concise comme

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \tag{7.31}$$

où $\mathbf{f}(\cdot)$ est une fonction de $\mathbb{R}^n$ vers $\mathbb{R}^n$. Le numéro spécial [247] contient plusieurs articles de présentation de méthodes de résolution de (7.31). Un guide pratique concis pour la résolution de systèmes d'équations non linéaires par la méthode de Newton et ses variantes est [123].

### *7.4.1 Méthode de point fixe*

Comme dans le cas à une inconnue, on peut toujours transformer (7.31) en

$$\mathbf{x} = \boldsymbol{\varphi}(\mathbf{x}), \tag{7.32}$$

par exemple en posant

$$\boldsymbol{\varphi}(\mathbf{x}) = \mathbf{x} + \lambda \mathbf{f}(\mathbf{x}), \tag{7.33}$$

où $\lambda \neq 0$ est un paramètre scalaire à choisir par l'utilisateur. Si elle existe, la limite de l'itération de point fixe

$$\mathbf{x}^{k+1} = \boldsymbol{\varphi}(\mathbf{x}^k), \quad k = 0, 1, \cdots \tag{7.34}$$

est une solution de (7.31).

Cette méthode converge vers la solution $\mathbf{x}^\star$ si $\boldsymbol{\varphi}(\cdot)$ est *contractante*, c'est à dire telle que

$$\exists \alpha < 1 : \forall (\mathbf{x}_1, \mathbf{x}_2), ||\boldsymbol{\varphi}(\mathbf{x}_1) - \boldsymbol{\varphi}(\mathbf{x}_2)|| < \alpha ||\mathbf{x}_1 - \mathbf{x}_2||, \tag{7.35}$$

et plus $\alpha$ est petit mieux c'est.

Pour $\mathbf{x}^1 = \mathbf{x}^k$ et $\mathbf{x}_2 = \mathbf{x}^\star$, (7.35) devient

$$\|\mathbf{x}^{k+1} - \mathbf{x}^\star\| < \alpha \|\mathbf{x}^k - \mathbf{x}^\star\|, \tag{7.36}$$

de sorte que la convergence est linéaire, de taux $\alpha$.

*Remarque 7.5.* Les méthodes itératives de la section 3.7.1 sont des méthodes de point fixe, et donc lentes. C'est un argument de plus en faveur des méthodes par sous-espaces de Krylov présentées en section 3.7.2, qui convergent en au plus $\dim \mathbf{x}$ itérations quand les calculs sont conduits de façon exacte. □

### *7.4.2 Méthode de Newton*

Comme dans le cas à une inconnue, $\mathbf{f}(\cdot)$ est approximée par son développement de Taylor au premier ordre au voisinage de $\mathbf{x}^k$

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}^k) + \mathbf{J}(\mathbf{x}^k)(\mathbf{x} - \mathbf{x}^k), \tag{7.37}$$

où $\mathbf{J}(\mathbf{x}^k)$ est la *jacobienne* $(n \times n)$ de $\mathbf{f}(\cdot)$ évaluée en $\mathbf{x}^k$

$$\mathbf{J}(\mathbf{x}^k) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}^k), \tag{7.38}$$

d'éléments

$$j_{i,l} = \frac{\partial f_i}{\partial x_l}(\mathbf{x}^k). \tag{7.39}$$

Le prochain point d'évaluation $\mathbf{x}^{k+1}$ est choisi pour annuler le membre de droite de (7.37). Une itération calcule ainsi

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \mathbf{J}^{-1}(\mathbf{x}^k)\mathbf{f}(\mathbf{x}^k). \tag{7.40}$$

La jacobienne n'est pas inversée, bien sûr. On évalue plutôt le terme correctif

$$\Delta \mathbf{x}^k = \mathbf{x}^{k+1} - \mathbf{x}^k \tag{7.41}$$

en résolvant le système linéaire

$$\mathbf{J}(\mathbf{x}^k)\Delta \mathbf{x}^k = -\mathbf{f}(\mathbf{x}^k), \tag{7.42}$$

et l'estimée suivante du vecteur solution est donnée par

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta \mathbf{x}^k. \tag{7.43}$$

*Remarque 7.6.* Le conditionnement de $\mathbf{J}(\mathbf{x}^k)$ nous informe sur la difficulté locale du problème, qui dépend de la valeur de $\mathbf{x}^k$. Même si le conditionnement de $\mathbf{J}(\mathbf{x}^\star)$ n'est

pas trop grand, le conditionnement de $\mathbf{J}(\mathbf{x}^k)$ peut prendre des valeurs très grandes pour certaines valeurs de $\mathbf{x}^k$ le long de la trajectoire de l'algorithme. □

Les propriétés de la méthode de Newton dans le cas multivariable sont similaires à celles du cas à une inconnue. Sous les hypothèses suivantes :

- $\mathbf{f}(\cdot)$ est continûment différentiable dans un domaine ouvert convexe $\mathbb{D}$ (H1),
- il existe $\mathbf{x}^\star$ dans $\mathbb{D}$ tel que $\mathbf{f}(\mathbf{x}^\star) = \mathbf{0}$ et $\mathbf{J}(\mathbf{x}^\star)$ est inversible (H2),
- $\mathbf{J}(\cdot)$ satisfait une condition de Lipschitz en $\mathbf{x}^\star$, c'est à dire qu'il existe une constante $\kappa$ telle que

$$\|\mathbf{J}(\mathbf{x}) - \mathbf{J}(\mathbf{x}^\star)\| \leqslant \kappa \|\mathbf{x} - \mathbf{x}^\star\| \quad \text{(H3)},$$

la vitesse de convergence asymptotique est quadratique pourvu que $\mathbf{x}^0$ soit suffisamment proche de $\mathbf{x}^\star$.

En pratique, la méthode peut ne pas converger vers une solution et l'initialisation demeure critique. Là encore, on peut éviter certains problèmes de divergence en utilisant une méthode de Newton amortie,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \lambda_k \Delta \mathbf{x}^k, \tag{7.44}$$

où le facteur d'amortissement positif $\lambda_k$ est normalement égal à un, à moins que $||\mathbf{f}(\mathbf{x}^{k+1})||$ ne se révèle plus grand que $||\mathbf{f}(\mathbf{x}^k)||$, auquel cas $\mathbf{x}^{k+1}$ est rejeté et $\lambda_k$ réduit (typiquement divisé par deux jusqu'à ce que $||\mathbf{f}(\mathbf{x}^{k+1})|| < ||\mathbf{f}(\mathbf{x}^k)||$).

*Remarque 7.7.* Dans le cas particulier d'un système d'équations linéaires $\mathbf{A}\mathbf{x} = \mathbf{b}$, avec $\mathbf{A}$ inversible,

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} \quad \text{et} \quad \mathbf{J} = \mathbf{A}, \quad \text{de sorte que} \quad \mathbf{x}^{k+1} = \mathbf{A}^{-1}\mathbf{b}. \tag{7.45}$$

La méthode de Newton évalue donc la solution unique en un pas. □

*Remarque 7.8.* La méthode de Newton joue aussi un rôle clé en optimisation, voir la section 9.3.4.2. □

### *7.4.3 Méthode de Broyden*

On peut simplifier la méthode de Newton en remplaçant dans (7.42) la jacobienne $\mathbf{J}(\mathbf{x}^k)$ par $\mathbf{J}(\mathbf{x}^0)$, qui est alors évaluée et factorisée une seule fois. La méthode qui en résulte, connue sous le nom de *méthode des cordes*, peut diverger dans des cas où la méthode de Newton convergerait. Les *méthodes de quasi-Newton* se confrontent à cette difficulté en mettant à jour une estimée de la jacobienne (ou de son inverse) à chaque itération [51]. Elles jouent aussi un rôle important en optimisation sans contrainte, voir la section 9.3.4.5.

Dans le contexte des équations non linéaires, la méthode de quasi-Newton la plus populaire est la *méthode de Broyden* [31]. On peut la voir comme une généralisation

de la méthode de la sécante de la section 7.3.3 où $\dot{f}(x_k)$ était approximée par une différence finie (voir la remarque 7.4). L'approximation

$$\dot{f}(x_k) \approx \frac{f_k - f_{k-1}}{x_k - x_{k-1}}, \tag{7.46}$$

devient

$$\mathbf{J}(\mathbf{x}^{k+1})\Delta\mathbf{x} \approx \Delta\mathbf{f}, \tag{7.47}$$

où

$$\Delta\mathbf{x} = \mathbf{x}^{k+1} - \mathbf{x}^k, \tag{7.48}$$

$$\Delta\mathbf{f} = \mathbf{f}(\mathbf{x}^{k+1}) - \mathbf{f}(\mathbf{x}^k). \tag{7.49}$$

L'information fournie par (7.47) est utilisée pour mettre à jour une approximation $\tilde{\mathbf{J}}_k$ de $\mathbf{J}(\mathbf{x}^{k+1})$ suivant

$$\tilde{\mathbf{J}}_{k+1} = \tilde{\mathbf{J}}_k + \mathbf{C}(\Delta\mathbf{x}, \Delta\mathbf{f}), \tag{7.50}$$

avec $\mathbf{C}(\Delta\mathbf{x}, \Delta\mathbf{f})$ une matrice de correction de rang un (c'est à dire le produit à droite d'un vecteur colonne par un vecteur ligne). Pour

$$\mathbf{C}(\Delta\mathbf{x}, \Delta\mathbf{f}) = \frac{(\Delta\mathbf{f} - \tilde{\mathbf{J}}_k\Delta\mathbf{x})}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{x}}\Delta\mathbf{x}^{\mathrm{T}}, \tag{7.51}$$

il est facile de vérifier que la formule de mise à jour (7.50) assure que

$$\tilde{\mathbf{J}}_{k+1}\Delta\mathbf{x} = \Delta\mathbf{f}, \tag{7.52}$$

comme suggéré par (7.47). L'équation (7.52) joue un rôle si central dans les méthodes de quasi-Newton qu'on l'a appelé l'*équation de quasi-Newton*. De plus, pour tout $\mathbf{w}$ tel que $\Delta\mathbf{x}^{\mathrm{T}}\mathbf{w} = 0$,

$$\tilde{\mathbf{J}}_{k+1}\mathbf{w} = \tilde{\mathbf{J}}_k\mathbf{w}, \tag{7.53}$$

de sorte que l'approximation n'est pas modifiée sur le complément orthogonal de $\Delta\mathbf{x}$.

Une autre façon d'arriver à la même correction de rang un est de chercher la matrice $\tilde{\mathbf{J}}_{k+1}$ la plus proche de $\tilde{\mathbf{J}}_k$ pour la norme de Frobenius sous la contrainte (7.52) [51].

Il est plus intéressant, cependant, de mettre à jour une approximation $\mathbf{M} = \tilde{\mathbf{J}}^{-1}$ de l'*inverse* de la jacobienne, de façon à éviter d'avoir à résoudre un système d'équations linéaires à chaque itération. Pourvu que $\tilde{\mathbf{J}}_k$ soit inversible et que

$$1 + \mathbf{v}^{\mathrm{T}}\tilde{\mathbf{J}}_k^{-1}\mathbf{u} \neq 0, \tag{7.54}$$

la formule de Bartlett, Sherman et Morrison [90] implique que

$$\left(\tilde{\mathbf{J}}_k + \mathbf{u}\mathbf{v}^{\mathrm{T}}\right)^{-1} = \tilde{\mathbf{J}}_k^{-1} - \frac{\tilde{\mathbf{J}}_k^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\tilde{\mathbf{J}}_k^{-1}}{1+\mathbf{v}^{\mathrm{T}}\tilde{\mathbf{J}}_k^{-1}\mathbf{u}}. \tag{7.55}$$

Pour mettre à jour l'estimée de $\mathbf{J}^{-1}(\mathbf{x}^{k+1})$ suivant

$$\mathbf{M}_{k+1} = \mathbf{M}_k - \mathbf{C}'(\Delta\mathbf{x}, \Delta\mathbf{f}), \tag{7.56}$$

il suffit de poser

$$\mathbf{u} = \frac{(\Delta\mathbf{f} - \tilde{\mathbf{J}}_k\Delta\mathbf{x})}{\|\Delta\mathbf{x}\|_2} \tag{7.57}$$

et

$$\mathbf{v} = \frac{\Delta\mathbf{x}}{\|\Delta\mathbf{x}\|_2} \tag{7.58}$$

dans (7.51). Comme

$$\tilde{\mathbf{J}}_k^{-1}\mathbf{u} = \frac{\mathbf{M}_k\Delta\mathbf{f} - \Delta\mathbf{x}}{\|\Delta\mathbf{x}\|_2}, \tag{7.59}$$

il n'est pas nécessaire de connaître $\tilde{\mathbf{J}}_k$ pour utiliser (7.55), et

$$\begin{aligned}
\mathbf{C}'(\Delta\mathbf{x}, \Delta\mathbf{f}) &= \frac{\tilde{\mathbf{J}}_k^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\tilde{\mathbf{J}}_k^{-1}}{1+\mathbf{v}^{\mathrm{T}}\tilde{\mathbf{J}}_k^{-1}\mathbf{u}}, \\
&= \frac{\frac{(\mathbf{M}_k\Delta\mathbf{f}-\Delta\mathbf{x})\Delta\mathbf{x}^{\mathrm{T}}\mathbf{M}_k}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{x}}}{1+\frac{\Delta\mathbf{x}^{\mathrm{T}}(\mathbf{M}_k\Delta\mathbf{f}-\Delta\mathbf{x})}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{x}}}, \\
&= \frac{(\mathbf{M}_k\Delta\mathbf{f} - \Delta\mathbf{x})\Delta\mathbf{x}^{\mathrm{T}}\mathbf{M}_k}{\Delta\mathbf{x}^{\mathrm{T}}\mathbf{M}_k\Delta\mathbf{f}}.
\end{aligned} \tag{7.60}$$

Le terme correctif $\mathbf{C}'(\Delta\mathbf{x}, \Delta\mathbf{f})$ est donc aussi une matrice de rang un. Comme avec la méthode de Newton, une procédure d'amortissement est en général utilisée, telle que

$$\Delta\mathbf{x} = \lambda\mathbf{d}, \tag{7.61}$$

où la direction de recherche $\mathbf{d}$ est choisie comme dans la méthode de Newton, avec $\mathbf{J}^{-1}(\mathbf{x}^k)$ remplacée par $\mathbf{M}_k$, de sorte que

$$\mathbf{d} = -\mathbf{M}_k\mathbf{f}(\mathbf{x}^k). \tag{7.62}$$

Le terme correctif devient alors

$$\mathbf{C}'(\Delta\mathbf{x}, \Delta\mathbf{f}) = \frac{(\mathbf{M}_k\Delta\mathbf{f} - \lambda\mathbf{d})\mathbf{d}^{\mathrm{T}}\mathbf{M}_k}{\mathbf{d}^{\mathrm{T}}\mathbf{M}_k\Delta\mathbf{f}}. \tag{7.63}$$

En résumé, partant de $k = 0$ et de la paire $(\mathbf{x}^0, \mathbf{M}_0)$, ($\mathbf{M}_0$ peut être prise égale à $\mathbf{J}^{-1}(\mathbf{x}^0)$, ou plus simplement à la matrice identité,) la méthode procède comme suit :

1. Calculer $\mathbf{f}^k = \mathbf{f}(\mathbf{x}^k)$.
2. Calculer $\mathbf{d} = -\mathbf{M}_k\mathbf{f}^k$.

3. Trouver $\widehat{\lambda}$ tel que
$$\|\mathbf{f}(\mathbf{x}^k + \widehat{\lambda}\mathbf{d})\| < \|\mathbf{f}^k\| \tag{7.64}$$
et prendre
$$\mathbf{x}^{k+1} = \mathbf{x}^k + \widehat{\lambda}\mathbf{d}, \tag{7.65}$$
$$\mathbf{f}^{k+1} = \mathbf{f}(\mathbf{x}^{k+1}). \tag{7.66}$$
4. Calculer $\Delta\mathbf{f} = \mathbf{f}^{k+1} - \mathbf{f}^k$.
5. Calculer
$$\mathbf{M}_{k+1} = \mathbf{M}_k - \frac{(\mathbf{M}_k\Delta\mathbf{f} - \widehat{\lambda}\mathbf{d})\mathbf{d}^{\mathrm{T}}\mathbf{M}_k}{\mathbf{d}^{\mathrm{T}}\mathbf{M}_k\Delta\mathbf{f}}. \tag{7.67}$$
6. Incrémenter $k$ d'une unité et répéter du pas 2.

Sous les mêmes hypothèses (H1) à (H3) sous lesquelles la méthode de Newton converge de façon quadratique, la méthode de Broyden converge superlinéairement (pourvu que $\mathbf{x}^0$ soit suffisamment proche de $\mathbf{x}^\star$ et $\mathbf{M}_0$ suffisamment proche de $\mathbf{J}^{-1}(\mathbf{x}^\star)$) [51]. Ceci ne signifie pas nécessairement que la méthode de Broyden requiert plus de calculs que la méthode de Newton, puisque les itérations de Broyden sont souvent bien plus simples que celles de Newton.

## 7.5 D'où partir ?

Toutes les méthodes présentées ici pour résoudre des systèmes d'équations non linéaires sont itératives. A l'exception de la méthode par bissection, qui est fondée sur un raisonnement sur les intervalles et améliore à coup sûr la précision avec laquelle une solution est localisée, elles partent d'un point d'évaluation initial (de deux points pour la méthode de la sécante) pour calculer de nouveaux points d'évaluation dont on espère qu'ils seront plus proches de l'*une* des solutions. Même si une bonne approximation d'une solution est connue a priori, et à moins que le temps de calcul ne l'interdise, c'est alors une bonne idée que d'essayer plusieurs points initiaux tirés au hasard dans le domaine d'intérêt $\mathbb{X}$. Cette stratégie, connue sous le nom de *multistart*, est une tentative particulièrement simple de trouver des solutions par recherche aléatoire. Bien qu'elle puisse parfois trouver toutes les solutions, il n'y a aucune garantie qu'elle y parvienne.

*Remarque 7.9.* Les *méthodes de continuation*, encore appelées *méthodes par homotopie*, sont une alternative intéressante au *multistart*. Elles transforment progressivement les solutions connues d'un système d'équations $\mathbf{e}(\mathbf{x}) = \mathbf{0}$ facile à résoudre en celles de (7.31). Dans ce but, elles résolvent
$$\mathbf{h}_\lambda(\mathbf{x}) = \mathbf{0}, \tag{7.68}$$
où

$$\mathbf{h}_\lambda(\mathbf{x}) = \lambda \mathbf{f}(\mathbf{x}) + (1-\lambda)\mathbf{e}(\mathbf{x}), \tag{7.69}$$

avec $\lambda$ variant de zéro à un. En pratique, il est souvent nécessaire de permettre à $\lambda$ de décroître temporairement sur la route de zéro à un, et la mise en œuvre n'est pas triviale. Voir [173] pour une introduction. □

## 7.6 Quand s'arrêter ?

On ne peut pas permettre aux algorithmes itératifs de tourner sans fin. Il faut donc spécifier des critères d'arrêt. Mathématiquement, on devrait arrêter quand une solution a été atteinte, c'est à dire quand $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. Du point de vue du calcul numérique, ceci n'a pas grand sens, et on peut décider à la place d'arrêter quand $||\mathbf{f}(\mathbf{x}^k)|| < \delta$, où $\delta$ est un seuil positif à choisir par l'utilisateur, ou quand $||\mathbf{f}(\mathbf{x}^k) - \mathbf{f}(\mathbf{x}^{k-1})|| < \delta$. Le premier de ces deux critères peut n'être jamais satisfait si $\delta$ est trop petit ou si $\mathbf{x}^0$ a été mal choisi, ce qui fournit une motivation pour utiliser le second.

Avec l'une ou l'autre de ces stratégies, le nombre d'itérations pourra changer drastiquement pour un seuil donné si on multiplie de façon arbitraire les équations par un nombre réel très grand ou très petit. On peut préférer un critère d'arrêt qui ne présente pas cette propriété, comme d'arrêter quand

$$||\mathbf{f}(\mathbf{x}^k)|| < \delta ||\mathbf{f}(\mathbf{x}^0)|| \tag{7.70}$$

(ce qui peut ne jamais arriver) ou quand

$$||\mathbf{f}(\mathbf{x}^k) - \mathbf{f}(\mathbf{x}^{k-1})|| < \delta ||\mathbf{f}(\mathbf{x}^k) + \mathbf{f}(\mathbf{x}^{k-1})||. \tag{7.71}$$

On peut aussi décider d'arrêter quand

$$\frac{||\mathbf{x}^k - \mathbf{x}^{k-1}||}{||\mathbf{x}^k|| + \text{realmin}} \leqslant \text{eps}, \tag{7.72}$$

ou quand

$$\frac{||\mathbf{f}(\mathbf{x}^k) - \mathbf{f}(\mathbf{x}^{k-1})||}{||\mathbf{f}(\mathbf{x}^k)|| + \text{realmin}} \leqslant \text{eps}, \tag{7.73}$$

où *eps* est la précision relative de la représentation flottante employée (encore appelée *epsilon machine*), et *realmin* le plus petit nombre flottant normalisé strictement positif, placé aux dénominateurs des membres de gauche de (7.72) et (7.73) pour protéger contre des divisions par zéro. Quand des flottants en double précision sont utilisés, comme dans MATLAB, les ordinateurs à la norme IEEE 754 vérifient

$$\text{eps} \approx 2.22 \cdot 10^{-16} \tag{7.74}$$

et

$$\text{realmin} \approx 2.225 \cdot 10^{-308}. \tag{7.75}$$

Une dernière idée intéressante est d'arrêter quand il n'y a plus de chiffre significatif dans le résultat de l'évaluation de $\mathbf{f}(\mathbf{x}^k)$, c'est à dire quand l'on n'est plus sûr qu'une solution n'a pas été atteinte. Il faut pour cela disposer de méthodes pour évaluer la précision de résultats numériques, comme celles décrites au chapitre 14.

Plusieurs critères d'arrêt peuvent être combinés, et il faut aussi spécifier un nombre maximal d'itérations, ne serait-ce qu'en protection contre d'autres tests mal conçus.

## 7.7 Exemples MATLAB

### *7.7.1 Une équation à une inconnue*

Quand $f(x) = x^2 - 3$, l'équation $f(x) = 0$ a deux solutions réelles, à savoir

$$x = \pm\sqrt{3} \approx \pm 1.732050807568877. \tag{7.76}$$

Résolvons la avec les quatre méthodes présentées en section 7.3.

#### 7.7.1.1 Utilisation de la méthode de Newton

Un script très primitif mettant (7.16) en œuvre est

```
clear all
Kmax = 10;
x = zeros(Kmax,1);
x(1) = 1;
f = @(x) x.^2-3;
fdot = @(x) 2*x;
for k=1:Kmax,
    x(k+1) = x(k)-f(x(k))/fdot(x(k));
end
x
```

Il produit

```
x =
     1.000000000000000e+00
     2.000000000000000e+00
     1.750000000000000e+00
     1.732142857142857e+00
     1.732050810014728e+00
     1.732050807568877e+00
     1.732050807568877e+00
```

```
 1.732050807568877e+00
 1.732050807568877e+00
 1.732050807568877e+00
 1.732050807568877e+00
```

Bien qu'une solution précise soit obtenue très rapidement, ce script peut être amélioré de multiples façons.

Tout d'abord, il n'y a aucune raison d'itérer quand la solution a été obtenue (au moins au niveau de précision de la représentation à virgule flottante utilisée). Une règle d'arrêt plus sophistiquée qu'un simple nombre maximum d'itérations doit donc être spécifiée. On peut, par exemple, utiliser (7.72) et remplacer la boucle du script précédent par

```
for k=1:Kmax,
    x(k+1) = x(k)-f(x(k))/fdot(x(k));
    if ((abs(x(k+1)-x(k)))/(abs(x(k+1)+realmin))<=eps)
        break
    end
end
```

Cette nouvelle boucle termine après seulement six itérations.

Une deuxième amélioration est de mettre en œuvre une stratégie *multistart*, de façon à rechercher d'autres solutions. On peut écrire, par exemple,

```
clear all
Smax = 10; % nombre d'initialisations
Kmax = 10; % nombre maximal d'itérations
           % par initialisation
Init = 2*rand(Smax,1)-1; % entre -1 et 1
x = zeros(Kmax,1);
Solutions = zeros(Smax,1);
f = @(x) x.^2-3;
fdot = @(x) 2*x;
for i=1:Smax,
    x(1) = Init(i);
    for k=1:Kmax,
        x(k+1) = x(k)-f(x(k))/fdot(x(k));
        if ((abs(x(k+1)-x(k)))/...
                (abs(x(k+1)+realmin))<=eps)
            break
        end
    end
    Solutions(i) = x(k+1);
end
Solutions
```

dont une exécution typique fournit

```
Solutions =
    -1.732050807568877e+00
    -1.732050807568877e+00
    -1.732050807568877e+00
     1.732050807568877e+00
     1.732050807568877e+00
     1.732050807568877e+00
     1.732050807568877e+00
     1.732050807568877e+00
    -1.732050807568877e+00
     1.732050807568877e+00
```

Les deux solutions ont donc été localisées (rappelons qu'il n'y a pas de garantie que le *multistart* y parvienne). Il n'a pas été nécessaire d'introduire un amortissement sur ce problème simple.

#### 7.7.1.2 Utilisation de la méthode de la sécante

Il est facile de transformer le script qui précède pour mettre en œuvre (7.14), comme suit

```
clear all
Smax = 10; % nombre d'initialisations
Kmax = 20; % nombre maximal d'itérations
           % par initialisation
Init = 2*rand(Smax,1)-1; % entre -1 et 1
x = zeros(Kmax,1);
Solutions = zeros(Smax,1);
f = @(x) x.^2-3;
for i=1:Smax,
    x(1) = Init(i);
    x(2) = x(1)+0.1; % pas très malin...
    for k=2:Kmax,
        x(k+1) = x(k) - (x(k)-x(k-1))...
            *f(x(k))/(f(x(k))-f(x(k-1)));
        if ((abs(x(k+1)-x(k)))/...
                (abs(x(k+1)+realmin))<=eps)
            break
        end
    end
    Solutions(i) = x(k+1);
end
Solutions
```

La boucle interne est typiquement interrompue après une douzaine d'itérations, ce qui confirme que la méthode de la sécante est plus lente que la méthode de Newton, et une exécution typique produit

```
Solutions =
     1.732050807568877e+00
     1.732050807568877e+00
    -1.732050807568877e+00
    -1.732050807568877e+00
    -1.732050807568877e+00
     1.732050807568877e+00
    -1.732050807568877e+00
     1.732050807568877e+00
    -1.732050807568877e+00
     1.732050807568877e+00
```

de sorte que la méthode de la sécante avec *multistart* est capable de trouver les deux solutions avec la même précision que la méthode de Newton.

#### 7.7.1.3 Utilisation d'une itération de point fixe

Essayons

$$x_{k+1} = x_k + \lambda(x_k^2 - 3), \tag{7.77}$$

comme mis en œuvre dans le script

```
clear all
lambda = 0.5 % réglable
Kmax = 50;   % nombre maximal d'itérations
f = @(x) x.^2-3;
x = zeros(Kmax+1,1);
x(1) = 2*rand(1)-1; % entre -1 et 1
for k=1:Kmax,
    x(k+1) = x(k)+lambda*f(x(k));
end
Solution = x(Kmax+1)
```

Il faut tâtonner un peu pour trouver une valeur de $\lambda$ qui assure la convergence vers une solution approchée. Pour $\lambda = 0.5$, le script converge vers une approximation de $-\sqrt{3}$ tandis que pour $\lambda = -0.5$ il converge vers une approximation de $\sqrt{3}$. Dans les deux cas, la convergence est encore plus lente qu'avec la méthode de la sécante. Pour $\lambda = 0.5$, par exemple, 50 itérations d'une exécution typique ont produit

```
Solution = -1.732050852324972e+00
```

et 100 itérations

```
Solution = -1.732050807568868e+00
```

#### 7.7.1.4 Utilisation de la méthode par bissection

Le script qui suit recherche une solution dans $[0,2]$, dont on sait qu'elle existe puisque $f(\cdot)$ est continue et que $f(0)\cdot f(2)<0$.

```
clear all
lower = zeros(52,1);
upper = zeros(52,1);
tiny = 1e-12;     % seuil d'arrêt
f = @(x) x.^2-3;
a = 0;
b = 2.;
lower(1) = a;
upper(1) = b;
for i=2:63
    c = (a+b)/2;
    if (f(c) == 0)
        break;
    elseif (b-a<tiny)    % intervalle trop petit
        break;
    elseif (f(a)*f(c)<0)
        b = c;
    else
        a = c;
    end
    lower(i) = a;
    upper(i) = b;
end
lower
upper
```

La convergence des bornes de $[a,b]$ vers $\sqrt{3}$ est lente, comme mis en évidence ci-dessous par leurs dix premières valeurs.

```
lower =
                         0
     1.000000000000000e+00
     1.500000000000000e+00
     1.500000000000000e+00
     1.625000000000000e+00
     1.687500000000000e+00
     1.718750000000000e+00
     1.718750000000000e+00
     1.726562500000000e+00
     1.730468750000000e+00
```

et

```
upper =
     2.000000000000000e+00
     2.000000000000000e+00
     2.000000000000000e+00
     1.750000000000000e+00
     1.750000000000000e+00
     1.750000000000000e+00
     1.750000000000000e+00
     1.734375000000000e+00
     1.734375000000000e+00
     1.734375000000000e+00
```

Le dernier intervalle calculé est

$$[a,b] = [1.732050807568157, 1.732050807569067]. \tag{7.78}$$

Sa longueur est en effet inférieure à $10^{-12}$, et il contient bien $\sqrt{3}$.

## *7.7.2 Systèmes d'équations à plusieurs inconnues*

Le système d'équations

$$\begin{aligned} x_1^2 x_2^2 &= 9, & (7.79)\\ x_1^2 x_2 - 3x_2 &= 0. & (7.80) \end{aligned}$$

peut s'écrire $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, avec

$$\begin{aligned} \mathbf{x} &= (x_1, x_2)^{\mathrm{T}}, & (7.81)\\ f_1(\mathbf{x}) &= x_1^2 x_2^2 - 9, & (7.82)\\ f_2(\mathbf{x}) &= x_1^2 x_2 - 3x_2. & (7.83) \end{aligned}$$

Il a quatre solutions en $x_1$ et $x_2$, avec $x_1 = \pm\sqrt{3}$ et $x_2 = \pm\sqrt{3}$. Résolvons-le avec deux méthodes présentées en section 7.4 et une méthode non présentée.

### 7.7.2.1 Utilisation de la méthode de Newton

La méthode de Newton exploite la jacobienne de $\mathbf{f}(\cdot)$, donnée par

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 x_2^2 & 2x_1^2 x_2 \\ 2x_1 x_2 & x_1^2 - 3 \end{bmatrix}. \tag{7.84}$$

La fonction **f** et sa jacobienne **J** sont évaluées par la fonction MATLAB suivante

```
function[F,J] = SysNonLin(x)
% fonction
F = zeros(2,1);
J = zeros(2,2);
F(1) = x(1)^2*x(2)^2-9;
F(2) = x(1)^2*x(2)-3*x(2);
% jacobienne
J(1,1) = 2*x(1)*x(2)^2;
J(1,2) = 2*x(1)^2*x(2);
J(2,1) = 2*x(1)*x(2);
J(2,2) = x(1)^2-3;
end
```

La méthode de Newton (non amortie) avec *multistart* est mise en œuvre par le script

```
clear all
format LONGE
Smax = 10; % nombre d'initialisations
Kmax = 20; % nombre maximal d'itérations
           % par initialisation
Init = 2*rand(2,Smax)-1; % éléments entre -1 et 1
Solutions = zeros(Smax,2);
X = zeros(2,1);
Xplus = zeros(2,1);
for i=1:Smax
    X = Init(:,i);
    for k=1:Kmax
        [F,J] = SysNonLin(X);
        DeltaX = -J\F;
        Xplus = X + DeltaX;
        [Fplus] = SysNonLin(Xplus);
        if (norm(Fplus-F)/(norm(F)+realmin)<=eps)
            break
        end
        X = Xplus;
    end
    Solutions(i,:) = Xplus;
end
Solutions
```

Une exécution typique de ce script produit

```
Solutions =
     1.732050807568877e+00     1.732050807568877e+00
    -1.732050807568877e+00     1.732050807568877e+00
     1.732050807568877e+00    -1.732050807568877e+00
     1.732050807568877e+00    -1.732050807568877e+00
```

```
 1.732050807568877e+00    -1.732050807568877e+00
-1.732050807568877e+00    -1.732050807568877e+00
-1.732050807568877e+00     1.732050807568877e+00
-1.732050807568877e+00     1.732050807568877e+00
-1.732050807568877e+00    -1.732050807568877e+00
-1.732050807568877e+00     1.732050807568877e+00
```

où chaque ligne correspond à la solution évaluée pour une valeur initiale donnée de **x**. Les quatre solutions ont donc été évaluées de façon précise, et l'amortissement n'était une fois de plus pas nécessaire sur ce problème simple.

*Remarque 7.10.* Le calcul formel sur ordinateur peut ici être utilisé pour générer l'expression formelle de la jacobienne. Le script qui suit utilise la *Symbolic Math Toolbox* pour ce faire.

```
syms x y
X = [x;y]
F = [x^2*y^2-9;x^2*y-3*y]
J = jacobian(F,X)
```

Il produit

```
X =
 x
 y

F =
 x^2*y^2 - 9
 y*x^2 - 3*y

J =
[ 2*x*y^2, 2*x^2*y]
[   2*x*y, x^2 - 3]
```

□

#### 7.7.2.2 Utilisation de `fsolve`

Le script qui suit tente de résoudre (7.79) avec `fsolve`, fourni dans l'*Optimization Toolbox* et fondé sur la minimisation de

$$J(\mathbf{x}) = \sum_{i=1}^{n} f_i^2(\mathbf{x}) \tag{7.85}$$

par la méthode de Levenberg et Marquardt, présentée en section 9.3.4.4, ou par une autre variante robuste de la méthode de Newton (voir la documentation de `fsolve` pour plus de détails). La fonction **f** et sa jacobienne **J** sont évaluées par la même fonction MATLAB qu'en section 7.7.2.1.

```
clear all
Smax = 10; % nombre d'initialisations
Init = 2*rand(Smax,2)-1; % entre -1 et 1
Solutions = zeros(Smax,2);
options = optimset('Jacobian','on');
for i=1:Smax
    x0 = Init(i,:);
    Solutions(i,:) = fsolve(@SysNonLin,x0,options);
end
Solutions
```

Un résultat typique est

```
Solutions =
    -1.732050808042171e+00    -1.732050808135796e+00
     1.732050807568913e+00     1.732050807568798e+00
    -1.732050807570181e+00    -1.732050807569244e+00
     1.732050807120480e+00     1.732050808372865e+00
    -1.732050807568903e+00     1.732050807568869e+00
     1.732050807569296e+00     1.732050807569322e+00
     1.732050807630857e+00    -1.732050807642701e+00
     1.732050807796109e+00    -1.732050808527067e+00
    -1.732050807966248e+00    -1.732050807938446e+00
    -1.732050807568886e+00     1.732050807568879e+00
```

où chaque ligne correspond à la solution évaluée pour une valeur initiale donnée de **x**. Les quatre solutions ont donc été trouvées, quoique moins précisément qu'avec la méthode de Newton.

#### 7.7.2.3 Utilisation de la méthode de Broyden

Le m-file de la méthode de Broyden fourni par John Penny [144] est disponible via le *MATLAB central file exchange*. Il est utilisé dans le script qui suit sous le nom de `BroydenByPenny`.

```
clear all
Smax = 10; % nombre d'initialisations
Init = 2*rand(2,Smax)-1; % entre -1 et 1
Solutions = zeros(Smax,2);
NumberOfIterations = zeros(Smax,1);
n = 2;
tol = 1.e-10;
for i=1:Smax
    x0 = Init(:,i);
    [Solutions(i,:), NumberOfIterations(i)]...
        = BroydenByPenny(x0,@SysNonLin,n,tol);
```

```
end
Solutions
NumberOfIterations
```

Une exécution typique de ce script produit

```
Solutions =
    -1.732050807568899e+00    -1.732050807568949e+00
    -1.732050807568901e+00     1.732050807564629e+00
     1.732050807568442e+00    -1.732050807570081e+00
    -1.732050807568877e+00     1.732050807568877e+00
     1.732050807568591e+00     1.732050807567701e+00
     1.732050807569304e+00     1.732050807576298e+00
     1.732050807568429e+00    -1.732050807569200e+00
     1.732050807568774e+00     1.732050807564450e+00
     1.732050807568853e+00    -1.732050807568735e+00
    -1.732050807568868e+00     1.732050807568897e+00
```

Le nombre des itérations nécessaires à l'obtention de chacune de ces dix paires de résultats varie de 18 à 134 (quoique l'une des paires de résultats d'une autre exécution ait été obtenue après 291 503 itérations). Rappelons que la méthode de Broyden n'utilise pas la jacobienne de **f**, contrairement aux deux autres méthodes présentées.

Si, en poussant notre chance, nous tentons d'obtenir des résultats plus précis en imposant `tol = 1.e-15;` alors une exécution typique produit

```
Solutions =
                      NaN                       NaN
                      NaN                       NaN
                      NaN                       NaN
                      NaN                       NaN
     1.732050807568877e+00     1.732050807568877e+00
     1.732050807568877e+00    -1.732050807568877e+00
                      NaN                       NaN
                      NaN                       NaN
     1.732050807568877e+00     1.732050807568877e+00
     1.732050807568877e+00    -1.732050807568877e+00
```

Si certains résultats sont en effet plus précis, la méthode échoue donc dans un nombre significatif de cas, comme indiqué par `NaN`, l'acronyme de *Not a Number*.

## 7.8 En résumé

— La résolution de systèmes d'équations non linéaires est beaucoup plus complexe qu'avec des équations linéaires. On peut ne pas savoir à l'avance le nombre des solutions, ou même s'il existe une solution.

— Les techniques présentées dans ce chapitre sont itératives, et visent principalement à trouver l'une de ces solutions.
— La qualité d'un vecteur $\mathbf{x}^k$ candidat au titre de solution peut être évaluée en calculant $\mathbf{f}(\mathbf{x}^k)$.
— Si une méthode de recherche échoue, cela ne prouve pas qu'il n'y a pas de solution.
— La vitesse de convergence asymptotique pour des racines isolées est typiquement linéaire pour une méthode de point fixe, superlinéaire pour les méthodes de la sécante et de Broyden et quadratique pour la méthode de Newton.
— L'initialisation joue un rôle crucial, et le *multistart* peut être utilisé pour explorer le domaine d'intérêt à la recherche de toutes les solutions qu'il contient. Il n'y a cependant pas de garantie que cette stratégie réussisse.
— Pour un budget de calcul donné, arrêter d'itérer dès que possible permet d'essayer d'autres points de départ.

# Chapitre 8
# Introduction à l'optimisation

## 8.1 Un mot de mise en garde

Savoir optimiser un indice de performance n'implique pas que ce soit une bonne idée de le faire. Minimiser, par exemple, le nombre de transistors dans un circuit intégré ou le nombre de lignes de code dans un programme peut conduire à des produits complexes à comprendre, à corriger, à documenter et à mettre à jour. Avant de s'embarquer dans une optimisation, il faut donc s'assurer qu'elle fait sens pour le vrai problème qu'il s'agit de résoudre.

Quand tel est le cas, les conséquences du choix d'un indice de performance spécifique ne doivent pas être sous-estimées. Pour minimiser une somme de valeurs absolues, par exemple, il est préférable d'utiliser d'autres méthodes que pour minimiser une somme de carrés, et la solution optimale obtenue sera différente.

Il y a de nombreux livres introductifs excellents sur des aspects variés de l'optimisation, dont [185, 155, 73, 122, 173, 14, 169, 22, 10]. On trouvera des exposés introductifs intéressants dans [247]. La seconde édition de l'*Encyclopedia of Optimization* récemment parue ne contient pas moins de 4626 pages d'articles introductifs et de synthèses [63].

## 8.2 Exemples

*Exemple 8.1. Estimation de paramètres*

Pour estimer les paramètres d'un modèle mathématique à partir de données expérimentales, une approche classique est de chercher la valeur (qu'on espère unique) du vecteur de paramètres $\mathbf{x} \in \mathbb{R}^n$ qui minimise la fonction de coût quadratique

$$J(\mathbf{x}) = \mathbf{e}^{\mathrm{T}}(\mathbf{x})\mathbf{e}(\mathbf{x}) = \sum_{i=1}^{N} e_i^2(\mathbf{x}), \tag{8.1}$$

où le vecteur d'erreur $\mathbf{e}(\mathbf{x}) \in \mathbb{R}^N$ est la différence entre un vecteur $\mathbf{y}$ de données expérimentales et un vecteur $\mathbf{y}_\mathrm{m}(\mathbf{x})$ de sorties correspondantes du modèle

$$\mathbf{e}(\mathbf{x}) = \mathbf{y} - \mathbf{y}_\mathrm{m}(\mathbf{x}). \tag{8.2}$$

Le plus souvent, on n'impose aucune contrainte à $\mathbf{x}$, qui peut donc prendre n'importe quelle valeur dans $\mathbb{R}^n$, de sorte que c'est de l'*optimisation sans contrainte*, considérée au chapitre 9. □

*Exemple 8.2. Management*

Une compagnie peut souhaiter maximiser ses bénéfices sous des contraintes sur sa production, minimiser le coût d'un produit sous des contraintes sur ses performances ou minimiser le temps nécessaire à sa mise sur le marché sous des contraintes de coût. C'est de l'*optimisation sous contraintes*, considérée au chapitre 10. □

*Exemple 8.3. Logistique*

Un voyageur de commerce peut souhaiter visiter un ensemble de villes en minimisant la distance totale à parcourir. La solution optimale est alors une liste ordonnée de villes, pas nécessairement codée sous forme numérique. C'est de l'*optimisation combinatoire*, considérée au chapitre 11. □

## 8.3 Taxonomie

Un synonyme d'optimisation est *programmation*. Ce terme a été choisi par des mathématiciens travaillant sur la logistique pendant la seconde guerre mondiale, avant l'omniprésence de l'ordinateur. Dans ce contexte, un *programme* est un problème d'optimisation.

La *fonction d'objectif* (ou *indice de performance*) $J(\cdot)$ est une fonction à valeur scalaire de *n variables de décision* scalaires $x_i$, $i = 1, \cdots, n$. Ces variables sont placées dans un vecteur de décision $\mathbf{x}$, et l'*ensemble admissible* $\mathbb{X}$ est l'ensemble des valeurs que $\mathbf{x}$ peut prendre. Quand la fonction d'objectif doit être minimisée, c'est une *fonction de coût*. Quand elle doit être maximisée, c'est une *fonction d'utilité*. Il est trivial de transformer une fonction d'utilité $U(\cdot)$ en une fonction de coût $J(\cdot)$, par exemple en posant

$$J(\mathbf{x}) = -U(\mathbf{x}). \tag{8.3}$$

Il n'y a donc pas de perte de généralité à ne considérer que des problèmes de minimisation. La notation

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}\in\mathbb{X}} J(\mathbf{x}) \tag{8.4}$$

signifie que

$$\forall \mathbf{x} \in \mathbb{X}, \quad J(\widehat{\mathbf{x}}) \leqslant J(\mathbf{x}). \tag{8.5}$$

Tout $\widehat{\mathbf{x}}$ qui satisfait (8.5) est un *minimiseur global*, et le coût correspondant $J(\widehat{\mathbf{x}})$ est le *minimum global*. Notons que le minimum global est unique quand il existe, tandis qu'il peut y avoir plusieurs minimiseurs globaux.

Les deux exemples qui suivent illustrent des situations à éviter, si possible.

*Exemple 8.4.* Quand $J(x) = -x$ et $\mathbb{X}$ est un intervalle ouvert $(a,b) \subset \mathbb{R}$ (c'est à dire que l'intervalle ne contient pas ses extrémités $a$ et $b$), il n'y a pas de minimiseur (ou maximiseur) global, ni de minimum (ou maximum) global. L'*infimum* est $J(b)$, et le *supremum* $J(a)$. □

*Exemple 8.5.* Quand $J(x) = x$ et $\mathbb{X} = \mathbb{R}$, il n'y a pas de minimiseur (ou maximiseur) global, ni de minimum (ou maximum) global. l'infimum est $-\infty$ et le supremum $+\infty$. □

Si l'on sait seulement que (8.5) est valide dans un voisinage $\mathbb{V}(\widehat{\mathbf{x}})$ de $\widehat{\mathbf{x}}$, c'est à dire que

$$\forall \mathbf{x} \in \mathbb{V}(\widehat{\mathbf{x}}), \quad J(\widehat{\mathbf{x}}) \leqslant J(\mathbf{x}), \tag{8.6}$$

alors $\widehat{\mathbf{x}}$ est un *minimiseur local*, et $J(\widehat{\mathbf{x}})$ un *minimum local*.

*Remarque 8.1.* Bien que ceci ne soit pas toujours fait dans la littérature, il est éclairant de distinguer les minima des minimiseurs (et les maxima des maximiseurs). □

Dans la figure 8.1, $x_1$ et $x_2$ sont tous les deux des minimiseurs globaux, associés à l'unique minimum global $J_1$, tandis que $x_3$ n'est qu'un minimiseur local, puisque le minimum local $J_3$ est plus grand que $J_1$.

Idéalement, on voudrait trouver tous les minimiseurs globaux et le minimum global correspondant. En pratique, cependant, prouver qu'un minimiseur donné est global est souvent impossible. Trouver un minimiseur local peut déjà considérablement améliorer les performances par rapport à la situation initiale.

Les problèmes d'optimisation peuvent être classés suivant le type de leur domaine admissible $\mathbb{X}$ :

— $\mathbb{X} = \mathbb{R}^n$ correspond à de l'*optimisation continue sans contrainte* (chapitre 9).

— $\mathbb{X} \subsetneq \mathbb{R}^n$ correspond à de l'*optimisation sous contrainte(s)* (chapitre 10). Les contraintes expriment que certaines valeurs des variables de décision ne sont pas acceptables (par exemple que certaines variables doivent être positives). Nous distinguons les *contraintes d'égalité*

$$c_j^{\mathrm{e}}(\mathbf{x}) = 0, \quad j = 1, \cdots, n_{\mathrm{e}}, \tag{8.7}$$

et les *contraintes d'inégalité*

$$c_j^{\mathrm{i}}(\mathbf{x}) \leqslant 0, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{8.8}$$

Plus concisément, nous écrivons

$$\mathbf{c}^{\mathrm{e}}(\mathbf{x}) = \mathbf{0} \tag{8.9}$$

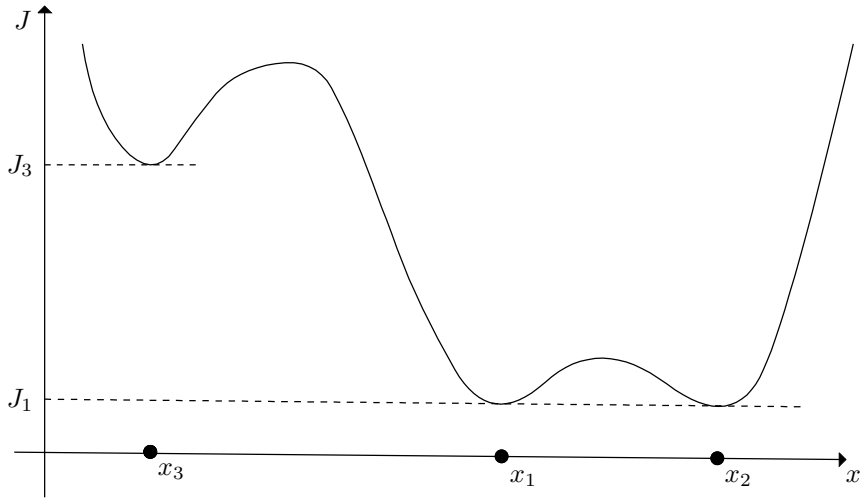


**Fig. 8.1** Minima et minimiseurs

et

$$\mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant \mathbf{0}, \tag{8.10}$$

à interpréter comme valide composante par composante.

— Quand $\mathbb{X}$ est fini et que les variables de décision ne sont pas quantitatives, on parle d'*optimisation combinatoire* (chapitre 11).
— Quand $\mathbb{X}$ est un ensemble de fonctions, on parle d'*optimisation fonctionnelle*, rencontrée par exemple en commande optimale [56] mais pas considérée dans ce livre.

*Remarque 8.2.* Rien n'interdit aux contraintes qui définissent $\mathbb{X}$ d'impliquer des quantités numériques calculées via un modèle à partir des valeurs numériques prises par les variables de décision. En commande optimale, par exemple, on peut exiger que l'état du système dynamique à commander satisfasse des contraintes d'inégalité à des instants donnés. □

*Remarque 8.3.* Chaque fois que possible, nous écrirons les contraintes d'inégalité sous la forme $c_j^{\mathrm{i}}(\mathbf{x}) \leqslant 0$ plutôt que sous la forme $c_j^{\mathrm{i}}(\mathbf{x}) < 0$, pour permettre à $\mathbb{X}$ d'être un ensemble fermé (c'est à dire qui contienne sa frontière). Quand $c_j^{\mathrm{i}}(\mathbf{x}) = 0$, la $j$-ème contrainte d'inégalité est dite *saturée* (ou *active*). □

*Remarque 8.4.* Quand $\mathbb{X}$ est tel que certains éléments $x_i$ du vecteur de décision $\mathbf{x}$ ne peuvent prendre que des valeurs entières et quand ces valeurs ont un sens quantitatif, on peut préférer parler de *programmation en nombres entiers* plutôt que d'optimisation combinatoire, bien que les deux termes soient parfois utilisés de façon

interchangeable. Un problème de programmation en nombres entiers peut être transformé en un problème d'optimisation continue sous contraintes. Si, par exemple, $\mathbb{X}$ est tel que $x_i \in \{0,1,2,3\}$, alors on peut imposer la contrainte

$$x_i(1-x_i)(2-x_i)(3-x_i) = 0. \tag{8.11}$$

□

*Remarque 8.5.* Le nombre $n = \dim \mathbf{x}$ des variables de décision a une forte influence sur la complexité du problème d'optimisation et sur les méthodes utilisables, à cause de ce qui est connu comme la *malédiction de la dimension*. Une méthode parfaitement viable pour $n = 2$ peut n'avoir aucune chance de succès pour $n = 50$, comme illustré par l'exemple suivant. □

*Exemple 8.6.* Soit $\mathbb{X}$ un hypercube unitaire à $n$ dimensions $[0,1] \times \cdots \times [0,1]$. Supposons une minimisation par recherche aléatoire, avec $\mathbf{x}^k$ $(k = 1, \cdots, N)$ tiré au hasard dans $\mathbb{X}$ suivant une loi uniforme et le vecteur de décision $\widehat{\mathbf{x}}^k$ associé au coût le plus bas obtenu jusqu'ici pris comme estimée d'un minimiseur global. La longueur du côté d'un hypercube $\mathbb{H}$ qui a une probabilité $p$ d'être atteint est $\alpha = p^{1/n}$, et cette longueur croît très vite avec $n$. Pour $p = 10^{-3}$, par exemple, $\alpha = 10^{-3}$ si $n = 1$, $\alpha \approx 0.5$ si $n = 10$ et $\alpha \approx 0.87$ si $n = 50$. Quand $n$ augmente, il devient donc très vite impossible d'explorer une petite région de l'espace de décision. Pour dire cela autrement, si l'on considère qu'il faut 100 points pour échantillonner l'intervalle $[0,1]$, alors il faudra $100^n$ points dans $\mathbb{X}$ pour obtenir une densité similaire. Heureusement, les régions vraiment intéressantes des espaces de décision de grande dimension correspondent souvent à des hypersurfaces de dimension plus basse qui peuvent encore être explorées efficacement, pourvu que des méthodes de recherche plus sophistiquées soient utilisées. □

Le type de la fonction de coût a aussi une forte influence sur le type de méthode de minimisation à utiliser.

— Quand $J(\mathbf{x})$ est *linéaire* en $\mathbf{x}$, on peut l'écrire

$$J(\mathbf{x}) = \mathbf{c}^{\mathrm{T}}\mathbf{x}. \tag{8.12}$$

Il *faut* alors introduire des contraintes pour éviter que $\mathbf{x}$ ne tende vers l'infini dans la direction $-\mathbf{c}$, ce qui n'aurait en général aucun sens. Si les contraintes sont linéaires (ou affines) en $\mathbf{x}$, alors le problème relève de la *programmation linéaire* (voir la section 10.6).

— Si $J(\mathbf{x})$ est *quadratique* en $\mathbf{x}$ et peut s'écrire

$$J(\mathbf{x}) = [\mathbf{A}\mathbf{x} - \mathbf{b}]^{\mathrm{T}}\mathbf{Q}[\mathbf{A}\mathbf{x} - \mathbf{b}], \tag{8.13}$$

où $\mathbf{A}$ est une matrice connue telle que $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ soit inversible, $\mathbf{Q}$ est une matrice de pondération symétrique définie positive connue et $\mathbf{b}$ est un vecteur connu, et si $\mathbb{X} = \mathbb{R}^n$, alors la méthode des *moindres carrés linéaires* peut être utilisée pour évaluer le minimiseur global unique de la fonction de coût (voir la section 9.2).

— Quand $J(\mathbf{x})$ est *non linéaire* en $\mathbf{x}$ (sans être quadratique), il faut distinguer deux cas.
  — Si $J(\mathbf{x})$ est *différentiable*, par exemple quand on minimise

$$J(\mathbf{x}) = \sum_{i=1}^{N} [e_i(\mathbf{x})]^2, \tag{8.14}$$

  avec $e_i(\mathbf{x})$ différentiable, alors on peut utiliser des développements en série de Taylor de la fonction de coût, ce qui conduit aux méthodes du gradient et de Newton et à leurs variantes (voir la section 9.3.4).
  — Si $J(\mathbf{x})$ n'est pas différentiable, par exemple quand on minimise

$$J(\mathbf{x}) = \sum_{i} |e_i(\mathbf{x})|, \tag{8.15}$$

  ou

$$J(\mathbf{x}) = \max_{\mathbf{v}} e(\mathbf{x}, \mathbf{v}), \tag{8.16}$$

  alors il faut faire appel à des méthodes spécifiques (voir les sections 9.3.5, 9.4.1.2 et 9.4.2.1). Même une fonction d'apparence aussi inoffensive que (8.15), qui est différentiable presque partout si les $e_i(\mathbf{x})$ le sont, ne peut être minimisée par une méthode itérative fondée sur un développement limité de la fonction de coût, car une telle méthode se ruera en général sur un point où la fonction de coût n'est pas différentiable pour y rester bloquée.
— Quand $J(\mathbf{x})$ est *convexe* sur $\mathbb{X}$, on peut exploiter la puissance des méthodes d'*optimisation convexe*, pourvu que $\mathbb{X}$ soit aussi convexe. Voir la section 10.7.

*Remarque 8.6.* Le temps nécessaire pour une évaluation de $J(\mathbf{x})$ a aussi des conséquences sur les types de méthodes employables. Quand chaque évaluation ne prend qu'une fraction de seconde, des algorithmes évolutionnaires ou par exploration aléatoire peuvent s'avérer viables. Tel n'est plus le cas quand chaque évaluation prend plusieurs heures, par exemple parce qu'elle implique la simulation d'un modèle complexe à base de connaissances, car le nombre d'évaluations de la fonction de coût est alors sévèrement limité, voir la section 9.4.3. □

## 8.4 Que diriez-vous d'un repas gratuit ?

Dans le contexte de l'optimisation, un repas gratuit (*free lunch*), ce serait une méthode universelle, capable de traiter n'importe quel problème d'optimisation, ce qui éliminerait le besoin de s'adapter aux spécificités du problème à traiter. Ceci aurait pu être le Saint Graal de l'optimisation évolutionnaire, si Wolpert et Macready n'avaient pas publié leurs théorèmes *no free lunch* (NFL).

### *8.4.1 Ça n'existe pas*

Les théorèmes NFL dans [254] (voir aussi [109]) reposent sur les hypothèses suivantes :

1. un oracle est disponible, qui retourne la valeur numérique de $J(\mathbf{x})$ pour n'importe quelle valeur numérique de $\mathbf{x}$ appartenant à $\mathbb{X}$,
2. l'espace de recherche $\mathbb{X}$ est fini,
3. la fonction de coût $J(\cdot)$ ne peut prendre qu'un nombre fini de valeurs numériques,
4. rien d'autre n'est connu de $J(\cdot)$ a priori,
5. Les algorithmes $\mathscr{A}_i$ en compétition sont déterministes,
6. les problèmes de minimisation $\mathscr{M}_j$ qui peuvent être générés sous les hypothèses 2 et 3 ont tous la même probabilité,
7. la performance $\mathscr{P}_N(\mathscr{A}_i, \mathscr{M}_j)$ de l'algorithme $\mathscr{A}_i$ sur le problème de minimisation $\mathscr{M}_j$ pour $N$ points $\mathbf{x}^k \in \mathbb{X}$ visités distincts et ordonnés dans le temps ne dépend que des valeurs prises par $\mathbf{x}^k$ et $J(\mathbf{x}^k)$, $k = 1, \cdots, N$.

Les hypothèses 2 et 3 sont toujours satisfaites quand on calcule avec des nombres à virgule flottante. Supposons, par exemple, qu'on utilise des flottants en double précision sur 64 bits. Alors

- le nombre représentant $J(\mathbf{x})$ ne peut pas prendre plus de $2^{64}$ valeurs,
- la représentation de $\mathbb{X}$ ne peut pas avoir plus de $(2^{64})^{\dim \mathbf{x}}$ éléments, avec $\dim \mathbf{x}$ le nombre de variables de décision.

Une borne supérieure du nombre $\sharp\mathscr{M}$ de problèmes de minimisation traitables est donc $(2^{64})^{\dim \mathbf{x}+1}$.

L'hypothèse 4 interdit d'exploiter des connaissances supplémentaires éventuelles sur le problème de minimisation à résoudre, comme le fait de savoir qu'il est convexe.

L'hypothèse 5 est satisfaite par toutes les méthodes de minimisation de type boîte noire usuelles comme le recuit simulé ou les algorithmes évolutionnaires, même s'ils semblent impliquer de l'aléatoire, puisque tout générateur de nombres pseudo-aléatoires est déterministe pour une graine donnée.

La mesure de performance peut être, par exemple, la meilleure valeur de la fonction de coût obtenue jusqu'ici

$$\mathscr{P}_N(\mathscr{A}_i, \mathscr{M}_j) = \min_{k=1}^{N} J(\mathbf{x}^k). \tag{8.17}$$

Notons que le temps nécessaire à l'algorithme pour visiter $N$ points distincts dans $\mathbb{X}$ ne peut pas être pris en compte dans la mesure de performance.

Nous ne considérons que le premier des théorèmes NFL de [254], qui peut être résumé ainsi : pour toute paire d'algorithmes $(\mathscr{A}_1, \mathscr{A}_2)$, la performance moyenne sur tous les problèmes de minimisation est *la même*, c'est à dire que

$$\frac{1}{\sharp\mathscr{M}}\sum_{j=1}^{\sharp\mathscr{M}}\mathscr{P}_N(\mathscr{A}_1,\mathscr{M}_j)=\frac{1}{\sharp\mathscr{M}}\sum_{j=1}^{\sharp\mathscr{M}}\mathscr{P}_N(\mathscr{A}_2,\mathscr{M}_j). \tag{8.18}$$

En d'autres termes, si $\mathscr{A}_1$ a de meilleures performances que $\mathscr{A}_2$ en moyenne sur une série donnée de problèmes de minimisation, alors $\mathscr{A}_2$ *doit* avoir de meilleures performances que $\mathscr{A}_1$ en moyenne sur *tous* les autres...

*Exemple 8.7.* Soit $\mathscr{A}_1$ un algorithme de *descente*, qui sélectionne parmi les voisins de $\mathbf{x}^k$ dans $\mathbb{X}$ l'un de ceux au coût le plus bas pour en faire $\mathbf{x}^{k+1}$. Soit $\mathscr{A}_2$ un algorithme de *montée* qui sélectionne à la place l'un des voisins au coût le plus haut, et soit $\mathscr{A}_3$ un algorithme qui tire $\mathbf{x}^k$ au hasard dans $\mathbb{X}$. Mesurons les performances par le plus petit coût atteint après l'exploration de $N$ points distincts dans $\mathbb{X}$. La performance moyenne de ces trois algorithmes est la même. En d'autres termes, *l'algorithme n'a pas d'importance en moyenne*, et le fait de montrer qu'$\mathscr{A}_1$ a de meilleures performances qu'$\mathscr{A}_2$ or $\mathscr{A}_3$ sur quelques cas tests ne peut contredire ce fait troublant. □

### *8.4.2 Vous pouvez quand même obtenir un repas bon marché*

Aucun algorithme ne peut donc prétendre être meilleur que les autres en termes de performances moyennes sur tous les types de problèmes. Pire, on peut prouver avec des arguments de complexité que *l'optimisation globale est impossible* dans le cas le plus général [169].

Il faut tout de même noter que la plupart des $\sharp\mathscr{M}$ problèmes de minimisation sur lesquels les performances moyennes sont calculées par (8.18) n'ont *aucun intérêt* du point de vue des applications. On a en général affaire à des classes spécifiques de problèmes de minimisation, pour lesquels certains algorithmes sont en effet meilleurs que d'autres. Quand la classe des problèmes à considérer est réduite, même légèrement, certains algorithmes évolutionnaires peuvent être préférables à d'autres, comme le montre [60] sur un exemple jouet. Des restrictions supplémentaires, comme de requérir que $J(\cdot)$ soit convexe, peuvent être jugées plus coûteuses mais permettent d'utiliser des algorithmes beaucoup plus puissants.

L'optimisation continue sans contrainte sera considérée en premier, au chapitre 9.

## 8.5 En résumé

— Avant d'entreprendre une optimisation, vérifiez que cela fait sens pour le problème qu'il s'agit de traiter.
— On peut toujours transformer un problème de maximisation en un problème de minimisation, de sorte qu'il n'est pas restrictif de ne considérer que des minimisations.

— Il est utile de distinguer *minima* et *minimiseurs*.
— On peut classifier les problèmes d'optimisation suivant le type du domaine admissible $\mathbb{X}$ pour leurs variables de décision.
— Le type de la fonction de coût a une influence forte sur les classes de méthodes d'optimisation utilisables. Les fonctions de coût non différentiables ne peuvent être minimisées en utilisant des méthodes fondées sur un développement de Taylor de la fonction de coût.
— La dimension de vecteur de décision $\mathbf{x}$ est un facteur clé à prendre en compte dans le choix d'un algorithme, à cause de la malédiction de la dimension.
— Le temps nécessaire pour une évaluation de la fonction de coût doit aussi être pris en considération.
— Il n'y a pas de repas gratuit.

# Chapitre 9
# Optimiser sans contrainte

Dans ce chapitre, le vecteur de décision $\mathbf{x}$ est supposé appartenir à $\mathbb{R}^n$. Il n'y a pas de contrainte d'égalité, et les contraintes d'inégalité éventuelles sont supposées ne pas être saturées en tout minimiseur, de sorte qu'elles pourraient aussi bien ne pas exister (sauf peut-être pour empêcher temporairement $\mathbf{x}$ de s'aventurer en terrain inconnu).

## 9.1 Conditions théoriques d'optimalité

Les conditions d'optimalité présentées ici ont inspiré des algorithmes et des conditions d'arrêt utiles. Supposons la fonction de coût $J(\cdot)$ différentiable, et écrivons son développement de Taylor au premier ordre au voisinage d'un minimiseur $\widehat{\mathbf{x}}$

$$J(\widehat{\mathbf{x}}+\delta\mathbf{x}) = J(\widehat{\mathbf{x}}) + \sum_{i=1}^{n} \frac{\partial J}{\partial x_i}(\widehat{\mathbf{x}})\delta x_i + \mathrm{o}(||\delta\mathbf{x}||), \tag{9.1}$$

ou, de façon plus concise,

$$J(\widehat{\mathbf{x}}+\delta\mathbf{x}) = J(\widehat{\mathbf{x}}) + \mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\delta\mathbf{x} + \mathrm{o}(||\delta\mathbf{x}||), \tag{9.2}$$

avec $\mathbf{g}(\mathbf{x})$ le *gradient* de la fonction de coût évalué en $\mathbf{x}$

$$\mathbf{g}(\mathbf{x}) = \frac{\partial J}{\partial \mathbf{x}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_n} \end{bmatrix} (\mathbf{x}). \tag{9.3}$$

*Exemple 9.1. Analogie topographique*

Si $J(\mathbf{x})$ est l'altitude au point $\mathbf{x}$, avec $x_1$ sa latitude et $x_2$ sa longitude, alors $\mathbf{g}(\mathbf{x})$ est la direction de montée la plus raide, c'est à dire la direction dans laquelle l'altitude monte le plus rapidement quand on quitte $\mathbf{x}$. □

Pour que $\widehat{\mathbf{x}}$ soit un minimiseur de $J(\cdot)$ (au moins localement), il faut que le terme du premier ordre en $\delta\mathbf{x}$ ne contribue jamais à faire décroître le coût. Il doit donc satisfaire

$$\mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\delta\mathbf{x} \geqslant 0 \quad \forall \delta\mathbf{x} \in \mathbb{R}^n. \tag{9.4}$$

L'équation (9.4) doit rester vraie quand $\delta\mathbf{x}$ est remplacé par $-\delta\mathbf{x}$, il faut donc que

$$\mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\delta\mathbf{x} = 0 \quad \forall \delta\mathbf{x} \in \mathbb{R}^n. \tag{9.5}$$

Comme il n'y a pas de contrainte sur $\delta\mathbf{x}$, ceci n'est possible que si le gradient du coût en $\widehat{\mathbf{x}}$ est nul. Une condition *nécessaire* d'optimalité au premier ordre est donc

$$\mathbf{g}(\widehat{\mathbf{x}}) = \mathbf{0}. \tag{9.6}$$

Cette *condition de stationnarité* ne suffit pas à garantir que $\widehat{\mathbf{x}}$ soit un minimiseur, même localement. Il peut tout aussi bien s'agir d'un maximiseur local (figure 9.1) ou d'un *point en selle*, c'est à dire d'un point à partir duquel le coût augmente dans certaines directions et diminue dans d'autres. Si une fonction de coût différentiable n'a pas de point stationnaire, alors le problème d'optimisation associé n'a pas de sens en l'absence de contrainte.

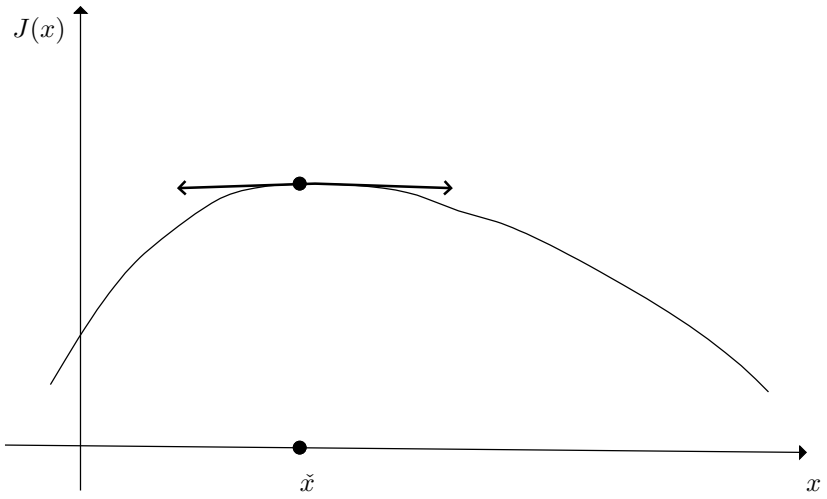


**Fig. 9.1** Le point stationnaire $\check{x}$ est un maximiseur

Considérons maintenant le développement de Taylor au second ordre de la fonction de coût au voisinage de $\widehat{\mathbf{x}}$

$$J(\widehat{\mathbf{x}}+\delta\mathbf{x}) = J(\widehat{\mathbf{x}}) + \mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\delta\mathbf{x} + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{\partial^2 J}{\partial x_i \partial x_j}(\widehat{\mathbf{x}})\delta x_i \delta x_j + \mathrm{o}(||\delta\mathbf{x}||^2), \tag{9.7}$$

ou, de façon plus concise,

$$J(\widehat{\mathbf{x}}+\delta\mathbf{x}) = J(\widehat{\mathbf{x}}) + \mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^{\mathrm{T}}\mathbf{H}(\widehat{\mathbf{x}})\delta\mathbf{x} + \mathrm{o}(||\delta\mathbf{x}||^2), \tag{9.8}$$

où $\mathbf{H}(\mathbf{x})$ est la *hessienne* de la fonction de coût évalué en $\mathbf{x}$

$$\mathbf{H}(\mathbf{x}) = \frac{\partial^2 J}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}). \tag{9.9}$$

C'est une matrice symétrique, dont l'élément en position $(i, j)$ est donné par

$$h_{i,j}(\mathbf{x}) = \frac{\partial^2 J}{\partial x_i \partial x_j}(\mathbf{x}). \tag{9.10}$$

Si la condition nécessaire d'optimalité au premier ordre (9.6) est satisfaite, alors

$$J(\widehat{\mathbf{x}}+\delta\mathbf{x}) = J(\widehat{\mathbf{x}}) + \frac{1}{2}\delta\mathbf{x}^{\mathrm{T}}\mathbf{H}(\widehat{\mathbf{x}})\delta\mathbf{x} + \mathrm{o}(||\delta\mathbf{x}||^2), \tag{9.11}$$

et le terme du second ordre en $\delta\mathbf{x}$ ne doit jamais contribuer à faire décroître le coût. Une condition *nécessaire* d'optimalité au second ordre est donc

$$\delta\mathbf{x}^{\mathrm{T}}\mathbf{H}(\widehat{\mathbf{x}})\delta\mathbf{x} \geqslant 0 \quad \forall\delta\mathbf{x}, \tag{9.12}$$

de sorte que toutes les valeurs propres de $\mathbf{H}(\widehat{\mathbf{x}})$ doivent être positives ou nulles. Ceci revient à dire que $\mathbf{H}(\widehat{\mathbf{x}})$ doit être symétrique définie non-négative, ce qu'on note

$$\mathbf{H}(\widehat{\mathbf{x}}) \succcurlyeq 0. \tag{9.13}$$

Ensemble, (9.6) et (9.13) ne forment pas une condition suffisante d'optimalité, même localement, car les valeurs propres nulles de $\mathbf{H}(\widehat{\mathbf{x}})$ sont associées à des vecteurs propres dans la direction desquels il est possible de s'éloigner de $\widehat{\mathbf{x}}$ sans faire croître la contribution au coût du terme du second ordre. Il faudrait alors considérer des termes d'ordre plus élevé pour conclure. Pour prouver, par exemple, que $J(x) = x^{1000}$ a un minimiseur local en $\widehat{x} = 0$ via un développement de Taylor, il faudrait calculer toutes les dérivées de cette fonction de coût jusqu'à l'ordre 1000, car toutes les dérivées d'ordre inférieur sont nulles en $\widehat{x}$.

La condition plus restrictive

$$\delta\mathbf{x}^{\mathrm{T}}\mathbf{H}(\widehat{\mathbf{x}})\delta\mathbf{x} > 0 \quad \forall\delta\mathbf{x}, \tag{9.14}$$

qui impose que toutes les valeurs propres de $\mathbf{H}(\widehat{\mathbf{x}})$ soient strictement positives, fournit une condition *suffisante* d'optimalité *locale* au second ordre (pourvu que la condition nécessaire du premier ordre (9.6) soit aussi satisfaite). Elle équivaut à dire que $\mathbf{H}(\widehat{\mathbf{x}})$ est symétrique définie positive, ce qu'on note

$$\mathbf{H}(\widehat{\mathbf{x}}) \succ 0. \tag{9.15}$$

En résumé, une condition *nécessaire* d'optimalité de $\widehat{\mathbf{x}}$ est

$$\mathbf{g}(\widehat{\mathbf{x}}) = \mathbf{0} \quad \text{et} \quad \mathbf{H}(\widehat{\mathbf{x}}) \succcurlyeq 0, \tag{9.16}$$

et une condition *suffisante* d'optimalité *locale* de $\widehat{\mathbf{x}}$ est

$$\mathbf{g}(\widehat{\mathbf{x}}) = \mathbf{0} \quad \text{et} \quad \mathbf{H}(\widehat{\mathbf{x}}) \succ 0. \tag{9.17}$$

*Remarque 9.1.* Il n'y a *pas*, en général, de condition nécessaire et suffisante d'optimalité, même locale. □

*Remarque 9.2.* Quand on ne sait rien d'autre de la fonction de coût, la satisfaction de (9.17) ne garantit pas que $\widehat{\mathbf{x}}$ soit un minimiseur global. □

*Remarque 9.3.* Les conditions sur la hessienne ne sont valides que pour une minimisation. Pour une maximisation, il faut y remplacer $\succcurlyeq$ par $\preccurlyeq$, et $\succ$ par $\prec$. □

*Remarque 9.4.* Comme le suggère (9.6), des méthodes de résolution de systèmes d'équations vues au chapitre 3 (pour les systèmes linéaires) et au chapitre 7 (pour les systèmes non linéaires) peuvent aussi être utilisées pour rechercher des minimiseurs. On peut alors exploiter les propriétés spécifiques de la jacobienne de la fonction gradient (c'est à dire de la hessienne), dont (9.13) nous dit qu'elle doit être symétrique définie non-négative en tout minimiseur local ou global. □

*Exemple 9.2. Retour sur le krigeage*

On peut établir les équations (5.61) et (5.64) du prédicteur par krigeage grâce aux conditions d'optimalité théoriques (9.6) et (9.15). Supposons, comme en section 5.4.3, que $N$ mesures aient été effectuées pour obtenir

$$y_i = f(\mathbf{x}^i), \quad i = 1, \cdots, N. \tag{9.18}$$

Dans sa version la plus simple, le krigeage interprète ces résultats comme des réalisations d'un processus gaussien à moyenne nulle $Y(\mathbf{x})$. On a donc

$$\forall \mathbf{x}, \quad E\{Y(\mathbf{x})\} = 0 \tag{9.19}$$

et

$$\forall \mathbf{x}^i, \forall \mathbf{x}^j, \quad E\{Y(\mathbf{x}^i)Y(\mathbf{x}^j)\} = \sigma_y^2 r(\mathbf{x}^i, \mathbf{x}^j), \tag{9.20}$$

où $r(\cdot,\cdot)$ est une fonction de corrélation, telle que $r(\mathbf{x},\mathbf{x}) = 1$, et où $\sigma_y^2$ est la variance du processus gaussien. Soit $\widehat{Y}(\mathbf{x})$ une combinaison linéaire des $Y(\mathbf{x}^i)$, de sorte que

$$\widehat{Y}(\mathbf{x}) = \mathbf{c}^{\mathrm{T}}(\mathbf{x})\mathbf{Y}, \tag{9.21}$$

où $\mathbf{Y}$ est le *vecteur* aléatoire

$$\mathbf{Y} = [Y(\mathbf{x}^1), Y(\mathbf{x}^2), \cdots, Y(\mathbf{x}^N)]^{\mathrm{T}} \tag{9.22}$$

et où $\mathbf{c}(\mathbf{x})$ est un vecteur de poids. $\widehat{Y}(\mathbf{x})$ est un prédicteur *non biaisé* de $Y(\mathbf{x})$, puisque pour tout $\mathbf{x}$

$$E\{\widehat{Y}(\mathbf{x}) - Y(\mathbf{x})\} = E\{\widehat{Y}(\mathbf{x})\} - E\{Y(\mathbf{x})\} = \mathbf{c}^{\mathrm{T}}(\mathbf{x})E\{\mathbf{Y}\} = 0. \tag{9.23}$$

Il n'y a donc pas d'erreur systématique quel que soit le vecteur de poids $\mathbf{c}(\mathbf{x})$. Le *meilleur prédicteur linéaire non biaisé* de $Y(\mathbf{x})$ choisit $\mathbf{c}(\mathbf{x})$ pour minimiser la variance de l'erreur de prédiction en $\mathbf{x}$. Comme

$$[\widehat{Y}(\mathbf{x}) - Y(\mathbf{x})]^2 = \mathbf{c}^{\mathrm{T}}(\mathbf{x})\mathbf{Y}\mathbf{Y}^{\mathrm{T}}\mathbf{c}(\mathbf{x}) + [Y(\mathbf{x})]^2 - 2\mathbf{c}^{\mathrm{T}}(\mathbf{x})\mathbf{Y}Y(\mathbf{x}), \tag{9.24}$$

la variance de l'erreur de prédiction vaut

$$\begin{aligned} E\{[\widehat{Y}(\mathbf{x}) - Y(\mathbf{x})]^2\} &= \mathbf{c}^{\mathrm{T}}(\mathbf{x})E\left\{\mathbf{Y}\mathbf{Y}^{\mathrm{T}}\right\}\mathbf{c}(\mathbf{x}) + \sigma_y^2 - 2\mathbf{c}^{\mathrm{T}}(\mathbf{x})E\{\mathbf{Y}Y(\mathbf{x})\} \\ &= \sigma_y^2\left[\mathbf{c}^{\mathrm{T}}(\mathbf{x})\mathbf{R}\mathbf{c}(\mathbf{x}) + 1 - 2\mathbf{c}^{\mathrm{T}}(\mathbf{x})\mathbf{r}(\mathbf{x})\right], \end{aligned} \tag{9.25}$$

où $\mathbf{R}$ et $\mathbf{r}(\mathbf{x})$ sont définis par (5.62) et (5.63). La minimisation de cette variance par rapport à $\mathbf{c}$ est donc équivalente à la minimisation de

$$J(\mathbf{c}) = \mathbf{c}^{\mathrm{T}}\mathbf{R}\mathbf{c} + 1 - 2\mathbf{c}^{\mathrm{T}}\mathbf{r}(\mathbf{x}). \tag{9.26}$$

La condition d'optimalité au premier ordre (9.6) se traduit par

$$\frac{\partial J}{\partial \mathbf{c}}(\widehat{\mathbf{c}}) = 2\mathbf{R}\widehat{\mathbf{c}} - 2\mathbf{r}(\mathbf{x}) = \mathbf{0}. \tag{9.27}$$

Pourvu que $\mathbf{R}$ soit inversible, comme elle devrait l'être, (9.27) implique que le vecteur de pondération optimal est

$$\widehat{\mathbf{c}}(\mathbf{x}) = \mathbf{R}^{-1}\mathbf{r}(\mathbf{x}). \tag{9.28}$$

Comme $\mathbf{R}$ est symétrique, (9.21) et (9.28) impliquent que

$$\widehat{Y}(\mathbf{x}) = \mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{Y}. \tag{9.29}$$

La moyenne prédite sur la base des données $\mathbf{y}$ est ainsi

$$\widehat{y}(\mathbf{x}) = \mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{y}, \tag{9.30}$$

qui est (5.61). Remplaçons dans (9.25) $\mathbf{c}(\mathbf{x})$ par sa valeur optimale $\widehat{\mathbf{c}}(\mathbf{x})$ pour obtenir la variance (optimale) de la prédiction

$$\begin{aligned}\widehat{\sigma}^2(\mathbf{x}) &= \sigma_y^2\left[\mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{R}\mathbf{R}^{-1}\mathbf{r}(\mathbf{x})+1-2\mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{r}(\mathbf{x})\right]\\ &= \sigma_y^2\left[1-\mathbf{r}^{\mathrm{T}}(\mathbf{x})\mathbf{R}^{-1}\mathbf{r}(\mathbf{x})\right], \qquad (9.31)\end{aligned}$$

qui est (5.64).

La condition (9.17) est satisfaite, pourvu que

$$\frac{\partial^2 J}{\partial \mathbf{c}\partial \mathbf{c}^{\mathrm{T}}}(\widehat{\mathbf{c}}) = 2\mathbf{R} \succ 0. \qquad (9.32)$$

□

*Remarque 9.5.* L'exemple 9.2 néglige le fait que $\sigma_y^2$ est inconnu et que la fonction de corrélation $r(\mathbf{x}^i,\mathbf{x}^j)$ implique souvent un vecteur $\mathbf{p}$ de paramètres à estimer à partir des données, de sorte que $\mathbf{R}$ et $\mathbf{r}(\mathbf{x})$ devrait en fait s'écrire $\mathbf{R}(\mathbf{p})$ et $\mathbf{r}(\mathbf{x},\mathbf{p})$. L'approche la plus courante pour estimer $\mathbf{p}$ et $\sigma_y^2$ est celle du *maximum de vraisemblance*. La densité de probabilité du vecteur des données $\mathbf{y}$ est alors maximisée sous l'hypothèse que ce vecteur a été généré par un modèle de paramètres $\mathbf{p}$ et $\sigma_y^2$. Les estimées au sens du maximum de vraisemblance de $\mathbf{p}$ et $\sigma_y^2$ sont ainsi obtenues en solvant un autre problème d'optimisation, comme

$$\widehat{\mathbf{p}} = \arg\min_{\mathbf{p}}\left[N\ln\left(\frac{\mathbf{y}^{\mathrm{T}}\mathbf{R}^{-1}(\mathbf{p})\mathbf{y}}{N}\right)+\ln\det\mathbf{R}(\mathbf{p})\right] \qquad (9.33)$$

et

$$\widehat{\sigma}_y^2 = \frac{\mathbf{y}^{\mathrm{T}}\mathbf{R}^{-1}(\widehat{\mathbf{p}})\mathbf{y}}{N}. \qquad (9.34)$$

En remplaçant dans (5.61) et dans (5.64) $\mathbf{R}$ par $\mathbf{R}(\widehat{\mathbf{p}})$, $\mathbf{r}(\mathbf{x})$ par $\mathbf{r}(\mathbf{x},\widehat{\mathbf{p}})$ et $\sigma_y^2$ par $\widehat{\sigma}_y^2$, on obtient un meilleur estimateur linéaire non biaisé empirique [205]. □

## 9.2 Moindres carrés linéaires

Les conditions théoriques d'optimalité (9.6) et (9.15) trouvent une autre application directe avec la méthode des moindres carrés linéaires [139, 18], pour laquelle elles fournissent une solution optimale explicite. Cette méthode s'applique dans le cas particulier important où *la fonction de coût est quadratique en le vecteur de décision* $\mathbf{x}$. (L'exemple 9.2 illustrait déjà ce cas particulier, avec $\mathbf{c}$ le vecteur de décision.) La fonction de coût est maintenant supposée quadratique en une erreur elle-même affine en $\mathbf{x}$.

### *9.2.1 Coût quadratique en l'erreur*

Soit $\mathbf{y}$ un vecteur de données numériques et $\mathbf{f}(\mathbf{x})$ la sortie d'un modèle de ces données, où $\mathbf{x}$ est un vecteur de paramètres à estimer (les variables de décision). Il y a d'habitude plus de données que de paramètres, de sorte que

$$N = \dim\ \mathbf{y} = \dim\ \mathbf{f}(\mathbf{x}) > n = \dim\ \mathbf{x}. \tag{9.35}$$

Il n'y a donc en général pas de solution en $\mathbf{x}$ du système d'équations

$$\mathbf{y} = \mathbf{f}(\mathbf{x}). \tag{9.36}$$

Il faut alors remplacer l'interpolation des données par leur approximation. Définissons l'erreur comme le vecteur des *résidus*

$$\mathbf{e}(\mathbf{x}) = \mathbf{y} - \mathbf{f}(\mathbf{x}). \tag{9.37}$$

La stratégie la plus communément utilisée pour estimer $\mathbf{x}$ à partir des données est de minimiser une fonction de coût quadratique en $\mathbf{e}(\mathbf{x})$, telle que

$$J(\mathbf{x}) = \mathbf{e}^{\mathrm{T}}(\mathbf{x})\mathbf{W}\mathbf{e}(\mathbf{x}), \tag{9.38}$$

où $\mathbf{W} \succ 0$ est une matrice de pondération *connue*, choisie par l'utilisateur. L'*estimée de* $\mathbf{x}$ *au sens des moindres carrés pondérés* est alors

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}\in\mathbb{R}^n} [\mathbf{y} - \mathbf{f}(\mathbf{x})]^{\mathrm{T}}\mathbf{W}[\mathbf{y} - \mathbf{f}(\mathbf{x})]. \tag{9.39}$$

On peut toujours calculer, par exemple avec la méthode de Cholesky vue en section 3.8.1, une matrice $\mathbf{M}$ telle que

$$\mathbf{W} = \mathbf{M}^{\mathrm{T}}\mathbf{M}, \tag{9.40}$$

de sorte que

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}\in\mathbb{R}^n} [\mathbf{M}\mathbf{y} - \mathbf{M}\mathbf{f}(\mathbf{x})]^{\mathrm{T}}[\mathbf{M}\mathbf{y} - \mathbf{M}\mathbf{f}(\mathbf{x})]. \tag{9.41}$$

En posant $\mathbf{y}' = \mathbf{M}\mathbf{y}$ et $\mathbf{f}'(\mathbf{x}) = \mathbf{M}\mathbf{f}(\mathbf{x})$, on transforme le problème initial en un problème de *moindres carrés non pondérés* :

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}\in\mathbb{R}^n} J'(\mathbf{x}), \tag{9.42}$$

où

$$J'(\mathbf{x}) = ||\mathbf{y}' - \mathbf{f}'(\mathbf{x})||_2^2, \tag{9.43}$$

avec $||\cdot||_2^2$ le carré de la norme $l_2$. Sauf quand $\mathbf{W}$ est déjà la matrice identité $N \times N$, nous supposerons dans ce qui suit que cette transformation a été faite, mais omettrons les signes prime pour simplifier les notations.

### 9.2.2 Coût quadratique en les variables de décision

Quand $\mathbf{f}(\cdot)$ est linéaire en ses arguments, on peut écrire

$$\mathbf{f}(\mathbf{x}) = \mathbf{F}\mathbf{x}, \tag{9.44}$$

où $\mathbf{F}$ est une *matrice de régression $N \times n$ connue*. L'erreur

$$\mathbf{e}(\mathbf{x}) = \mathbf{y} - \mathbf{F}\mathbf{x} \tag{9.45}$$

est donc affine en $\mathbf{x}$. Ceci implique que la fonction de coût (9.43) est quadratique en $\mathbf{x}$ :

$$J(\mathbf{x}) = ||\mathbf{y} - \mathbf{F}\mathbf{x}||_2^2 = (\mathbf{y} - \mathbf{F}\mathbf{x})^{\mathrm{T}}(\mathbf{y} - \mathbf{F}\mathbf{x}). \tag{9.46}$$

La condition nécessaire d'optimalité au premier ordre (9.6) requiert que le gradient de $J(\cdot)$ en $\widehat{\mathbf{x}}$ soit nul. Puisque (9.46) est quadratique en $\mathbf{x}$, le gradient de la fonction de coût est affine en $\mathbf{x}$, et donné par

$$\frac{\partial J}{\partial \mathbf{x}}(\mathbf{x}) = -2\mathbf{F}^{\mathrm{T}}(\mathbf{y} - \mathbf{F}\mathbf{x}) = -2\mathbf{F}^{\mathrm{T}}\mathbf{y} + 2\mathbf{F}^{\mathrm{T}}\mathbf{F}\mathbf{x}. \tag{9.47}$$

Supposons pour le moment $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ inversible. Ceci est vrai si et seulement si toutes les colonnes de $\mathbf{F}$ sont linéairement indépendantes, et implique que $\mathbf{F}^{\mathrm{T}}\mathbf{F} \succ 0$. La condition nécessaire d'optimalité au premier ordre

$$\frac{\partial J}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) = \mathbf{0} \tag{9.48}$$

se traduit alors par la célèbre *formule des moindres carrés*

$$\widehat{\mathbf{x}} = \left(\mathbf{F}^{\mathrm{T}}\mathbf{F}\right)^{-1}\mathbf{F}^{\mathrm{T}}\mathbf{y}, \tag{9.49}$$

qui donne explicitement le point stationnaire *unique* de la fonction de coût. De plus, puisque $\mathbf{F}^{\mathrm{T}}\mathbf{F} \succ 0$, la condition suffisante d'optimalité locale (9.17) est satisfaite et (9.49) donne explicitement le minimiseur *global unique* de la fonction de coût. C'est un avantage considérable sur le cas général, où une telle solution explicite n'existe pas.

*Exemple 9.3. Régression polynomiale*

Soit $y_i$ la valeur d'une quantité d'intérêt mesurée à l'instant connu $t_i$ $(i = 1, \cdots, N)$. Supposons que ces données soient à approximer par un polynôme d'ordre $k$ sous forme de série de puissances

$$P_k(t, \mathbf{x}) = \sum_{i=0}^{k} p_i t^i, \tag{9.50}$$

où

$$\mathbf{x} = (p_0 \quad p_1 \quad \cdots \quad p_k)^{\mathrm{T}}. \tag{9.51}$$

Supposons aussi qu'il y ait plus de données que de paramètres ($N > n = k+1$). Pour calculer l'estimée $\widehat{\mathbf{x}}$ du vecteur des paramètres, on peut chercher la valeur de $\mathbf{x}$ qui minimise

$$J(\mathbf{x}) = \sum_{i=1}^{N} [y_i - P_k(t_i, \mathbf{x})]^2 = ||\mathbf{y} - \mathbf{F}\mathbf{x}||_2^2, \tag{9.52}$$

avec

$$\mathbf{y} = [y_1 \quad y_2 \quad \cdots \quad y_N]^{\mathrm{T}} \tag{9.53}$$

et

$$\mathbf{F} = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^k \\ 1 & t_2 & t_2^2 & \cdots & t_2^k \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & t_N & t_N^2 & \cdots & t_N^k \end{bmatrix}. \tag{9.54}$$

*Mathématiquement*, la solution optimale est alors donnée par (9.49). □

*Remarque 9.6.* Le point clé de l'exemple 9.3 est que la sortie $P_k(t, \mathbf{x})$ du modèle est linéaire en $\mathbf{x}$. Ainsi, par exemple, la fonction

$$f(t, \mathbf{x}) = x_1 \mathrm{e}^{-t} + x_2 t^2 + \frac{x_3}{t} \tag{9.55}$$

pourrait bénéficier d'un traitement similaire. □

Malgré sa concision élégante, (9.49) ne devrait être utilisée que *rarement* pour calculer des estimées au sens des moindres carrés, et ce pour au moins deux raisons.

D'abord, l'inversion de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ requiert en général des calculs inutiles, et il est alors moins coûteux de résoudre le système d'équations linéaires

$$\mathbf{F}^{\mathrm{T}}\mathbf{F}\widehat{\mathbf{x}} = \mathbf{F}^{\mathrm{T}}\mathbf{y}, \tag{9.56}$$

qu'on appelle les *équations normales*. Comme $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ est supposée définie positive pour le moment, on peut utiliser une factorisation de Cholesky pour ce faire. La résolution des équations normales est l'approche la plus économique, qui ne doit être utilisée que sur des problèmes bien conditionnés.

Ensuite, le conditionnement de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ est presque toujours beaucoup plus mauvais que celui de $\mathbf{F}$, pour des raisons expliquées en section 9.2.4. Ceci suggère d'utiliser des méthodes telles que celles présentées dans les deux sections qui suivent, qui évitent le calcul de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$.

Parfois, cependant, $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ prend une forme diagonale particulièrement simple. Ceci peut être dû à de la planification d'expériences, comme dans l'exemple 9.4,

où à un choix adapté de la représentation du modèle, comme dans l'exemple 9.5. La résolution des équations normales (9.56) devient alors triviale et il n'y a plus de raison de l'éviter.

*Exemple 9.4. Plan d'expériences factoriel pour un modèle quadratique*

Supposons qu'une quantité d'intérêt $y(\mathbf{u})$ soit modélisée comme

$$y_{\mathrm{m}}(\mathbf{u},\mathbf{x}) = p_0 + p_1 u_1 + p_2 u_2 + p_3 u_1 u_2, \tag{9.57}$$

où $u_1$ et $u_2$ sont des facteurs d'entrée, dont les valeurs peuvent être choisies librement dans l'intervalle normalisé $[-1,1]$ et où

$$\mathbf{x} = (p_0, \cdots, p_3)^{\mathrm{T}}. \tag{9.58}$$

Les paramètres $p_1$ et $p_2$ quantifient respectivement les effets de $u_1$ et $u_2$ seuls, tandis que $p_3$ quantifie l'effet de l'interaction de $u_1$ et $u_2$. Notons qu'il n'y a pas de terme en $u_1^2$ ou $u_2^2$. Le vecteur $\mathbf{x}$ des paramètres est à estimer à partir des données expérimentales $y(\mathbf{u}^i)$, $i = 1, \cdots, N$, en minimisant

$$J(\mathbf{x}) = \sum_{i=1}^{N} \left[ y(\mathbf{u}^i) - y_{\mathrm{m}}(\mathbf{u}^i, \mathbf{x}) \right]^2. \tag{9.59}$$

Un *plan factoriel complet* à deux niveaux requiert de collecter des données à toutes les combinaisons possibles des valeurs extrêmes $\{-1,1\}$ des facteurs, comme dans le tableau 9.1, et ce schéma de collecte peut être répété pour diminuer l'influence du bruit de mesure. Supposons qu'on le répète une fois, de sorte que $N = 8$. Les éléments de la matrice de régression $\mathbf{F}$ résultante (de dimensions $8 \times 4$) sont alors ceux du tableau 9.2 privé de sa première ligne et de sa première colonne.

**Tableau 9.1** Plan factoriel complet à deux niveaux

| Numéro de l'expérience | Valeur de $u_1$ | Valeur de $u_2$ |
|---|---|---|
| 1 | $-1$ | $-1$ |
| 2 | $-1$ | 1 |
| 3 | 1 | $-1$ |
| 4 | 1 | 1 |

Il est trivial de vérifier que

$$\mathbf{F}^{\mathrm{T}}\mathbf{F} = 8\mathbf{I}_4, \tag{9.60}$$

de sorte que $\mathrm{cond}(\mathbf{F}^{\mathrm{T}}\mathbf{F}) = 1$, et (9.56) implique que

$$\widehat{\mathbf{x}} = \frac{1}{8}\mathbf{F}^{\mathrm{T}}\mathbf{y}. \tag{9.61}$$

Cet exemple peut être généralisé très facilement à un nombre quelconque de facteurs d'entrée, pourvu que le modèle polynomial quadratique ne contienne aucun terme

**Tableau 9.2** Construction de **F**

| Numéro de l'expérience | Constante | Valeur de $u_1$ | Valeur de $u_2$ | Valeur de $u_1u_2$ |
|---|---|---|---|---|
| 1 | 1 | −1 | −1 | 1 |
| 2 | 1 | −1 | 1 | −1 |
| 3 | 1 | 1 | −1 | −1 |
| 4 | 1 | 1 | 1 | 1 |
| 5 | 1 | −1 | −1 | 1 |
| 6 | 1 | −1 | 1 | −1 |
| 7 | 1 | 1 | −1 | −1 |
| 8 | 1 | 1 | 1 | 1 |

quadratique en un seul facteur d'entrée. Dans le cas contraire, tous les éléments de la colonne de **F** associée à un tel terme seraient égaux à un, et cette colonne serait donc identique à celle associée au terme constant. $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ ne serait donc plus inversible. On peut remédier à ce problème en utilisant des plans factoriels à trois niveaux. □

*Exemple 9.5. Approximation d'une fonction au sens des moindres carrés sur* $[-1,1]$

Nous recherchons le polynôme (9.50) qui approxime au mieux une fonction $f(\cdot)$ sur l'intervalle normalisé $[-1,1]$ au sens de la fonction de coût

$$J(\mathbf{x}) = \int_{-1}^{1} [f(\tau) - P_k(\tau,\mathbf{x})]^2 \mathrm{d}\tau. \tag{9.62}$$

La valeur optimale $\widehat{\mathbf{x}}$ du vecteur **x** des coefficients du polynôme satisfait la contrepartie continue des équations normales

$$\mathbf{M}\widehat{\mathbf{x}} = \mathbf{v}, \tag{9.63}$$

où $m_{i,j} = \int_{-1}^{1} \tau^{i-1}\tau^{j-1}\mathrm{d}\tau$ et $v_i = \int_{-1}^{1} \tau^{i-1} f(\tau)\mathrm{d}\tau$, et cond **M** se détériore drastiquement quand l'ordre $k$ du polynôme $P_k$ augmente. Si l'on écrit plutôt le polynôme sous la forme

$$P_k(t,\mathbf{x}) = \sum_{i=0}^{k} p_i \phi_i(t), \tag{9.64}$$

où **x** reste égal à $(p_0, p_1, p_2, \cdots, p_k)^{\mathrm{T}}$, mais où les $\phi_i$ sont des polynômes de Legendre, définis par (5.23), alors les éléments de **M** satisfont

$$m_{i,j} = \int_{-1}^{1} \phi_{i-1}(\tau)\phi_{j-1}(\tau)\mathrm{d}\tau = \beta_{i-1}\delta_{i,j}, \tag{9.65}$$

avec

$$\beta_{i-1} = \frac{2}{2i-1}. \tag{9.66}$$

Dans (9.65) $\delta_{i,j}$ est un *delta de Kronecker*, égal à un si $i = j$ et à zéro autrement, de sorte que **M** est diagonale. Ceci découple les équations scalaires contenues dans (9.63), et les coefficients optimaux $\widehat{p}_i$ dans la base de Legendre peuvent ainsi être calculés individuellement par

$$\widehat{p}_i = \frac{1}{\beta_i} \int_{-1}^{1} \phi_i(\tau) f(\tau) \mathrm{d}\tau, \quad i = 0, \cdots, k. \tag{9.67}$$

L'estimation des $\widehat{p}_i$ se ramène donc à l'évaluation d'intégrales définies (voir le chapitre 6). Si l'on veut augmenter d'une unité le degré du polynôme approximateur, il suffit de calculer $\widehat{p}_{k+1}$, puisque les autres coefficients sont inchangés. □

En général, cependant, il vaut mieux éviter de calculer $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ et utiliser une factorisation de $\mathbf{F}$, comme dans les deux sections qui suivent. L'histoire de la méthode des moindres carrés et de sa mise en œuvre via des factorisations de matrices est racontée dans [172], où le concept utile de *moindres carrés totaux* est également expliqué.

### 9.2.3 Moindres carrés linéaires via une factorisation QR

La factorisation QR a été présentée en section 3.6.5 pour les matrices carrées. Rappelons qu'elle peut être mise en œuvre par une série de transformations de Householder numériquement stables et que toute bibliothèque décente de programmes scientifiques en contient une implémentation.

Considérons maintenant une matrice $\mathbf{F}$ rectangulaire $N \times n$, avec $N \geqslant n$. La même approche qu'en section 3.6.5 permet de calculer une matrice $\mathbf{Q}$ orthonormale $N \times N$ et une matrice $\mathbf{R}$ triangulaire supérieure $N \times n$ telles que

$$\mathbf{F} = \mathbf{QR}. \tag{9.68}$$

Puisque les $N - n$ dernières lignes de $\mathbf{R}$ ne contiennent que des zéros, on peut aussi écrire

$$\mathbf{F} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{O} \end{bmatrix} = \mathbf{Q}_1 \mathbf{R}_1, \tag{9.69}$$

où $\mathbf{O}$ est une matrice de zéros. La factorisation de $\mathbf{F}$ la plus à droite dans (9.69) est appelée factorisation QR *réduite*, ou *thin QR factorization* [80]. $\mathbf{Q}_1$ a les mêmes dimensions que $\mathbf{F}$ et est telle que

$$\mathbf{Q}_1^{\mathrm{T}} \mathbf{Q}_1 = \mathbf{I}_n. \tag{9.70}$$

$\mathbf{R}_1$ est une matrice carrée et triangulaire supérieure, qui est inversible si les colonnes de $\mathbf{F}$ sont linéairement indépendantes.

Supposons que tel soit le cas, et prenons (9.69) en compte dans (9.49) pour obtenir

$$\begin{aligned} \widehat{\mathbf{x}} &= \left(\mathbf{F}^{\mathrm{T}}\mathbf{F}\right)^{-1} \mathbf{F}^{\mathrm{T}} \mathbf{y} && (9.71) \\ &= (\mathbf{R}_1^{\mathrm{T}} \mathbf{Q}_1^{\mathrm{T}} \mathbf{Q}_1 \mathbf{R}_1)^{-1} \mathbf{R}_1^{\mathrm{T}} \mathbf{Q}_1^{\mathrm{T}} \mathbf{y} && (9.72) \\ &= \mathbf{R}_1^{-1} \left(\mathbf{R}_1^{\mathrm{T}}\right)^{-1} \mathbf{R}_1^{\mathrm{T}} \mathbf{Q}_1^{\mathrm{T}} \mathbf{y}, && (9.73) \end{aligned}$$

de sorte que

$$\widehat{\mathbf{x}} = \mathbf{R}_1^{-1}\mathbf{Q}_1^{\mathrm{T}}\mathbf{y}. \tag{9.74}$$

Il n'est bien sûr pas nécessaire d'inverser $\mathbf{R}_1$, et il vaut mieux calculer $\widehat{\mathbf{x}}$ en résolvant le système triangulaire

$$\mathbf{R}_1\widehat{\mathbf{x}} = \mathbf{Q}_1^{\mathrm{T}}\mathbf{y}. \tag{9.75}$$

L'estimée $\widehat{\mathbf{x}}$ au sens des moindres carrés est ainsi obtenue directement à partir de la factorisation QR de $\mathbf{F}$, sans jamais calculer $\mathbf{F}^{\mathrm{T}}\mathbf{F}$. Ceci a un coût, car plus de calculs sont requis que pour la résolution des équations normales (9.56).

*Remarque 9.7.* Plutôt que de factoriser $\mathbf{F}$, il peut s'avérer plus commode de factoriser la matrice composite $[\mathbf{F}|\mathbf{y}]$ pour obtenir

$$[\mathbf{F}|\mathbf{y}] = \mathbf{QR}. \tag{9.76}$$

Le coût $J(\mathbf{x})$ satisfait alors

$$\begin{aligned} J(\mathbf{x}) &= \|\mathbf{F}\mathbf{x}-\mathbf{y}\|_2^2 && (9.77)\\ &= \left\| [\mathbf{F}|\mathbf{y}] \begin{bmatrix} \mathbf{x} \\ -1 \end{bmatrix} \right\|_2^2 && (9.78)\\ &= \left\| \mathbf{Q}^{\mathrm{T}}[\mathbf{F}|\mathbf{y}] \begin{bmatrix} \mathbf{x} \\ -1 \end{bmatrix} \right\|_2^2 && (9.79)\\ &= \left\| \mathbf{R} \begin{bmatrix} \mathbf{x} \\ -1 \end{bmatrix} \right\|_2^2. && (9.80) \end{aligned}$$

Puisque $\mathbf{R}$ est triangulaire supérieure, on peut l'écrire

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{O} \end{bmatrix}, \tag{9.81}$$

où $\mathbf{O}$ est une matrice de zéros et où $\mathbf{R}_1$ est une matrice carrée et triangulaire supérieure

$$\mathbf{R}_1 = \begin{bmatrix} \mathbf{U} & \mathbf{v} \\ \mathbf{0}^{\mathrm{T}} & \alpha \end{bmatrix}. \tag{9.82}$$

L'équation (9.80) implique alors que

$$J(\mathbf{x}) = \|\mathbf{U}\mathbf{x}-\mathbf{v}\|_2^2 + \alpha^2, \tag{9.83}$$

de sorte que $\widehat{\mathbf{x}}$ est la solution du système linéaire

$$\mathbf{U}\widehat{\mathbf{x}} = \mathbf{v}, \tag{9.84}$$

et que le coût minimum est

$$J(\widehat{\mathbf{x}}) = \alpha^2. \tag{9.85}$$

$J(\widehat{\mathbf{x}})$ est ainsi obtenu directement à partir de la factorisation QR, sans avoir à résoudre (9.84). Ceci peut être particulièrement intéressant si l'on doit choisir entre plusieurs structures de modèles en compétition (par exemple des modèles polynomiaux d'ordre croissant) et qu'on ne veut calculer $\widehat{\mathbf{x}}$ que pour le meilleur d'entre eux. Notons que la structure de modèle qui conduit à la plus petite valeur de $J(\widehat{\mathbf{x}})$ est très souvent la plus complexe, de sorte qu'une pénalisation sous une forme ou sous une autre de la complexité du modèle est en général nécessaire. □

### *9.2.4 Moindres carrés linéaires via une SVD*

Une décomposition en valeurs singulières (ou SVD, pour *singular value decomposition*) requiert encore plus de calculs qu'une factorisation QR mais peut faciliter le traitement de problèmes où les colonnes de $\mathbf{F}$ sont linéairement dépendantes ou presque linéairement dépendantes, voir les sections 9.2.5 et 9.2.6.

Toute matrice $\mathbf{F}$ de dimensions $N \times n$ avec $N \geqslant n$ peut être factorisée comme

$$\mathbf{F} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}}, \tag{9.86}$$

où

— $\mathbf{U}$ a les mêmes dimensions que $\mathbf{F}$, et est telle que

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}_n, \tag{9.87}$$

— $\boldsymbol{\Sigma}$ est une matrice diagonale $n \times n$, dont les éléments diagonaux $\sigma_i$ sont les *valeurs singulières* de $\mathbf{F}$, avec $\sigma_i \geqslant 0$,

— $\mathbf{V}$ est une matrice $n \times n$ telle que

$$\mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I}_n. \tag{9.88}$$

L'équation (9.86) implique que

$$\mathbf{F}\mathbf{V} = \mathbf{U}\boldsymbol{\Sigma}, \tag{9.89}$$

$$\mathbf{F}^{\mathrm{T}}\mathbf{U} = \mathbf{V}\boldsymbol{\Sigma}. \tag{9.90}$$

En d'autres termes,

$$\mathbf{F}\mathbf{v}^i = \sigma_i \mathbf{u}^i, \tag{9.91}$$

$$\mathbf{F}^{\mathrm{T}}\mathbf{u}^i = \sigma_i \mathbf{v}^i, \tag{9.92}$$

où $\mathbf{v}^i$ est la $i$-ème colonne de $\mathbf{V}$ et $\mathbf{u}^i$ la $i$-ème colonne de $\mathbf{U}$. C'est pourquoi $\mathbf{v}^i$ et $\mathbf{u}^i$ sont respectivement appelés *vecteurs singuliers à droite et à gauche*.

*Remarque 9.8.* Si (9.88) implique que

$$\mathbf{V}^{-1} = \mathbf{V}^{\mathrm{T}}, \tag{9.93}$$

par contre (9.87) ne fournit aucune recette magique pour inverser **U**, qui n'est pas carrée ! □

Le calcul de la SVD (9.86) est classiquement mené à bien en deux étapes [77], [79]. Durant la première, on calcule des matrices orthonormales $\mathbf{P}_1$ et $\mathbf{Q}_1$ pour assurer que

$$\mathbf{B} = \mathbf{P}_1^{\mathrm{T}} \mathbf{F} \mathbf{Q}_1 \tag{9.94}$$

soit bidiagonale (c'est à dire qu'elle n'ait des éléments non nuls que sur sa diagonale principale et sur la diagonale juste au dessus), et que tous les éléments de ses $N-n$ dernières lignes soient nuls. Comme la multiplication d'une matrice à droite ou à gauche par une matrice orthonormale préserve ses valeurs singulières, les valeurs singulières de **B** sont les mêmes que celles de **F**. Le calcul de $\mathbf{P}_1$ et $\mathbf{Q}_1$ est mené à bien grâce à deux séries de transformations de Householder. Les dimensions de **B** sont identiques à celles de **F**, mais puisque les $N-n$ dernières lignes de **B** ne contiennent que des zéros, on forme une matrice $\tilde{\mathbf{P}}_1$ de dimensions $N \times n$ avec les $n$ premières colonnes de $\mathbf{P}_1$ pour obtenir la représentation plus économique

$$\tilde{\mathbf{B}} = \tilde{\mathbf{P}}_1^{\mathrm{T}} \mathbf{F} \mathbf{Q}_1, \tag{9.95}$$

où $\tilde{\mathbf{B}}$ est carrée, bidiagonale et formée des $n$ premières lignes de **B**.

Durant la seconde étape, on calcule des matrices orthonormales $\mathbf{P}_2$ et $\mathbf{Q}_2$ pour assurer que

$$\mathbf{\Sigma} = \mathbf{P}_2^{\mathrm{T}} \tilde{\mathbf{B}} \mathbf{Q}_2 \tag{9.96}$$

soit diagonale. Ceci est effectué par une variante de l'algorithme d'itération QR présenté en section 4.3.6. Globalement,

$$\mathbf{\Sigma} = \mathbf{P}_2^{\mathrm{T}} \tilde{\mathbf{P}}_1^{\mathrm{T}} \mathbf{F} \mathbf{Q}_1 \mathbf{Q}_2, \tag{9.97}$$

et (9.86) est satisfaite, avec $\mathbf{U} = \tilde{\mathbf{P}}_1 \mathbf{P}_2$ et $\mathbf{V}^{\mathrm{T}} = \mathbf{Q}_2^{\mathrm{T}} \mathbf{Q}_1^{\mathrm{T}}$.

Le lecteur est invité à consulter [49] pour plus de détails sur des méthodes modernes de calcul de SVD, et [50] pour se faire une idée des efforts qui ont été consacrés à des améliorations d'efficacité et de robustesse. Des programmes de calcul de SVD sont largement disponibles, et il faut éviter avec soin toute tentative d'en bricoler un nouveau.

*Remarque 9.9.* La SVD a de nombreuses autres applications que l'évaluation d'estimées au sens des moindres carrés linéaires de façon numériquement robuste. Voici quelques unes de ses propriétés les plus importantes :

- les rangs de **F** et $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ sont égaux au nombre des valeurs singulières non nulles de **F**, de sorte que $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ est inversible si et seulement si toutes les valeurs singulières de **F** diffèrent de zéro ;
- Les valeurs singulières de **F** sont les racines carrées des valeurs propres de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ (mais ce n'est pas comme cela qu'elles sont calculées) ;
- Si les valeurs singulières de **F** sont classées par ordre décroissant et si $\sigma_k > 0$, alors la matrice de rang $k$ la plus proche de **F** au sens de la norme spectrale est

$$\widehat{\mathbf{F}}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\mathrm{T}}, \tag{9.98}$$

et elle est telle que

$$||\mathbf{F} - \widehat{\mathbf{F}}_k||_2 = \sigma_{k+1}. \tag{9.99}$$

□

En supposant toujours, pour le moment, que $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ est inversible, remplaçons $\mathbf{F}$ dans (9.49) par $\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}}$ pour obtenir

$$\begin{aligned} \widehat{\mathbf{x}} &= (\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^{\mathrm{T}}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}})^{-1}\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^{\mathrm{T}}\mathbf{y} && (9.100)\\ &= (\mathbf{V}\boldsymbol{\Sigma}^2\mathbf{V}^{\mathrm{T}})^{-1}\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^{\mathrm{T}}\mathbf{y} && (9.101)\\ &= (\mathbf{V}^{\mathrm{T}})^{-1}\boldsymbol{\Sigma}^{-2}\mathbf{V}^{-1}\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^{\mathrm{T}}\mathbf{y}. && (9.102) \end{aligned}$$

Puisque

$$(\mathbf{V}^{\mathrm{T}})^{-1} = \mathbf{V}, \tag{9.103}$$

ceci revient à écrire

$$\widehat{\mathbf{x}} = \mathbf{V}\boldsymbol{\Sigma}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{y}. \tag{9.104}$$

Comme avec la factorisation QR, la solution $\widehat{\mathbf{x}}$ est donc évaluée sans jamais calculer $\mathbf{F}^{\mathrm{T}}\mathbf{F}$. L'inversion de $\boldsymbol{\Sigma}$ est triviale, puisque $\boldsymbol{\Sigma}$ est diagonale.

*Remarque 9.10.* Toutes les méthodes d'obtention de $\widehat{\mathbf{x}}$ qui viennent d'être décrites sont *mathématiquement* équivalentes, mais leurs propriétés *numériques* diffèrent. □

Nous avons vu en section 3.3 que le conditionnement d'une matrice carrée pour la norme spectrale est le rapport de sa plus grande valeur singulière sur sa plus petite, et ceci reste vrai pour des matrices rectangulaires telles que $\mathbf{F}$. Or

$$\mathbf{F}^{\mathrm{T}}\mathbf{F} = \mathbf{V}\boldsymbol{\Sigma}^{\mathrm{T}}\mathbf{U}^{\mathrm{T}}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}} = \mathbf{V}\boldsymbol{\Sigma}^2\mathbf{V}^{\mathrm{T}}, \tag{9.105}$$

qui est une SVD de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$. Chaque valeur singulière de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ est donc égale au carré de la valeur singulière correspondante de $\mathbf{F}$. Pour la norme spectrale, ceci implique que

$$\operatorname{cond}\left(\mathbf{F}^{\mathrm{T}}\mathbf{F}\right) = (\operatorname{cond}\mathbf{F})^2. \tag{9.106}$$

L'utilisation des équations normales peut ainsi conduire à une dégradation drastique du conditionnement. Si, par exemple, $\operatorname{cond}\mathbf{F} = 10^{10}$, alors $\operatorname{cond}\left(\mathbf{F}^{\mathrm{T}}\mathbf{F}\right) = 10^{20}$ et il y a peu d'espoir d'obtenir des résultats précis quand on résout les équations normales avec des flottants en double précision.

*Remarque 9.11.* L'évaluation de $\operatorname{cond}\mathbf{F}$ pour la norme spectrale demande à peu près autant d'efforts que le calcul d'une SVD de $\mathbf{F}$, de sorte qu'on peut préférer utiliser une autre définition du conditionnement en cas de doute pour décider s'il vaut la peine de calculer une SVD. La fonction MATLAB `condest` fournit une valeur approchée du conditionnement pour la norme 1. □

Les approches à base de factorisation QR ou de SVD sont conçues pour ne pas détériorer le conditionnement du système linéaire à résoudre. La factorisation QR y parvient avec moins de calculs que la SVD, et devrait donc être l'outil standard pour la résolution des problèmes de moindres carrés linéaires. Nous verrons sur quelques exemples que la solution obtenue via une factorisation QR peut se révéler légèrement plus précise que celle obtenue via une SVD. La SVD peut être préférée quand le problème est extrêmement mal conditionné, pour des raisons détaillées dans les deux sections qui suivent.

### 9.2.5 *Que faire si* $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ *n'est pas inversible ?*

Quand $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ n'est pas inversible, certaines colonnes de $\mathbf{F}$ sont linéairement dépendantes. La solution au sens des moindres carrés n'est alors plus unique. Ceci ne devrait pas arriver en principe, si le modèle a été bien choisi (après tout, il suffit de supprimer des colonnes appropriées de $\mathbf{F}$ et les paramètres qui leur correspondent pour assurer l'indépendance linéaire des colonnes restantes de $\mathbf{F}$). Ce cas pathologique est néanmoins intéressant, comme une version chimiquement pure d'un problème beaucoup plus fréquent, à savoir la quasi dépendance linéaire de colonnes de $\mathbf{F}$, qui sera considérée en section 9.2.6.

Parmi l'infinité non dénombrable d'estimées au sens des moindres carrés qui existent dans ce cas dégénéré, la plus petite au sens de la norme euclidienne est donnée par

$$\widehat{\mathbf{x}} = \mathbf{V}\widehat{\mathbf{\Sigma}}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{y}, \tag{9.107}$$

où $\widehat{\mathbf{\Sigma}}^{-1}$ est une matrice diagonale, dont le $i$-ème élément diagonal est égal à $1/\sigma_i$ si $\sigma_i \neq 0$ et à zéro autrement.

*Remarque 9.12.* Contrairement à ce que suggère cette notation, $\widehat{\mathbf{\Sigma}}^{-1}$ est singulière, bien sûr. □

### 9.2.6 *Régularisation de problèmes mal conditionnés*

Il arrive souvent que le rapport des valeurs singulières extrêmes de $\mathbf{F}$ soit très grand, ce qui indique que certaines colonnes de $\mathbf{F}$ sont presque linéairement dépendantes. Le conditionnement de $\mathbf{F}$ est alors aussi très grand, et celui de $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ encore pire. En conséquence, bien que $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ reste inversible mathématiquement, l'estimée au sens des moindres carrés $\widehat{\mathbf{x}}$ devient très sensible à de petites variations des données, ce qui fait de l'estimation un problème *mal conditionné*. Parmi les nombreuses approches de *régularisation* disponibles pour faire face à cette difficulté, une approche particulièrement simple est de forcer à zéro toute valeur singulière de $\mathbf{F}$ qui est plus petite qu'un seuil $\delta$ à régler par l'utilisateur. Ceci revient à approxi-

mer $\mathbf{F}$ par une matrice de rang inférieur, à laquelle la procédure de la section 9.2.5 peut alors être appliquée. La solution régularisée est encore donnée par

$$\widehat{\mathbf{x}} = \mathbf{V}\widehat{\mathbf{\Sigma}}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{y}, \tag{9.108}$$

mais le $i$-ème élément diagonal de la matrice diagonale $\widehat{\mathbf{\Sigma}}^{-1}$ est maintenant égal à $1/\sigma_i$ si $\sigma_i > \delta$ et à zéro dans le cas contraire.

*Remarque 9.13.* Quand de l'information est disponible a priori sur les valeurs possibles de $\mathbf{x}$, une *approche bayésienne* de la régularisation peut être préférable [244]. Si, par exemple, la distribution a priori de $\mathbf{x}$ est supposée gaussienne, de moyenne $\mathbf{x}_0$ connue et de variance $\mathbf{\Omega}$ connue, alors l'estimée $\widehat{\mathbf{x}}_{\mathrm{map}}$ de $\mathbf{x}$ *au sens du maximum a posteriori* satisfait le système linéaire

$$(\mathbf{F}^{\mathrm{T}}\mathbf{F} + \mathbf{\Omega}^{-1})\widehat{\mathbf{x}}_{\mathrm{map}} = \mathbf{F}^{\mathrm{T}}\mathbf{y} + \mathbf{\Omega}^{-1}\mathbf{x}_0, \tag{9.109}$$

qui devrait être beaucoup mieux conditionné que les équations normales. □

## 9.3 Méthodes itératives

Quand la fonction de coût $J(\cdot)$ n'est pas quadratique en son argument, la méthode des moindres carrés linéaires de la section 9.2 ne s'applique pas, et l'on est souvent conduit à utiliser des méthodes itératives d'optimisation non linéaire, encore connue sous le nom de programmation non linéaire. À partir d'une estimée $\mathbf{x}^k$ d'un minimiseur à l'itération $k$, ces méthodes calculent $\mathbf{x}^{k+1}$ tel que

$$J(\mathbf{x}^{k+1}) \leqslant J(\mathbf{x}^k). \tag{9.110}$$

Pourvu que $J(\mathbf{x})$ soit borné inférieurement (comme c'est le cas si $J(\mathbf{x})$ est une norme), ceci assure la convergence de la suite $\{J(\mathbf{x}^k)\}_{k=0}^{\infty}$. Sauf si l'algorithme reste coincé en $\mathbf{x}^0$, les performances telles que mesurées par la fonction de coût auront donc été améliorées.

Ceci soulève deux questions importantes que nous laisserons de côté jusqu'à la section 9.3.4.8 :

— d'où partir (comment choisir $\mathbf{x}^0$) ?
— quand s'arrêter ?

Avant de quitter complètement les moindres carrés linéaires, considérons un cas où ils peuvent être utilisés pour réduire la dimension de l'espace de recherche.

### *9.3.1 Moindres carrés séparables*

Supposons que la fonction de coût reste quadratique en l'erreur

$$J(\mathbf{x}) = \|\mathbf{y} - \mathbf{f}(\mathbf{x})\|_2^2, \tag{9.111}$$

et que le vecteur de décision $\mathbf{x}$ puisse être coupé en deux vecteurs $\mathbf{p}$ et $\boldsymbol{\theta}$, de telle façon que

$$\mathbf{f}(\mathbf{x}) = \mathbf{F}(\boldsymbol{\theta})\mathbf{p}. \tag{9.112}$$

Le vecteur d'erreur

$$\mathbf{e}(\mathbf{x}) = \mathbf{y} - \mathbf{F}(\boldsymbol{\theta})\mathbf{p} \tag{9.113}$$

est alors affine en $\mathbf{p}$. Pour toute valeur donnée de $\boldsymbol{\theta}$, la valeur optimale correspondante $\widehat{\mathbf{p}}(\boldsymbol{\theta})$ de $\mathbf{p}$ peut donc être calculée par moindres carrés linéaires, afin de limiter la recherche non linéaire à l'espace des $\boldsymbol{\theta}$.

*Exemple 9.6. Ajustement d'une somme d'exponentielles sur des données*
Si la $i$-ème donnée $y_i$ est modélisée comme

$$f_i(\mathbf{p}, \boldsymbol{\theta}) = \sum_{j=1}^{m} p_j \, e^{-\theta_j t_i}, \tag{9.114}$$

où l'instant de de mesure $t_i$ est connu, alors le résidu $y_i - f_i(\mathbf{p}, \boldsymbol{\theta})$ est affine en $\mathbf{p}$ et non linéaire en $\boldsymbol{\theta}$. La dimension de l'espace de recherche peut donc être divisée par deux en utilisant les moindres carrés linéaires pour calculer $\widehat{\mathbf{p}}(\boldsymbol{\theta})$, ce qui est une simplification considérable. □

## 9.3.2 Recherche unidimensionnelle

De nombreuses méthodes itératives d'optimisation multivariables définissent des directions dans lesquelles des recherches unidimensionnelles sont effectuées. Comme il faut souvent de nombreuses recherches de ce type, leur but est modeste : elles doivent assurer une diminution significative du coût pour aussi peu de calculs que possible. Les méthodes pour ce faire sont plus des recettes de cuisine sophistiquées que de la science dure ; celles présentées brièvement ci-dessous résultent d'une sélection naturelle à laquelle peu d'autres ont survécu.

*Remarque 9.14.* Une alternative au choix d'une direction suivi d'une recherche dans cette direction correspond aux *méthodes à régions de confiance*, ou *trust-region methods* [173]. Ces méthodes utilisent un modèle quadratique comme approximation de la fonction d'objectif sur une région de confiance pour choisir *simultanément* la direction et la longueur du déplacement du vecteur de décision. La taille de la région de confiance est mise à jour sur la base des performances passés de l'algorithme. □

### 9.3.2.1 Interpolation parabolique

Soit $\lambda$ le paramètre scalaire associé à la direction de recherche $\mathbf{d}$. La valeur à lui donner peut être choisie à l'aide d'une *interpolation parabolique*. Un polynôme du

second ordre $P_2(\lambda)$ est alors utilisé pour interpoler

$$f(\lambda) = J(\mathbf{x}^k + \lambda \mathbf{d}) \tag{9.115}$$

en $\lambda_i$, $i = 1,2,3$, avec $\lambda_1 < \lambda_2 < \lambda_3$. La formule d'interpolation de Lagrange (5.14) se traduit par

$$\begin{aligned} P_2(\lambda) &= \frac{(\lambda-\lambda_2)(\lambda-\lambda_3)}{(\lambda_1-\lambda_2)(\lambda_1-\lambda_3)} f(\lambda_1) \\ &+ \frac{(\lambda-\lambda_1)(\lambda-\lambda_3)}{(\lambda_2-\lambda_1)(\lambda_2-\lambda_3)} f(\lambda_2) \\ &+ \frac{(\lambda-\lambda_1)(\lambda-\lambda_2)}{(\lambda_3-\lambda_1)(\lambda_3-\lambda_2)} f(\lambda_3). \end{aligned} \tag{9.116}$$

*Pourvu que* $P_2(\lambda)$ soit convexe et que les points $(\lambda_i, f(\lambda_i))$ $(i = 1,2,3)$ ne soient pas colinéaires, $P_2(\lambda)$ est minimal en

$$\widehat{\lambda} = \lambda_2 - \frac{1}{2} \frac{(\lambda_2-\lambda_1)^2[f(\lambda_2)-f(\lambda_3)] - (\lambda_2-\lambda_3)^2[f(\lambda_2)-f(\lambda_1)]}{(\lambda_2-\lambda_1)[f(\lambda_2)-f(\lambda_3)] - (\lambda_2-\lambda_3)[f(\lambda_2)-f(\lambda_1)]}, \tag{9.117}$$

qui est alors utilisé pour calculer

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \widehat{\lambda}\mathbf{d}. \tag{9.118}$$

Les ennuis commencent quand les points $(\lambda_i, f(\lambda_i))$ sont colinéaires (car le dénominateur de (9.117) devient alors égal à zéro) ou quand $P_2(\lambda)$ se révèle concave (car $P_2(\lambda)$ est alors *maximal* en $\widehat{\lambda}$). C'est pourquoi on utilise en pratique des méthodes de recherche unidimensionnelles plus sophistiquées, comme la méthode de Brent.

#### 9.3.2.2 Méthode de Brent

La méthode de Brent [29] est une stratégie d'interpolation parabolique sécurisée, décrite de façon très détaillée dans [186]. Contrairement à la méthode de Wolfe de la section 9.3.2.3, elle ne requiert pas l'évaluation du gradient du coût et est donc intéressante quand ce gradient n'est pas disponible ou quand il est évalué par différences finies et donc coûteux.

La première étape est d'encadrer un minimiseur (local) $\widehat{\lambda}$ dans un intervalle $[\lambda_{\min}, \lambda_{\max}]$ en se déplaçant dans la direction qui fait baisser le coût jusqu'à ce que $f(\lambda)$ recommence à augmenter. La recherche unidimensionnelle est alors restreinte à cet intervalle. Quand la fonction $f(\lambda)$ définie par (9.115) est considérée comme suffisamment coopérative, ce qui veut dire (entre autres) que le polynôme interpolateur $P_2(\lambda)$ est convexe et que son minimiseur est dans $[\lambda_{\min}, \lambda_{\max}]$, la méthode utilise (9.117) et (9.118) pour calculer $\widehat{\lambda}$ puis $\mathbf{x}^{k+1}$. En cas de difficulté, elle bascule sur une approche plus lente mais plus robuste. Si le gradient du coût était disponible,

$\dot{f}(\lambda)$ serait facile à calculer et on pourrait employer la méthode de bissection de la section 7.3.1 pour résoudre $\dot{f}(\widehat{\lambda}) = 0$. A la place, $f(\lambda)$ est évaluée en deux points $\lambda_{k,1}$ et $\lambda_{k,2}$, pour acquérir une information de pente. Ces points sont positionnés pour que, à l'itération $k$, $\lambda_{k,1}$ et $\lambda_{k,2}$ soient à une fraction $\alpha$ des extrémités de l'intervalle de recherche courant $[\lambda^k_{\min}, \lambda^k_{\max}]$, avec

$$\alpha = \frac{\sqrt{5}-1}{2} \approx 0.618. \tag{9.119}$$

Ainsi,

$$\lambda_{k,1} = \lambda^k_{\min} + (1-\alpha)(\lambda^k_{\max} - \lambda^k_{\min}), \tag{9.120}$$
$$\lambda_{k,2} = \lambda^k_{\min} + \alpha(\lambda^k_{\max} - \lambda^k_{\min}). \tag{9.121}$$

Si $f(\lambda_{k,1}) < f(\lambda_{k,2})$, alors le sous-intervalle $(\lambda_{k,2}, \lambda^k_{\max}]$ est éliminé, ce qui laisse

$$[\lambda^{k+1}_{\min}, \lambda^{k+1}_{\max}] = [\lambda^k_{\min}, \lambda_{k,2}], \tag{9.122}$$

Dans le cas contraire, le sous-intervalle $[\lambda^k_{\min}, \lambda_{k,1})$ est éliminé, ce qui laisse

$$[\lambda^{k+1}_{\min}, \lambda^{k+1}_{\max}] = [\lambda_{k,1}, \lambda^k_{\max}]. \tag{9.123}$$

Dans les deux cas, l'un des deux points d'évaluation de l'itération $k$ reste dans l'intervalle de recherche ainsi mis à jour, et se révèle commodément situé à une fraction $\alpha$ de l'une de ses extrémités. Chaque itération sauf la première ne requiert donc qu'une évaluation additionnelle de la fonction de coût, car l'autre point est l'un des deux utilisés lors de l'itération précédente. Ceci correspond à la méthode de la *section dorée*, à cause de la relation entre $\alpha$ et le nombre d'or.

Même si la méthode de la section dorée fait un usage économe des évaluations du coût, elle est beaucoup plus lente que l'interpolation parabolique dans ses bons jours, et la méthode de Brent bascule sur (9.117) et (9.118) dès que les conditions redeviennent favorables.

*Remarque 9.15.* Quand il faut à peu près le même temps pour évaluer le gradient de la fonction de coût que pour évaluer la fonction de coût elle-même, on peut remplacer la méthode de Brent par une interpolation cubique sécurisée, où l'on demande à un polynôme de degré trois d'interpoler $f(\lambda)$ et d'avoir la même pente en deux points d'essai [73]. La méthode de la section dorée peut alors être remplacée par une méthode de bissection à la recherche de $\widehat{\lambda}$ tel que $\dot{f}(\widehat{\lambda}) = 0$ quand les résultats de l'interpolation cubique deviennent inacceptables. □

#### 9.3.2.3 Méthode de Wolfe

La méthode de Wolfe [173, 22, 184] met en œuvre une *recherche unidimensionnelle inexacte*, ce qui veut dire qu'elle recherche une valeur de $\lambda$ *raisonnable* plutôt

qu'optimale. Tout comme dans la remarque 9.15, la méthode de Wolfe suppose que la fonction gradient $\mathbf{g}(\cdot)$ peut être évaluée. Cette méthode est en général employée pour les recherches unidimensionnelles des algorithmes de quasi-Newton et de gradients conjugués présentés dans les sections 9.3.4.5 et 9.3.4.6.

Deux inégalités sont utilisées pour spécifier les propriétés à satisfaire par $\lambda$. La première, connue sous le nom de *condition d'Armijo*, dit que $\lambda$ doit assurer une *diminution suffisamment rapide* du coût lors d'un déplacement à partir de $\mathbf{x}^k$ dans la direction de recherche $\mathbf{d}$. Elle se traduit par

$$J(\mathbf{x}^{k+1}(\lambda)) \leqslant J(\mathbf{x}^k) + \alpha_1 \lambda \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{d}, \tag{9.124}$$

où

$$\mathbf{x}^{k+1}(\lambda) = \mathbf{x}^k + \lambda \mathbf{d} \tag{9.125}$$

et où le coût est considéré comme une fonction de $\lambda$. Si cette fonction est dénotée par $f(\cdot)$

$$f(\lambda) = J(\mathbf{x}^k + \lambda \mathbf{d}), \tag{9.126}$$

alors

$$\dot{f}(0) = \frac{\partial J(\mathbf{x}^k + \lambda \mathbf{d})}{\partial \lambda}(\lambda = 0) = \frac{\partial J}{\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}^k) \cdot \frac{\partial \mathbf{x}^{k+1}}{\partial \lambda} = \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{d}. \tag{9.127}$$

Ainsi, $\mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{d}$ dans (9.124) est la pente initiale de la fonction de coût vue comme une fonction de $\lambda$. La condition d'Armijo fournit une borne supérieure de la valeur souhaitable pour $J(\mathbf{x}^{k+1}(\lambda))$, borne qui est affine en $\lambda$. Puisque $\mathbf{d}$ est une direction de descente, $\mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{d} < 0$ et $\lambda > 0$. La condition (9.124) dit que plus $\lambda$ est grand plus le coût doit devenir petit. Le paramètre interne $\alpha_1$ doit vérifier $0 < \alpha_1 < 1$, et on le prend en général petit (une valeur typique est $\alpha_1 = 10^{-4}$).

La condition d'Armijo est satisfaite pour tout $\lambda$ suffisamment petit, de sorte qu'il faut induire une stratégie plus audacieuse. C'est le rôle de la seconde inégalité, connue comme la *condition de courbure*, qui demande que $\lambda$ satisfasse aussi

$$\dot{f}(\lambda) \geqslant \alpha_2 \dot{f}(0), \tag{9.128}$$

où $\alpha_2 \in (\alpha_1, 1)$ (une valeur typique est $\alpha_2 = 0.5$). L'équation (9.128) se traduit par

$$\mathbf{g}^{\mathrm{T}}(\mathbf{x}^k + \lambda \mathbf{d})\mathbf{d} \geqslant \alpha_2 \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{d}. \tag{9.129}$$

Comme $\dot{f}(0) < 0$, tout $\lambda$ tel que $\dot{f}(\lambda) > 0$ satisfera (9.128). Pour éviter ceci, les *conditions de Wolfe fortes* remplacent la condition de courbure (9.129) par

$$|\mathbf{g}^{\mathrm{T}}(\mathbf{x}^k + \lambda \mathbf{d})\mathbf{d}| \leqslant |\alpha_2 \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{d}|, \tag{9.130}$$

tout en gardant inchangée la condition d'Armijo (9.124). Avec (9.130), $\dot{f}(\lambda)$ peut encore devenir positif, mais ne peut plus devenir trop grand.

Pourvu que la fonction de coût $J(\cdot)$ soit lisse et bornée par en dessous, l'existence de valeurs de $\lambda$ qui satisfont les conditions de Wolfe et les conditions de

Wolfe fortes est garantie. On trouve dans [173] les principes d'une recherche unidimensionnelle dont on peut garantir qu'elle trouvera un $\lambda$ satisfaisant les conditions de Wolfe fortes. Plusieurs mises en œuvres informatiques de ces principes sont dans le domaine public.

### *9.3.3 Combinaison de recherches unidimensionnelles*

Quand on dispose d'un algorithme de recherche unidimensionnelle, il est tentant de pratiquer des recherches multidimensionnelles en effectuant cycliquement des recherches unidimensionnelles sur chacune des composantes de $\mathbf{x}$ successivement. C'est cependant une *mauvaise idée*, car la recherche est alors confinée à des déplacements le long des axes de l'espace de décision alors que d'autres directions de recherche pourraient être beaucoup plus appropriées. La figure 9.2 illustre une situation où l'on doit minimiser l'altitude par rapport à la longitude $x_1$ et à la latitude $x_2$ près d'une rivière. La taille des déplacements devient vite désespérément petite parce qu'il n'est pas possible de se déplacer le long de la vallée.

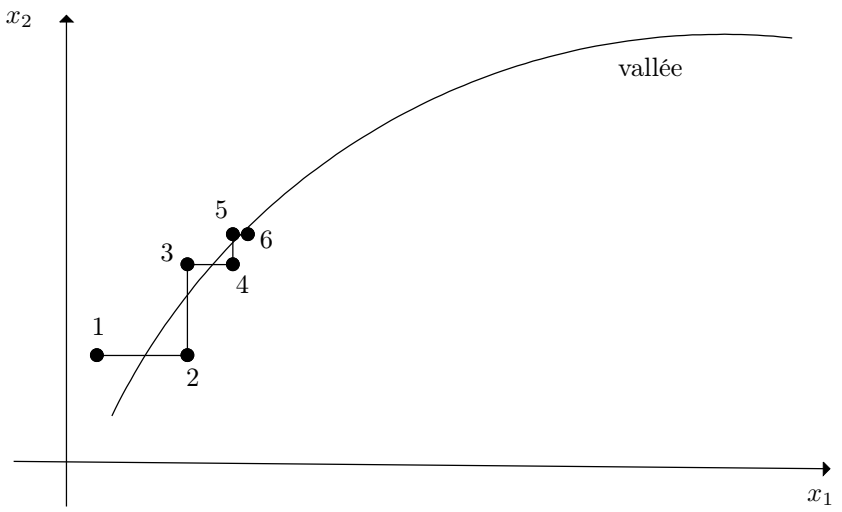


**Fig. 9.2** Mauvaise idée pour combiner des recherches unidimensionnelles

Une bien meilleure approche est l'*algorithme de Powell*, comme suit :

1. partant de $\mathbf{x}^k$, faire $n = \dim \mathbf{x}$ recherches unidimensionnelles successives dans des directions $\mathbf{d}^i$ linéairement indépendantes, $i = 1, \cdots, n$, pour obtenir $\mathbf{x}^{k+}$

(lors de la première itération, ces directions peuvent correspondre aux axes de l'espace de décision, comme dans la recherche cyclique) ;

2. faire une recherche unidimensionnelle de plus dans la direction moyenne des $n$ déplacements précédents

$$\mathbf{d} = \mathbf{x}^{k+} - \mathbf{x}^{k} \tag{9.131}$$

pour obtenir $\mathbf{x}^{k+1}$ ;

3. remplacer la *meilleure* des directions $\mathbf{d}^i$ en terme de réduction du coût par $\mathbf{d}$, incrémenter $k$ d'une unité et aller au pas 1.

Cette procédure est illustrée par la figure 9.3. Même si l'élimination au pas 3 de la direction de recherche qui a donné les meilleurs résultats peut heurter le sens de la justice du lecteur, cette décision contribue à maintenir l'indépendance linéaire des directions de recherche du pas 1, ce qui autorise tout changement de direction qui s'avérerait nécessaire après une longue séquence de déplacements presque colinéaires.

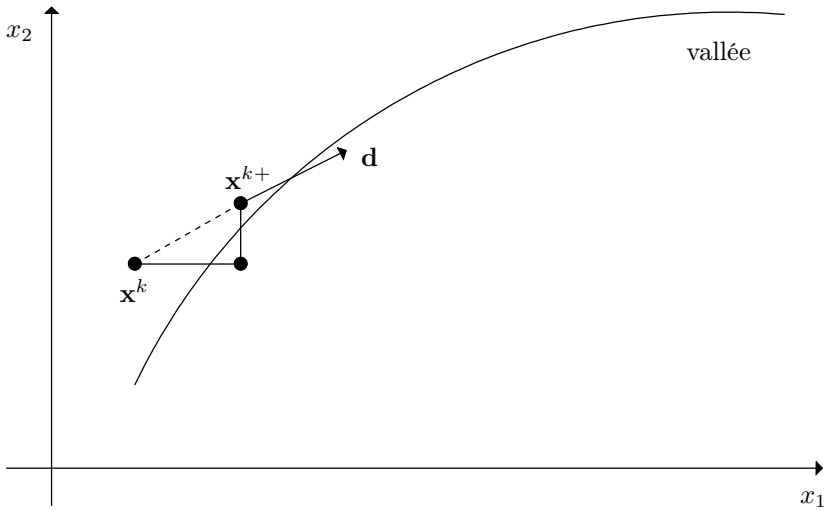


**Fig. 9.3** Algorithme de Powell pour combiner des recherches unidimensionnelles

### *9.3.4 Méthodes fondées sur un développement de Taylor du coût*

Supposons maintenant la fonction de coût suffisamment différentiable en $\mathbf{x}^k$ pour qu'on puisse calculer son développement de Taylor à l'ordre un ou deux au voisinage de $\mathbf{x}^k$. Ce développement peut alors être utilisé pour décider dans quelle direction la prochaine recherche unidimensionnelle doit être conduite.

*Remarque 9.16.* Pour établir les conditions théoriques d'optimalité en section 9.1, nous avons développé $J(\cdot)$ au voisinage de $\widehat{\mathbf{x}}$ alors qu'ici le développement est au voisinage de $\mathbf{x}^k$. □

#### 9.3.4.1 Méthode du gradient

Le développement *au premier ordre* de la fonction de coût au voisinage de $\mathbf{x}^k$ satisfait

$$J(\mathbf{x}^k + \delta\mathbf{x}) = J(\mathbf{x}^k) + \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\delta\mathbf{x} + \mathrm{o}(\|\delta\mathbf{x}\|). \tag{9.132}$$

La variation $\Delta J$ du coût résultant du déplacement $\delta\mathbf{x}$ est donc telle que

$$\Delta J = \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\delta\mathbf{x} + \mathrm{o}(\|\delta\mathbf{x}\|). \tag{9.133}$$

Quand $\delta\mathbf{x}$ est suffisamment petit pour que les termes d'ordre supérieur puissent être négligés, (9.133) suggère de prendre $\delta\mathbf{x}$ colinéaire avec le gradient en $\mathbf{x}^k$ et dans la direction opposée

$$\delta\mathbf{x} = -\lambda_k \mathbf{g}(\mathbf{x}^k), \quad \text{avec} \quad \lambda_k > 0. \tag{9.134}$$

Ceci correspond à la *méthode du gradient*

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \lambda_k \mathbf{g}(\mathbf{x}^k), \quad \text{avec} \quad \lambda_k > 0. \tag{9.135}$$

Si $J(\mathbf{x})$ était une altitude, alors son gradient pointerait dans la direction de la montée la plus raide. Ceci explique pourquoi la méthode du gradient est parfois appelée *steepest descent method*, ou méthode de la descente la plus raide.

On peut distinguer trois stratégies pour choisir $\lambda_k$ :

1. garder $\lambda_k$ à une valeur constante $\lambda$ ; c'est en général une mauvaise idée, car les valeurs convenables peuvent varier de plusieurs ordres de grandeur le long du chemin suivi par l'algorithme ; quand $\lambda$ est trop petit l'algorithme devient inutilement lent, tandis que quand $\lambda$ est trop grand il peut devenir instable à cause de la contribution des termes d'ordre supérieur ;
2. adapter $\lambda_k$ sur la base du comportement passé de l'algorithme ; si $J(\mathbf{x}^{k+1}) \leqslant J(\mathbf{x}^k)$ alors choisir $\lambda_{k+1}$ plus grand que $\lambda_k$, dans l'espoir d'accélérer la convergence, sinon repartir de $\mathbf{x}^k$ avec un $\lambda_k$ plus petit ;
3. choisir $\lambda_k$ par recherche unidimensionnelle pour minimiser $J(\mathbf{x}^k - \lambda_k \mathbf{g}(\mathbf{x}^k))$.

Quand $\lambda_k$ est optimal, les directions de recherche successives doivent être orthogonales :

$$\mathbf{g}(\mathbf{x}^{k+1}) \perp \mathbf{g}(\mathbf{x}^{k}), \tag{9.136}$$

et ceci est facile à vérifier.

*Remarque 9.17.* Plus généralement, pour tout algorithme d'optimisation itératif fondé sur une succession de recherches unidimensionnelles, il est instructif de tracer l'angle (non-orienté) $\theta(k)$ entre les directions de recherche successives $\mathbf{d}^k$ et $\mathbf{d}^{k+1}$,

$$\theta(k) = \arccos\left[\frac{(\mathbf{d}^{k+1})^{\mathrm{T}}\mathbf{d}^{k}}{\|\mathbf{d}^{k+1}\|_2 \cdot \|\mathbf{d}^{k}\|_2}\right], \tag{9.137}$$

en fonction de la valeur du compteur d'itérations $k$, ce qui reste simple quelle que soit la dimension de $\mathbf{x}$. Si $\theta(k)$ est répétitivement obtus, alors il se peut que l'algorithme soit en train d'osciller douloureusement, dans un mouvement en crabe autour d'une direction moyenne. Cette direction moyenne peut mériter une exploration (comme dans l'algorithme de Powell). Un angle répétitivement aigu suggère au contraire que les directions des déplacements sont cohérentes. □

La méthode du gradient a nombre d'avantages :
— elle est très simple à mettre en œuvre (pourvu qu'on sache comment calculer des gradients, voir la section 6.6),
— elle est robuste à des erreurs dans l'évaluation de $\mathbf{g}(\mathbf{x}^k)$ (avec une recherche unidimensionnelle efficace, la convergence vers un minimiseur local est garantie pourvu que l'erreur absolue sur la direction du gradient soit inférieure à $\pi/2$),
— son domaine de convergence vers un minimiseur donné est aussi grand qu'il peut l'être pour une telle méthode locale.

Sauf si la fonction de coût a des propriétés particulière comme la convexité (voir la section 10.7), la convergence vers un minimiseur global n'est pas garantie, mais cet inconvénient est partagé par toutes les méthodes itératives locales. Un inconvénient plus spécifique est le très grand nombre d'itérations qui peut s'avérer nécessaire pour obtenir une bonne approximation d'un minimiseur local. Après un départ en fanfare, la méthode du gradient devient en général de plus en plus lente, ce qui ne la rend utile que pour la phase initiale des recherches.

#### 9.3.4.2 Méthode de Newton

Considérons maintenant le développement au *second ordre* de la fonction de coût au voisinage de $\mathbf{x}^k$

$$J(\mathbf{x}^k + \delta\mathbf{x}) = J(\mathbf{x}^k) + \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^{\mathrm{T}}\mathbf{H}(\mathbf{x}^k)\delta\mathbf{x} + \mathrm{o}(\|\delta\mathbf{x}\|^2). \tag{9.138}$$

La variation $\Delta J$ du coût résultant du déplacement $\delta\mathbf{x}$ est telle que

$$\Delta J = \mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^{\mathrm{T}}\mathbf{H}(\mathbf{x}^k)\delta\mathbf{x} + \mathrm{o}(\|\delta\mathbf{x}\|^2). \tag{9.139}$$

Comme il n'y a pas de contrainte sur $\delta\mathbf{x}$, la condition nécessaire d'optimalité du premier ordre (9.6) se traduit par

$$\frac{\partial \Delta J}{\partial \delta \mathbf{x}}(\delta\widehat{\mathbf{x}}) = \mathbf{0}. \tag{9.140}$$

Quand $\delta\widehat{\mathbf{x}}$ est suffisamment petit pour que les termes d'ordre supérieur soient négligeables, (9.138) implique que

$$\frac{\partial \Delta J}{\partial \delta \mathbf{x}}(\delta\widehat{\mathbf{x}}) \approx \mathbf{H}(\mathbf{x}^k)\delta\widehat{\mathbf{x}} + \mathbf{g}(\mathbf{x}^k). \tag{9.141}$$

Ceci suggère de choisir le déplacement $\delta\widehat{\mathbf{x}}$ comme la solution du système d'équations linéaires

$$\mathbf{H}(\mathbf{x}^k)\delta\widehat{\mathbf{x}} = -\mathbf{g}(\mathbf{x}^k). \tag{9.142}$$

C'est la *méthode de Newton*, qu'on peut résumer par

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \mathbf{H}^{-1}(\mathbf{x}^k)\mathbf{g}(\mathbf{x}^k), \tag{9.143}$$

pourvu qu'on se rappelle qu'il serait inutilement compliqué d'inverser $\mathbf{H}(\mathbf{x}^k)$.

*Remarque 9.18.* La méthode de Newton pour l'optimisation est la même que la méthode de Newton pour la résolution de $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, puisque $\mathbf{H}(\mathbf{x})$ est la jacobienne de $\mathbf{g}(\mathbf{x})$. □

*Quand* elle converge vers un minimiseur (local), la méthode de Newton est incroyablement plus rapide que la méthode du gradient (il faut typiquement moins de dix itérations, au lieu de milliers). Même si chaque itération requiert plus de calculs, ceci est un net avantage. La convergence n'est cependant pas garantie, pour au moins deux raisons.

Premièrement, suivant le choix du vecteur initial $\mathbf{x}^0$, la méthode de Newton peut converger vers un maximiseur local ou un point en selle au lieu d'un minimiseur local, car elle se borne à rechercher $\mathbf{x}$ tel que la condition de stationnarité $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ soit satisfaite. Son domaine de convergence vers un minimiseur peut donc être significativement plus petit que celui de la méthode du gradient, comme illustré par la figure 9.4.

Deuxièmement, la *taille* du pas $\delta\widehat{\mathbf{x}}$ peut se révéler trop grande pour que les termes d'ordre plus élevé soient négligeables, même si la *direction* était appropriée. Ceci est facilement évité en introduisant un facteur d'amortissement positif $\lambda_k$ pour obtenir la *méthode de Newton amortie*

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \lambda_k \delta\widehat{\mathbf{x}}, \tag{9.144}$$

où $\delta\widehat{\mathbf{x}}$ reste calculé en résolvant (9.142). L'algorithme résultant peut être résumé en

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \lambda_k \mathbf{H}^{-1}(\mathbf{x}^k)\mathbf{g}(\mathbf{x}^k). \tag{9.145}$$

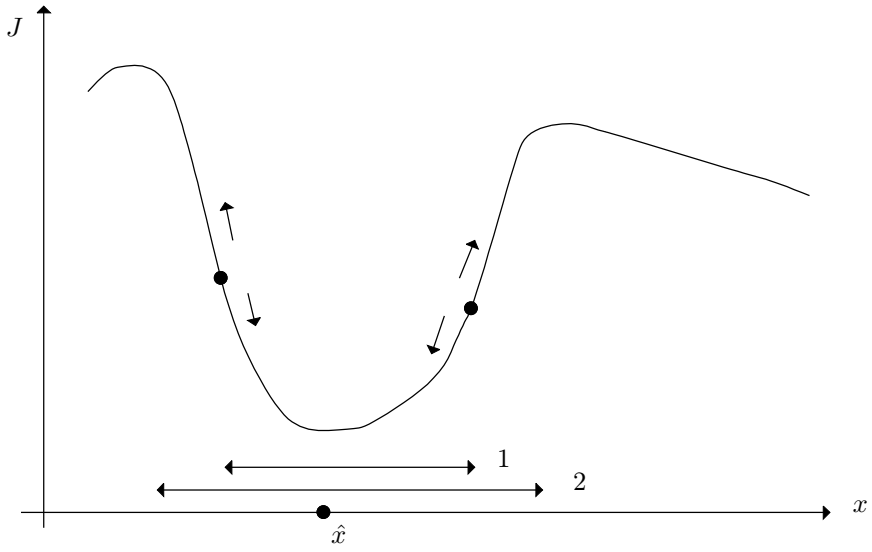


**Fig. 9.4** Le domaine de convergence de la méthode de Newton vers un minimiseur (1) est plus petit que celui de la méthode du gradient (2)

Le coefficient d'amortissement $\lambda_k$ peut être adapté ou optimisé par recherche unidimensionnelle, tout comme pour la méthode du gradient. Une différence importante est qu'on sait ici que la valeur nominale de $\lambda_k$ est égale à un, alors qu'une telle valeur nominale n'existe pas dans le cas de la méthode du gradient.

La méthode de Newton est particulièrement bien adaptée à la partie finale d'une recherche locale, quand la méthode du gradient est devenue trop lente pour être utile. Combiner un comportement initial proche de celui de la méthode du gradient avec un comportement final proche de celui de la méthode de Newton fait donc sens. Avant de décrire des tentatives dans cette direction, considérons un cas particulier important où la méthode de Newton peut être simplifiée de façon utile.

#### 9.3.4.3 Méthode de Gauss-Newton

La méthode de Gauss-Newton s'applique quand la fonction de coût peut s'exprimer comme la somme de $N \geqslant \dim \mathbf{x}$ termes scalaires quadratiques en une erreur

$$J(\mathbf{x}) = \sum_{l=1}^{N} w_l e_l^2(\mathbf{x}), \tag{9.146}$$

où les $w_l$ sont des poids positifs connus. L'erreur $e_l$ (aussi appelée *résidu*) peut, par exemple, être la différence entre des résultats de mesures $y_l$ et la sortie correspondante $y_\mathrm{m}(l,\mathbf{x})$ d'un modèle. Le gradient de la fonction de coût est alors

$$\mathbf{g}(\mathbf{x}) = \frac{\partial J}{\partial \mathbf{x}}(\mathbf{x}) = 2\sum_{l=1}^{N} w_l e_l(\mathbf{x}) \frac{\partial e_l}{\partial \mathbf{x}}(\mathbf{x}), \tag{9.147}$$

où $\frac{\partial e_l}{\partial \mathbf{x}}(\mathbf{x})$ est la *sensibilité au premier ordre* de l'erreur par rapport à $\mathbf{x}$. La hessienne du coût peut alors être calculée comme

$$\mathbf{H}(\mathbf{x}) = \frac{\partial \mathbf{g}}{\partial \mathbf{x}^\mathrm{T}}(\mathbf{x}) = 2\sum_{l=1}^{N} w_l \left[\frac{\partial e_l}{\partial \mathbf{x}}(\mathbf{x})\right]\left[\frac{\partial e_l}{\partial \mathbf{x}}(\mathbf{x})\right]^\mathrm{T} + 2\sum_{l=1}^{N} w_l e_l(\mathbf{x}) \frac{\partial^2 e_l}{\partial \mathbf{x}\partial \mathbf{x}^\mathrm{T}}(\mathbf{x}), \tag{9.148}$$

où $\frac{\partial^2 e_l}{\partial \mathbf{x}\partial \mathbf{x}^\mathrm{T}}(\mathbf{x})$ est la *sensibilité au second ordre* de l'erreur par rapport à $\mathbf{x}$. La méthode de Gauss-Newton amortie est obtenue en remplaçant $\mathbf{H}(\mathbf{x})$ dans la méthode de Newton amortie par l'approximation

$$\mathbf{H}_\mathrm{a}(\mathbf{x}) = 2\sum_{l=1}^{N} w_l \left[\frac{\partial e_l}{\partial \mathbf{x}}(\mathbf{x})\right]\left[\frac{\partial e_l}{\partial \mathbf{x}}(\mathbf{x})\right]^\mathrm{T}. \tag{9.149}$$

La méthode de Gauss-Newton amortie calcule donc

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \lambda_k \mathbf{d}^k, \tag{9.150}$$

où $\mathbf{d}^k$ est la solution du système linéaire

$$\mathbf{H}_\mathrm{a}(\mathbf{x}^k)\mathbf{d}^k = -\mathbf{g}(\mathbf{x}^k). \tag{9.151}$$

Le fait de remplacer $\mathbf{H}(\mathbf{x}^k)$ par $\mathbf{H}_\mathrm{a}(\mathbf{x}^k)$ a deux avantages. Le premier, évident, est que l'évaluation de la hessienne approchée $\mathbf{H}_\mathrm{a}(\mathbf{x})$ demande à peine plus de calcul que celle du gradient $\mathbf{g}(\mathbf{x})$, puisqu'on évite l'évaluation délicate des sensibilités au second ordre. Le second, plus inattendu, est que la méthode de Gauss-Newton amortie a le même domaine de convergence vers un minimiseur donné que la méthode du gradient, contrairement à la méthode de Newton. Ceci est dû au fait que $\mathbf{H}_\mathrm{a}(\mathbf{x}) \succ 0$ (sauf dans des cas pathologiques), de sorte que $\mathbf{H}_\mathrm{a}^{-1}(\mathbf{x}) \succ 0$. En conséquence, l'angle entre la direction de recherche $-\mathbf{g}(\mathbf{x}^k)$ de la méthode du gradient et la direction de recherche $-\mathbf{H}_\mathrm{a}^{-1}(\mathbf{x}^k)\mathbf{g}(\mathbf{x}^k)$ de la méthode de Gauss-Newton est inférieur à $\frac{\pi}{2}$ en valeur absolue.

Quand les résidus $e_l(\mathbf{x})$ sont de petite taille, la méthode de Gauss-Newton est beaucoup plus efficace que celle du gradient, pour un coût supplémentaire par itération limité. Ses performances tendent cependant à se détériorer quand la taille des résidus augmente, car la contribution de la partie négligée de la hessienne devient trop importante pour qu'on puisse l'ignorer [173]. Ceci est particulièrement vrai quand $e_l(\mathbf{x})$ est très non linéaire en $\mathbf{x}$, car la sensibilité au second ordre de

l'erreur est alors élevée. Dans une telle situation, on peut préférer une méthode de quasi-Newton, voir la section 9.3.4.5.

*Remarque 9.19.* Les fonctions de sensibilité peuvent être évaluées par différentiation automatique progressive, voir la section 6.6.4. □

*Remarque 9.20.* Quand $e_l = y_l - y_{\mathrm{m}}(l,\mathbf{x})$, la sensibilité au premier ordre de l'erreur satisfait

$$\frac{\partial}{\partial \mathbf{x}} e_l(\mathbf{x}) = -\frac{\partial}{\partial \mathbf{x}} y_{\mathrm{m}}(l,\mathbf{x}). \tag{9.152}$$

Si $y_{\mathrm{m}}(l,\mathbf{x})$ est obtenue en résolvant des équations différentielles ordinaires ou aux dérivées partielles, alors la sensibilité au premier ordre de la sortie du modèle $y_{\mathrm{m}}$ par rapport à $x_i$ peut être calculée en prenant la dérivée partielle au premier ordre des équations du modèle (y compris leurs conditions aux limites) par rapport à $x_i$ et en résolvant le système d'équations différentielles résultant. Voir l'exemple 9.7. En général, le calcul du vecteur de toutes les sensibilités au premier ordre en plus de la sortie du modèle demande donc de résoudre $(\dim \mathbf{x} + 1)$ systèmes d'équations différentielles. Ce nombre peut être réduit très significativement par application du principe de superposition pour les modèles décrits par des équations différentielles ordinaires, quand les sorties de ces modèles sont linéaires par rapport à leurs entrées et les conditions initiales sont nulles [244]. □

*Exemple 9.7.* Considérons le modèle différentiel

$$\begin{aligned} \dot{q}_1 &= -(x_1+x_3)q_1 + x_2 q_2, \\ \dot{q}_2 &= x_1 q_1 - x_2 q_2, \\ y_{\mathrm{m}}(t,\mathbf{x}) &= q_2(t,\mathbf{x}). \end{aligned} \tag{9.153}$$

avec les conditions initiales

$$q_1(0) = 1, \quad q_2(0) = 0. \tag{9.154}$$

Supposons qu'on veuille estimer le vecteur $\mathbf{x}$ de ses paramètres en minimisant

$$J(\mathbf{x}) = \sum_{i=1}^{N} [y(t_i) - y_{\mathrm{m}}(t_i,\mathbf{x})]^2, \tag{9.155}$$

où les valeurs numériques de $t_i$ et $y(t_i)$, $(i = 1,\cdots,N)$ sont connues et résultent d'expérimentation sur le système à modéliser. Le gradient et la hessienne approchée de la fonction de coût (9.155) peuvent être calculés à partir des sensibilités au premier ordre de $y_{\mathrm{m}}$ par rapport aux paramètres. Si $s_{j,k}$ est la sensibilité au premier ordre de $q_j$ par rapport à $x_k$,

$$s_{j,k}(t_i,\mathbf{x}) = \frac{\partial q_j}{\partial x_k}(t_i,\mathbf{x}), \tag{9.156}$$

alors le gradient de la fonction de coût est donné par

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} -2\sum_{i=1}^{N}[y(t_i) - q_2(t_i,\mathbf{x})]s_{2,1}(t_i,\mathbf{x}) \\ -2\sum_{i=1}^{N}[y(t_i) - q_2(t_i,\mathbf{x})]s_{2,2}(t_i,\mathbf{x}) \\ -2\sum_{i=1}^{N}[y(t_i) - q_2(t_i,\mathbf{x})]s_{2,3}(t_i,\mathbf{x}) \end{bmatrix},$$

et la hessienne approchée par

$$\mathbf{H}_{\mathrm{a}}(\mathbf{x}) = 2\sum_{i=1}^{N}\begin{bmatrix} s_{2,1}^2(t_i,\mathbf{x}) & s_{2,1}(t_i,\mathbf{x})s_{2,2}(t_i,\mathbf{x}) & s_{2,1}(t_i,\mathbf{x})s_{2,3}(t_i,\mathbf{x}) \\ s_{2,2}(t_i,\mathbf{x})s_{2,1}(t_i,\mathbf{x}) & s_{2,2}^2(t_i,\mathbf{x}) & s_{2,2}(t_i,\mathbf{x})s_{2,3}(t_i,\mathbf{x}) \\ s_{2,3}(t_i,\mathbf{x})s_{2,1}(t_i,\mathbf{x}) & s_{2,3}(t_i,\mathbf{x})s_{2,2}(t_i,\mathbf{x}) & s_{2,3}^2(t_i,\mathbf{x}) \end{bmatrix}.$$

Différentions (9.153) successivement par rapport à $x_1$, $x_2$ et $x_3$, pour obtenir

$$\begin{aligned} \dot{s}_{1,1} &= -(x_1+x_3)s_{1,1} + x_2 s_{2,1} - q_1, \\ \dot{s}_{2,1} &= x_1 s_{1,1} - x_2 s_{2,1} + q_1, \\ \dot{s}_{1,2} &= -(x_1+x_3)s_{1,2} + x_2 s_{2,2} + q_2, \\ \dot{s}_{2,2} &= x_1 s_{1,2} - x_2 s_{2,2} - q_2, \\ \dot{s}_{1,3} &= -(x_1+x_3)s_{1,3} + x_2 s_{2,3} - q_1, \\ \dot{s}_{2,3} &= x_1 s_{1,3} - x_2 s_{2,3}. \end{aligned} \tag{9.157}$$

Puisque $\mathbf{q}(0)$ ne dépend pas de $\mathbf{x}$, la condition initiale de chacune des sensibilités au premier ordre est nulle

$$s_{1,1}(0) = s_{2,1}(0) = s_{1,2}(0) = s_{2,2}(0) = s_{1,3}(0) = s_{2,3}(0) = 0. \tag{9.158}$$

La solution numérique du système de huit équations différentielles ordinaires du premier ordre (9.153, 9.157) pour les conditions initiales (9.154, 9.158) peut être obtenue par des méthodes décrites au chapitre 12. On peut préférer résoudre trois systèmes de quatre équations différentielles ordinaires du premier ordre, qui calculent chacun $x_1$, $x_2$ et les deux fonctions de sensibilité associées à l'un des trois paramètres. □

*Remarque 9.21.* Définissons le vecteur d'erreur comme

$$\mathbf{e}(\mathbf{x}) = [e_1(\mathbf{x}), e_2(\mathbf{x}), \cdots, e_N(\mathbf{x})]^{\mathrm{T}}, \tag{9.159}$$

et supposons que les $w_l$ ont été rendus égaux à un par la méthode décrite en section 9.2.1. L'équation (9.151) peut alors se réécrire

$$\mathbf{J}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{J}(\mathbf{x}^k)\mathbf{d}^k = -\mathbf{J}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{e}(\mathbf{x}^k), \tag{9.160}$$

où $\mathbf{J}(\mathbf{x})$ est la jacobienne du vecteur d'erreur

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{e}}{\partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}). \tag{9.161}$$

L'équation (9.160) est l'équation normale du problème de *moindres carrés linéaires*

$$\mathbf{d}^k = \arg\min_{\mathbf{d}} \|\mathbf{J}(\mathbf{x}^k)\mathbf{d}^k + \mathbf{e}(\mathbf{x}^k)\|_2^2, \tag{9.162}$$

et on peut obtenir une meilleure solution pour $\mathbf{d}^k$ en utilisant l'une des méthodes recommandées en section 9.2, par exemple via une factorisation QR de $\mathbf{J}(\mathbf{x}^k)$. Une SVD de $\mathbf{J}(\mathbf{x}^k)$ est plus compliquée mais permet de surveiller très facilement le conditionnement du problème local à résoudre. Quand la situation devient désespérée, elle permet aussi une régularisation. □

#### 9.3.4.4 Méthode de Levenberg et Marquardt

La *méthode de Levenberg* [141] est une première tentative de combiner les meilleures propriétés des méthodes du gradient et de Gauss-Newton pour la minimisation d'une somme de carrés. Le déplacement $\delta\widehat{\mathbf{x}}$ à l'itération $k$ est obtenu par résolution du système d'équations linéaires

$$\left[\mathbf{H}_{\mathrm{a}}(\mathbf{x}^k) + \mu_k \mathbf{I}\right] \delta\widehat{\mathbf{x}} = -\mathbf{g}(\mathbf{x}^k), \tag{9.163}$$

où la valeur donnée au scalaire réel $\mu_k > 0$ peut être choisie par minimisation unidimensionnelle de $J(\mathbf{x}^k + \delta\widehat{\mathbf{x}})$, vue comme une fonction de $\mu_k$.

Quand $\mu_k$ tend vers zéro, cette méthode se comporte comme une méthode de Gauss-Newton (sans amortissement), tandis que quand $\mu_k$ tend vers l'infini elle se comporte comme une méthode de gradient avec un pas dont la taille tend vers zéro.

Pour améliorer le conditionnement, Marquardt a suggéré dans [150] d'appliquer la même idée à une version mise à l'échelle de (9.163) :

$$\left[\mathbf{H}_{\mathrm{a}}^{\mathrm{s}} + \mu_k \mathbf{I}\right] \boldsymbol{\delta}^{\mathrm{s}} = -\mathbf{g}^{\mathrm{s}}, \tag{9.164}$$

avec

$$h_{i,j}^{\mathrm{s}} = \frac{h_{i,j}}{\sqrt{h_{i,i}}\sqrt{h_{j,j}}}, \quad g_i^{\mathrm{s}} = \frac{g_i}{\sqrt{h_{i,i}}} \quad \text{et} \quad \delta_i^{\mathrm{s}} = \frac{\delta\widehat{x}_i}{\sqrt{h_{i,i}}}, \tag{9.165}$$

où $h_{i,j}$ est l'élément de $\mathbf{H}_{\mathrm{a}}(\mathbf{x}^k)$ en position $(i,j)$, $g_i$ est le $i$-ème élément de $\mathbf{g}(\mathbf{x}^k)$ et $\delta\widehat{x}_i$ est le $i$-ème élément de $\delta\widehat{\mathbf{x}}$. Comme $h_{i,i} > 0$, on peut toujours faire cette mise à l'échelle. La $i$-ème ligne de (9.164) peut alors s'écrire

$$\sum_{j=1}^{n} \left(h_{i,j}^{\mathrm{s}} + \mu_k \delta_{i,j}\right) \delta_j^{\mathrm{s}} = -g_i^{\mathrm{s}}, \tag{9.166}$$

où $\delta_{i,j} = 1$ si $i = j$ et $\delta_{i,j} = 0$ sinon. Pour les variables de départ, (9.166) se traduit par

$$\sum_{j=1}^{n} \left(h_{i,j} + \mu_k \delta_{i,j} h_{i,i}\right) \delta\widehat{x}_j = -g_i. \tag{9.167}$$

En d'autres termes,

$$\left[\mathbf{H}_{\mathrm{a}}(\mathbf{x}^k) + \mu_k \cdot \operatorname{diag} \mathbf{H}_{\mathrm{a}}(\mathbf{x}^k)\right] \delta\widehat{\mathbf{x}} = -\mathbf{g}(\mathbf{x}^k), \tag{9.168}$$

où diag $\mathbf{H}_{\mathrm{a}}$ est une matrice diagonale avec les mêmes éléments diagonaux que $\mathbf{H}_{\mathrm{a}}$. C'est la *méthode de Levenberg et Marquardt*, utilisée en routine dans les logiciels d'estimation de paramètres non linéaire.

Un désavantage de cette méthode est qu'il faut résoudre un nouveau système d'équations linéaires chaque fois que la valeur de $\mu_k$ change, ce qui rend l'optimisation de $\mu_k$ significativement plus coûteuse qu'avec les recherches unidimensionnelles usuelles. C'est pourquoi on utilise en général une stratégie adaptative pour régler $\mu_k$ sur la base du comportement passé. Voir [150] pour plus de détails.

La méthode de Levenberg et Marquardt est l'une de celles mises en œuvre dans `lsqnonlin`, qui fait partie de la *MATLAB Optimization Toolbox*.

#### 9.3.4.5 Méthodes de quasi-Newton

Les méthodes de quasi-Newton [51] approximent la fonction de coût $J(\mathbf{x})$ après la $k$-ème itération par une fonction quadratique du vecteur de décision $\mathbf{x}$

$$J_{\mathrm{q}}(\mathbf{x}) = J(\mathbf{x}^k) + \mathbf{g}_{\mathrm{q}}^{\mathrm{T}}(\mathbf{x}^k)(\mathbf{x}-\mathbf{x}^k) + \frac{1}{2}(\mathbf{x}-\mathbf{x}^k)^{\mathrm{T}}\mathbf{H}_{\mathrm{q}}(\mathbf{x}-\mathbf{x}^k), \tag{9.169}$$

où

$$\mathbf{g}_{\mathrm{q}}(\mathbf{x}^k) = \frac{\partial J_{\mathrm{q}}}{\partial \mathbf{x}}(\mathbf{x}^k) \tag{9.170}$$

et

$$\mathbf{H}_{\mathrm{q}} = \frac{\partial^2 J_{\mathrm{q}}}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}. \tag{9.171}$$

Comme l'approximation est quadratique, sa hessienne $\mathbf{H}_{\mathrm{q}}$ ne dépend pas de $\mathbf{x}$, ce qui permet d'estimer $\mathbf{H}_{\mathrm{q}}^{-1}$ à partir du comportement de l'algorithme sur une série d'itérations.

*Remarque 9.22.* La fonction de coût $J(\mathbf{x})$ n'est bien sûr pas exactement quadratique en $\mathbf{x}$ (sinon, il serait bien préférable d'utiliser la méthode des moindres carrés linéaires de la section 9.2), mais une approximation quadratique devient en général de plus en plus satisfaisante quand $\mathbf{x}^k$ s'approche d'un minimiseur. □

La mise à jour de l'estimée de $\widehat{\mathbf{x}}$ est directement inspirée de la méthode de Newton amortie (9.145), en remplaçant $\mathbf{H}^{-1}$ par l'estimée $\mathbf{M}_k$ de $\mathbf{H}_{\mathrm{q}}^{-1}$ à l'itération $k$ :

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \lambda_k \mathbf{M}_k \mathbf{g}(\mathbf{x}^k), \tag{9.172}$$

où $\lambda_k$ est une fois encore obtenu par recherche unidimensionnelle.

Différentions $J_{\mathrm{q}}(\mathbf{x})$ comme donnée par (9.169) une fois par rapport à $\mathbf{x}$ et évaluons le résultat en $\mathbf{x}^{k+1}$ pour obtenir

$$\mathbf{g}_{\mathrm{q}}(\mathbf{x}^{k+1}) = \mathbf{g}_{\mathrm{q}}(\mathbf{x}^k) + \mathbf{H}_{\mathrm{q}}(\mathbf{x}^{k+1} - \mathbf{x}^k), \tag{9.173}$$

de sorte que

$$\mathbf{H}_{\mathrm{q}}\Delta\mathbf{x} = \Delta\mathbf{g}_{\mathrm{q}}, \tag{9.174}$$

où

$$\Delta\mathbf{g}_{\mathrm{q}} = \mathbf{g}_{\mathrm{q}}(\mathbf{x}^{k+1}) - \mathbf{g}_{\mathrm{q}}(\mathbf{x}^{k}) \tag{9.175}$$

et

$$\Delta\mathbf{x} = \mathbf{x}^{k+1} - \mathbf{x}^{k}. \tag{9.176}$$

L'équation (9.174) suggère l'équation de quasi-Newton

$$\tilde{\mathbf{H}}_{k+1}\Delta\mathbf{x} = \Delta\mathbf{g}, \tag{9.177}$$

avec $\widetilde{\mathbf{H}}_{k+1}$ l'approximation de la hessienne à l'itération $k+1$ et $\Delta\mathbf{g}$ la variation du gradient de la vraie fonction de coût entre les itérations $k$ et $k+1$. Ceci correspond à (7.52), où le rôle de la fonction $\mathbf{f}(\cdot)$ est tenu par la fonction gradient $\mathbf{g}(\cdot)$.

En posant $\mathbf{M}_{k+1} = \tilde{\mathbf{H}}_{k+1}^{-1}$, on peut réécrire (9.177) comme

$$\mathbf{M}_{k+1}\Delta\mathbf{g} = \Delta\mathbf{x}. \tag{9.178}$$

Cette dernière équation est utilisée pour mettre à jour $\mathbf{M}_k$ par

$$\mathbf{M}_{k+1} = \mathbf{M}_k + \mathbf{C}_k. \tag{9.179}$$

Le terme correctif $\mathbf{C}_k$ doit donc satisfaire

$$\mathbf{C}_k\Delta\mathbf{g} = \Delta\mathbf{x} - \mathbf{M}_k\Delta\mathbf{g}. \tag{9.180}$$

Puisque $\mathbf{H}^{-1}$ est symétrique, on choisit une estimée initiale $\mathbf{M}_0$ et des matrices de correction $\mathbf{C}_k$ symétriques. C'est une différence importante avec la méthode de Broyden présentée en section 7.4.3, puisque la jacobienne d'une fonction vectorielle générique n'est pas symétrique.

Les méthodes de quasi-Newton diffèrent par leurs expressions pour $\mathbf{C}_k$. La seule correction symétrique de rang un est celle de [32] :

$$\mathbf{C}_k = \frac{(\Delta\mathbf{x} - \mathbf{M}_k\Delta\mathbf{g})(\Delta\mathbf{x} - \mathbf{M}_k\Delta\mathbf{g})^{\mathrm{T}}}{(\Delta\mathbf{x} - \mathbf{M}_k\Delta\mathbf{g})^{\mathrm{T}}\Delta\mathbf{g}}, \tag{9.181}$$

qui suppose que $(\Delta\mathbf{x} - \mathbf{M}_k\Delta\mathbf{g})^{\mathrm{T}}\Delta\mathbf{g} \neq 0$. Il est trivial de vérifier qu'elle satisfait (9.180), mais les matrices $\mathbf{M}_k$ ainsi générées ne sont pas toujours définies positives.

La plupart des méthodes de quasi-Newton appartiennent à une famille définie dans [32] et donneraient des résultats identiques si les calculs étaient menés à bien de façon exacte [54]. Elles diffèrent cependant dans leur robustesse aux erreurs dans les évaluations de gradients. La plus populaire est BFGS (un acronyme de Broyden, Fletcher, Golfarb et Shanno, qui l'ont publiée indépendamment). BFGS utilise la correction

$$\mathbf{C}_k = \mathbf{C}_1 + \mathbf{C}_2, \tag{9.182}$$

où

$$\mathbf{C}_1 = \left(1 + \frac{\Delta\mathbf{g}^{\mathrm{T}}\mathbf{M}_k\Delta\mathbf{g}}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{g}}\right)\frac{\Delta\mathbf{x}\Delta\mathbf{x}^{\mathrm{T}}}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{g}} \tag{9.183}$$

et

$$\mathbf{C}_2 = -\frac{\Delta\mathbf{x}\Delta\mathbf{g}^{\mathrm{T}}\mathbf{M}_k + \mathbf{M}_k\Delta\mathbf{g}\Delta\mathbf{x}^{\mathrm{T}}}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{g}}. \tag{9.184}$$

Il est facile de vérifier que cette mise à jour satisfait (9.180) et peut aussi s'écrire

$$\mathbf{M}_{k+1} = \left(\mathbf{I} - \frac{\Delta\mathbf{x}\Delta\mathbf{g}^{\mathrm{T}}}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{g}}\right)\mathbf{M}_k\left(\mathbf{I} - \frac{\Delta\mathbf{g}\Delta\mathbf{x}^{\mathrm{T}}}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{g}}\right) + \frac{\Delta\mathbf{x}\Delta\mathbf{x}^{\mathrm{T}}}{\Delta\mathbf{x}^{\mathrm{T}}\Delta\mathbf{g}}. \tag{9.185}$$

Il est aussi facile de vérifier que

$$\Delta\mathbf{g}^{\mathrm{T}}\mathbf{M}_{k+1}\Delta\mathbf{g} = \Delta\mathbf{g}^{\mathrm{T}}\Delta\mathbf{x}, \tag{9.186}$$

de sorte que la recherche unidimensionnelle sur $\lambda_k$ doit assurer

$$\Delta\mathbf{g}^{\mathrm{T}}\Delta\mathbf{x} > 0 \tag{9.187}$$

pour que $\mathbf{M}_{k+1}$ soit positive définie. Tel est le cas quand on impose des conditions de Wolf fortes lors du calcul de $\lambda_k$ [69]. D'autres options sont de

— *geler* $\mathbf{M}$ chaque fois que $\Delta\mathbf{g}^{\mathrm{T}}\Delta\mathbf{x} \leqslant 0$ (en posant $\mathbf{M}_{k+1} = \mathbf{M}_k$),

— *redémarrer périodiquement*, ce qui impose à $\mathbf{M}_k$ d'être égale à la matrice identité toutes les $\dim\mathbf{x}$ itérations. (Si la fonction de coût était vraiment quadratique en $\mathbf{x}$ et les calculs étaient exacts, la convergence prendrait au plus $\dim\mathbf{x}$ itérations.)

La valeur initiale de l'approximation de $\mathbf{H}^{-1}$ est

$$\mathbf{M}_0 = \mathbf{I}, \tag{9.188}$$

de sorte que la méthode commence comme la méthode du gradient.

Par rapport à la méthode de Newton, la méthode de quasi-Newton résultante a plusieurs avantages :

— il n'est pas nécessaire de calculer la hessienne $\mathbf{H}$ de la fonction de coût à chaque itération,

— il n'est pas nécessaire de résoudre un système d'équations linéaires à chaque itération, puisqu'une approximation de $\mathbf{H}^{-1}$ est calculée,

— le domaine de convergence vers un minimiseur est le même que pour la méthode du gradient (pourvu qu'on prenne les mesures nécessaires pour assurer que $\mathbf{M}_k \succ 0, \forall k \geqslant 0$),

— l'estimée de l'inverse de la hessienne peut être utilisée pour étudier le conditionnement local du problème et pour évaluer la précision avec laquelle le minimiseur $\widehat{\mathbf{x}}$ a été évalué. Ceci est important quand on estime des paramètres physiques à partir de données expérimentales [244].

Il faut être aussi conscient, cependant, des désavantages suivants :

— les méthodes de quasi-Newton sont plutôt sensibles à des erreurs sur les calculs de gradients, puisqu'elles utilisent des différences entre des valeurs du

gradient pour mettre à jour les estimées de $\mathbf{H}^{-1}$ ; elles sont plus sensibles à de telles erreurs que la méthode de Gauss-Newton, par exemple ;

— mettre à jour la matrice $\mathbf{M}_k$ de dimensions $\dim \mathbf{x} \times \dim \mathbf{x}$ à chaque itération peut s'avérer déraisonnable si $\dim \mathbf{x}$ est très grand comme, par exemple, en traitement d'images.

Le dernier de ces désavantages est l'une des raisons principales pour utiliser à la place une méthode de gradients conjugués.

Les méthodes de quasi-Newton sont très largement utilisées, et prêtes à l'emploi dans les bibliothèques de programmes de calcul scientifique. BFGS est l'une de celles mises en œuvre dans `fminunc`, qui fait partie de la *MATLAB Optimization Toolbox*.

#### 9.3.4.6 Méthodes de gradients conjugués

Comme les méthodes de quasi-Newton, les méthodes de gradients conjugués [218, 91] approximent la fonction de coût par une fonction quadratique du vecteur de décision donnée par (9.169). Mais contrairement aux méthodes de quasi-Newton, elles ne cherchent pas à estimer $\mathbf{H}_\mathrm{q}$ ou son inverse, ce qui les rend particulièrement intéressantes quand la dimension de $\mathbf{x}$ est très grande.

L'estimée du minimiseur est mise à jour par recherche unidimensionnelle dans une direction $\mathbf{d}^k$, suivant

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \lambda_k \mathbf{d}^k. \tag{9.189}$$

Si $\mathbf{d}^k$ était calculée par la méthode de Newton, alors elle satisferait

$$\mathbf{d}^k = -\mathbf{H}^{-1}(\mathbf{x}^k)\mathbf{g}(\mathbf{x}^k), \tag{9.190}$$

et l'optimisation de $\lambda_k$ impliquerait que

$$\mathbf{g}^\mathrm{T}(\mathbf{x}^{k+1})\mathbf{d}^k = 0. \tag{9.191}$$

Comme $\mathbf{H}(\mathbf{x}^k)$ est symétrique, (9.190) implique que

$$\mathbf{g}^\mathrm{T}(\mathbf{x}^{k+1}) = -(\mathbf{d}^{k+1})^\mathrm{T}\mathbf{H}(\mathbf{x}^{k+1}), \tag{9.192}$$

de sorte que (9.191) se traduit par

$$(\mathbf{d}^{k+1})^\mathrm{T}\mathbf{H}(\mathbf{x}^{k+1})\mathbf{d}^k = 0. \tag{9.193}$$

Les directions de recherche successives de la méthode de Newton amortie de façon optimale sont donc *conjuguées* par rapport à la hessienne. Les méthodes de gradients conjugués visent à obtenir la même propriété par rapport à une approximation $\mathbf{H}_\mathrm{q}$ de cette hessienne. Comme les directions de recherche considérées ne sont pas des gradients, parler de "gradients conjugués" est trompeur, mais imposé par la tradition.

Un membre fameux de la famille des méthodes de gradients conjugués est la méthode de Polack-Ribière [183, 184], qui choisit

$$\mathbf{d}^{k+1} = -\mathbf{g}(\mathbf{x}^{k+1}) + \beta_k^{\mathrm{PR}}\, \mathbf{d}^k, \tag{9.194}$$

où

$$\beta_k^{\mathrm{PR}} = \frac{[\mathbf{g}(\mathbf{x}^{k+1}) - \mathbf{g}(\mathbf{x}^k)]^{\mathrm{T}} \mathbf{g}(\mathbf{x}^{k+1})}{\mathbf{g}^{\mathrm{T}}(\mathbf{x}^k)\mathbf{g}(\mathbf{x}^k)}. \tag{9.195}$$

Si la fonction de coût était vraiment donnée par (9.169), alors cette stratégie assurerait la conjugaison des directions $\mathbf{d}^{k+1}$ et $\mathbf{d}^k$ par rapport à $\mathbf{H}_{\mathrm{q}}$, *bien que* $\mathbf{H}_{\mathrm{q}}$ *ne soit ni connue ni estimée*, ce qui est un avantage considérable pour des problèmes de grande taille. La méthode est initialisée en prenant

$$\mathbf{d}^0 = -\mathbf{g}(\mathbf{x}^0). \tag{9.196}$$

Elle démarre donc comme une méthode de gradient. Tout comme avec les méthodes de quasi-Newton, une stratégie de redémarrage périodique peut être mise en œuvre, avec $\mathbf{d}^k$ pris égal à $\mathbf{g}(\mathbf{x}^k)$ toutes les $\dim \mathbf{x}$ itérations.

La satisfaction de conditions de Wolfe fortes durant la recherche unidimensionnelle ne garantit cependant pas que $\mathbf{d}^{k+1}$ telle que calculée avec la méthode de Polack-Ribière soit toujours une direction de descente [173]. Pour remédier à ce problème, il suffit de remplacer $\beta_k^{\mathrm{PR}}$ dans (9.194) par

$$\beta_k^{\mathrm{PR+}} = \max(\beta^{\mathrm{PR}}, 0). \tag{9.197}$$

Le principal désavantage des gradients conjugués par rapport à quasi-Newton est que l'inverse de la hessienne ne soit pas estimée. On peut donc préférer quasi-Newton si la dimension de $\mathbf{x}$ est suffisamment petite et si l'on souhaite évaluer le conditionnement local du problème d'optimisation ou caractériser l'incertitude sur $\widehat{\mathbf{x}}$.

*Exemple 9.8. Une application remarquable*

Comme déjà mentionné en section 3.7.2.2, les gradients conjugués sont utilisés pour résoudre de *grands* systèmes d'équations linéaires

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{9.198}$$

avec $\mathbf{A}$ symétrique et définie positive. De tels systèmes peuvent, par exemple, correspondre aux équations normales des moindres carrés linéaires. Résoudre (9.198) équivaut à minimiser le carré d'une norme quadratique convenablement pondérée

$$J(\mathbf{x}) = ||\mathbf{A}\mathbf{x} - \mathbf{b}||^2_{\mathbf{A}^{-1}} = (\mathbf{A}\mathbf{x} - \mathbf{b})^{\mathrm{T}} \mathbf{A}^{-1} (\mathbf{A}\mathbf{x} - \mathbf{b}) \tag{9.199}$$

$$= \mathbf{b}^{\mathrm{T}} \mathbf{A}^{-1} \mathbf{b} - 2\mathbf{b}^{\mathrm{T}}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}, \tag{9.200}$$

ce qui équivaut encore à minimiser

$$J(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - 2\mathbf{b}^{\mathrm{T}}\mathbf{x}. \tag{9.201}$$

La fonction de coût (9.201) est exactement quadratique, de sorte que sa hessienne ne dépend pas de $\mathbf{x}$, et que l'utilisation d'une méthode de gradients conjugués ne se traduit pas aucune approximation. Le gradient de la fonction de coût, utilisé par la méthode, est facile à calculer comme

$$\mathbf{g}(\mathbf{x}) = 2(\mathbf{A}\mathbf{x} - \mathbf{b}). \tag{9.202}$$

Une bonne approximation de la solution est souvent obtenue avec cette approche en bien moins des $\dim \mathbf{x}$ itérations théoriquement nécessaires. □

#### 9.3.4.7 Vitesses de convergence et complexité

Quand $\mathbf{x}^k$ s'approche suffisamment d'un minimiseur $\widehat{\mathbf{x}}$ (qui peut être local ou global), il devient possible d'étudier la vitesse de convergence (asymptotique) des principales méthodes itératives d'optimisation considérées jusqu'ici [155, 173, 51]. Nous supposons dans cette section $J(\cdot)$ deux fois continûment différentiable et $\mathbf{H}(\widehat{\mathbf{x}})$ symétrique définie positive, de sorte que toutes ses valeurs propres sont réelles et strictement positives.

Une *méthode de gradient* avec optimisation de la taille du pas a une vitesse de convergence *linéaire*, car

$$\limsup_{k\to\infty} \frac{\|\mathbf{x}^{k+1} - \widehat{\mathbf{x}}\|}{\|\mathbf{x}^k - \widehat{\mathbf{x}}\|} = \alpha, \quad \text{avec} \quad \alpha < 1. \tag{9.203}$$

Son taux de convergence $\alpha$ est tel que

$$\alpha \leqslant \left(\frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}}\right)^2, \tag{9.204}$$

où $\lambda_{\max}$ et $\lambda_{\min}$ sont la plus grande et la plus petite des valeurs propres de $\mathbf{H}(\widehat{\mathbf{x}})$, qui sont aussi sa plus grande et sa plus petite valeur singulière. La situation la plus favorable est quand toutes les valeurs propres de $\mathbf{H}(\widehat{\mathbf{x}})$ sont égales, de sorte que $\lambda_{\max} = \lambda_{\min}$, cond $\mathbf{H}(\widehat{\mathbf{x}}) = 1$ et $\alpha = 0$. Quand $\lambda_{\max} \gg \lambda_{\min}$, cond $\mathbf{H}(\widehat{\mathbf{x}}) \gg 1$, $\alpha$ est proche de un et la convergence devient très lente.

La *méthode de Newton* a une vitesse de convergence *quadratique*, pourvu que $\mathbf{H}(\cdot)$ satisfasse une condition de Lipschitz en $\widehat{\mathbf{x}}$, c'est à dire qu'il existe $\kappa$ tel que

$$\forall \mathbf{x}, \quad \|\mathbf{H}(\mathbf{x}) - \mathbf{H}(\widehat{\mathbf{x}})\| \leqslant \kappa \|\mathbf{x} - \widehat{\mathbf{x}}\|. \tag{9.205}$$

Ceci est *bien* mieux qu'une vitesse de convergence linéaire. Tant que l'effet des erreurs dues aux arrondis reste négligeable, le nombre de digits décimaux corrects dans $\mathbf{x}^k$ double approximativement à chaque itération.

La vitesse de convergence des méthodes de *Gauss-Newton* et de *Levenberg et Marquardt* se situe quelque part entre le linéaire et le quadratique, suivant la qualité de l'approximation de la hessienne, qui dépend elle-même de la taille des résidus.

Quand cette taille est suffisamment petite, la convergence est quadratique, mais elle devient linéaire quand les résidus sont trop grands.

Les *méthodes de quasi-Newton* ont une vitesse de convergence *superlinéaire* , avec

$$\limsup_{k\to\infty} \frac{\|\mathbf{x}^{k+1} - \widehat{\mathbf{x}}\|}{\|\mathbf{x}^k - \widehat{\mathbf{x}}\|} = 0. \tag{9.206}$$

Les *méthodes de gradients conjugués* ont elles aussi une vitesse de convergence *superlinéaire, mais sur* $\dim \mathbf{x}$ *itérations*. Elles requièrent donc approximativement $\dim \mathbf{x}$ fois plus d'itérations que des méthodes de quasi-Newton pour obtenir le même comportement asymptotique. Avec un redémarrage périodique toutes les $n = \dim \mathbf{x}$ itérations, les méthodes de gradients conjugués peuvent même atteindre une convergence quadratique sur $n$ pas, c'est à dire

$$\limsup_{k\to\infty} \frac{\|\mathbf{x}^{k+n} - \widehat{\mathbf{x}}\|}{\|\mathbf{x}^k - \widehat{\mathbf{x}}\|^2} = \alpha < \infty. \tag{9.207}$$

(En pratique, le redémarrage peut ne jamais avoir lieu si $n$ est suffisamment grand.)

*Remarque 9.23.* Bien sûr, les erreurs d'arrondi limitent la précision avec laquelle $\widehat{\mathbf{x}}$ peut être évalué par chacune de ces méthodes. □

*Remarque 9.24.* Ces résultats ne disent rien du comportement non asymptotique. Une méthode de gradient peut encore être beaucoup plus efficace que la méthode de Newton dans la phase initiale des recherches. □

La complexité doit aussi être prise en considération pour le choix d'une méthode. Si l'on peut négliger l'effort requis pour évaluer la fonction de coût et son gradient (plus sa hessienne pour la méthode de Newton), alors une itération de Newton requiert $O(n^3)$ flops, à comparer avec $O(n^2)$ flops pour une itération de quasi-Newton et $O(n)$ flops pour une itération de gradients conjugués. Sur un problème de grande taille, une itération de gradients conjugués demande donc beaucoup moins de calculs et de mémoire qu'une itération de quasi-Newton, qui elle-même demande beaucoup moins de calculs qu'une itération de Newton.

#### 9.3.4.8 D'où partir et quand s'arrêter ?

La plupart de ce qui a été dit dans les sections 7.5 et 7.6 reste valide. Quand la fonction de coût est convexe et différentiable, il n'y a qu'un minimiseur local, qui est aussi global, et les méthodes décrites jusqu'ici devraient converger vers ce minimiseur de n'importe quel point initial $\mathbf{x}^0$. Dans le cas contraire, il reste conseillé d'utiliser une stratégie *multistart*, à moins qu'on ne puisse s'offrir qu'une seule minimisation locale (disposer d'un point initial suffisamment bon devient alors crucial). En principe, la recherche locale devrait s'arrêter quand toutes les composantes du gradient de la fonction de coût sont nulles, de sorte que les critères d'arrêt sont similaires à ceux utilisés quand on résout des systèmes d'équations non linéaires.

### *9.3.5 Une méthode qui peut traiter des coûts non différentiables*

Aucune des méthodes fondées sur un développement de Taylor n'est applicable si la fonction de coût $J(\cdot)$ n'est pas différentiable. Même quand $J(\cdot)$ est différentiable presque partout, par exemple quand c'est une somme de valeurs absolues d'erreurs différentiables comme dans (8.15), ces méthodes se jetteront en général sur des points où elles ne sont plus valides.

Nombre d'approches sophistiquées ont été développées pour la minimisation de fonctions de coût non différentiables, fondées par exemple sur la notion de sous-gradient [220, 14, 170], mais elles ne seront pas abordées ici. Cette section ne présente qu'une approche, la célèbre *méthode du simplexe de Nelder et Mead* [245], à ne pas confondre avec la méthode du simplexe de Dantzig pour la programmation linéaire qui sera abordée en section 10.6. Des approches alternatives sont décrites en section 9.4.2.1 et au chapitre 11.

*Remarque 9.25.* La méthode de Nelder et Mead ne requiert pas la différentiabilité de la fonction de coût, mais reste bien sûr utilisable sur des fonctions différentiables. C'est un outil à tout faire remarquablement utile (et extrêmement populaire), bien qu'on en sache étonnamment peu sur ses propriétés théoriques [136, 135]. Elle est mise en œuvre dans la fonction MATLAB `fminsearch`. □

Un *simplexe* de $\mathbb{R}^n$ est un polytope convexe à $n+1$ sommets (un triangle quand $n=2$, un tétraèdre quand $n=3$, et ainsi de suite). L'idée de base de la méthode de Nelder et Mead est d'évaluer la fonction de coût en chaque sommet d'un simplexe dans l'espace de recherche, et de déduire des valeurs obtenues une transformation de ce simplexe qui le fasse ramper vers un minimiseur (local). Nous utiliserons ici un espace de recherche à deux dimensions à des fins d'illustration, mais la méthode est utilisable dans des espaces de dimension supérieure.

Trois sommets du simplexe courant seront distingués par des noms spécifiques :
- **b** est le meilleur (en termes de valeur du coût),
- **w** est le pire (nous voulons nous en éloigner ; il sera toujours rejeté du simplexe suivant, et son surnom est *sommet poubelle*),
- **s** est le plus mauvais après le pire.

Ainsi,

$$J(\mathbf{b}) \leqslant J(\mathbf{s}) \leqslant J(\mathbf{w}). \tag{9.208}$$

Quelques autres points jouent des rôles spéciaux :
- **c** est tel que ses coordonnées sont les moyennes arithmétiques des coordonnées des $n$ meilleurs sommets, c'est à dire de tous les sommets sauf **w**,
- $\mathbf{t}_{\text{ref}}$, $\mathbf{t}_{\text{exp}}$, $\mathbf{t}_{\text{in}}$ et $\mathbf{t}_{\text{out}}$ sont des points à tester.

Une itération de l'algorithme commence par une *réflexion* (figure 9.5), durant laquelle le point à tester est le symétrique du pire sommet courant par rapport au centre de gravité **c** de la face opposée

$$\mathbf{t}_{\text{ref}} = \mathbf{c} + (\mathbf{c} - \mathbf{w}) = 2\mathbf{c} - \mathbf{w}. \tag{9.209}$$

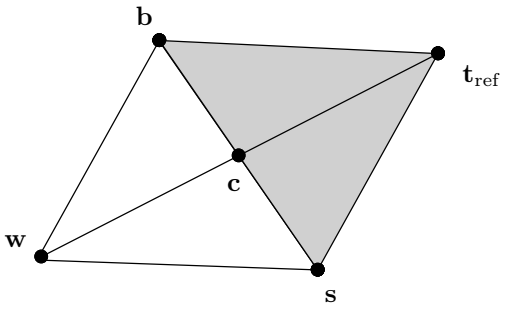


**Fig. 9.5** Réflexion (le nouveau simplexe potentiel est en gris)

Si $J(\mathbf{b}) \leqslant J(\mathbf{t}_{\mathrm{ref}}) \leqslant J(\mathbf{s})$, alors $\mathbf{w}$ est remplacé par $\mathbf{t}_{\mathrm{ref}}$. Si la réflexion a été plus réussie et $J(\mathbf{t}_{\mathrm{ref}}) < J(\mathbf{b})$, alors l'algorithme tente d'aller plus loin dans la même direction. C'est l'*expansion* (figure 9.6), où le point à tester devient

$$\mathbf{t}_{\mathrm{exp}} = \mathbf{c} + 2(\mathbf{c} - \mathbf{w}). \tag{9.210}$$

Si l'expansion réussit, c'est à dire si $J(\mathbf{t}_{\mathrm{exp}}) < J(\mathbf{t}_{\mathrm{ref}})$, alors $\mathbf{w}$ est remplacé par $\mathbf{t}_{\mathrm{exp}}$, sinon il reste remplacé par $\mathbf{t}_{\mathrm{ref}}$.

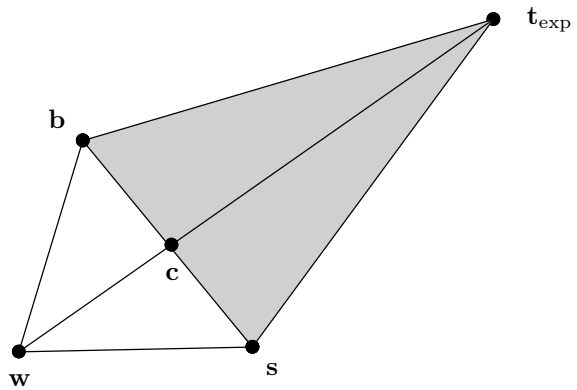


**Fig. 9.6** Expansion (le nouveau simplexe potentiel est en gris)

*Remarque 9.26.* Certains des sommets gardés d'une itération à la suivante doivent être renommés. Après une expansion réussie, par exemple, le point d'essai $\mathbf{t}_{\text{exp}}$ devient le meilleur sommet $\mathbf{b}$. □

Quand la réflexion est un échec plus net, c'est à dire quand $J(\mathbf{t}_{\text{ref}}) > J(\mathbf{s})$, deux types de contractions sont envisagés (figure 9.7). Si $J(\mathbf{t}_{\text{ref}}) < J(\mathbf{w})$, on tente une *contraction du côté réflexion* (ou *contraction externe*), avec le point d'essai

$$\mathbf{t}_{\text{out}} = \mathbf{c} + \frac{1}{2}(\mathbf{c} - \mathbf{w}) = \frac{1}{2}(\mathbf{c} + \mathbf{t}_{\text{ref}}), \tag{9.211}$$

tandis que si $J(\mathbf{t}_{\text{ref}}) \geqslant J(\mathbf{w})$ on tente une *contraction du côté du pire* (ou *contraction interne*), avec le point d'essai

$$\mathbf{t}_{\text{in}} = \mathbf{c} - \frac{1}{2}(\mathbf{c} - \mathbf{w}) = \frac{1}{2}(\mathbf{c} + \mathbf{w}). \tag{9.212}$$

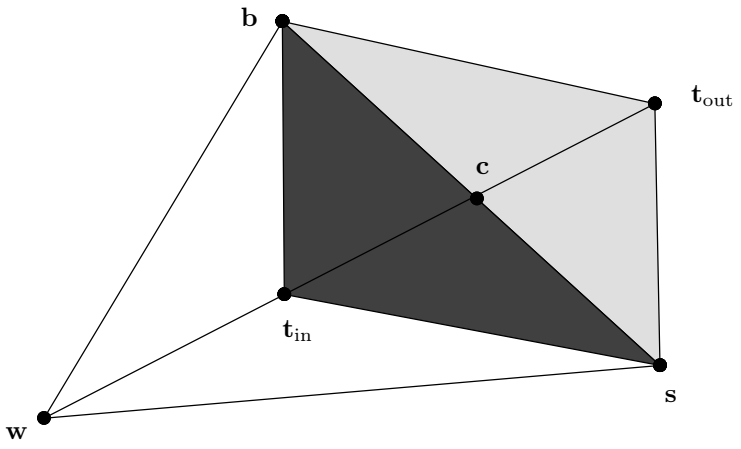


**Fig. 9.7** Contractions (les nouveaux simplexes potentiels sont en gris)

Soit $\widehat{\mathbf{t}}$ le meilleur point entre $\mathbf{t}_{\text{ref}}$ et $\mathbf{t}_{\text{in}}$ (ou entre $\mathbf{t}_{\text{ref}}$ et $\mathbf{t}_{\text{out}}$). Si $J(\widehat{\mathbf{t}}) < J(\mathbf{w})$, alors le pire sommet $\mathbf{w}$ est remplacé par $\widehat{\mathbf{t}}$.

Sinon, on effectue un *rétrécissement* (figure 9.8), durant lequel tous les autres sommets sont déplacés dans la direction du meilleur sommet, en divisant sa distance à $\mathbf{b}$ par deux, avant de commencer une nouvelle itération de l'algorithme, par une réflexion.

Les itérations s'arrêtent quand le volume du simplexe courant passe en dessous d'un seuil à choisir.

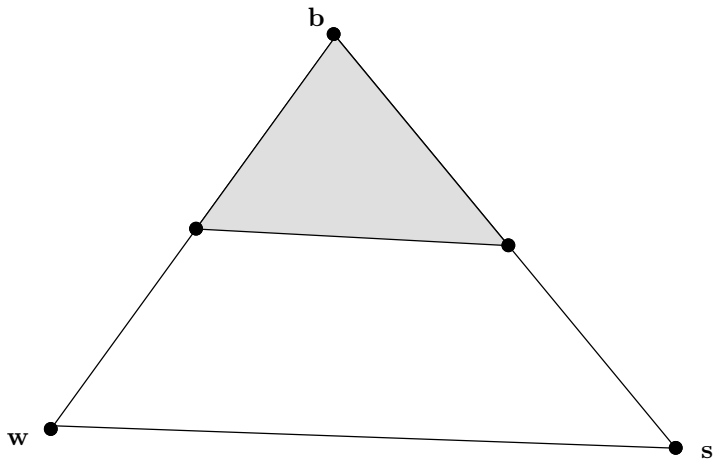


**Fig. 9.8** Rétrécissement (le nouveau simplexe est en gris)

## 9.4 Compléments

Cette section mentionne brièvement des extensions de méthodes d'optimisation sans contrainte visant à

— prendre en compte l'effet de perturbations sur la valeur de l'indice de performance,
— éviter de se faire piéger en des minimiseurs locaux qui ne sont pas globaux,
— diminuer le nombre des évaluations de la fonction de coût pour respecter des contraintes budgétaires,
— traiter des problèmes avec plusieurs objectifs en compétition.

### *9.4.1 Optimisation robuste*

Les performances dépendent souvent non seulement d'un vecteur de décision $\mathbf{x}$ but mais aussi de l'effet de perturbations. Nous supposons ici que ces perturbations peuvent être caractérisées par un vecteur $\mathbf{p}$ sur lequel de l'information a priori est disponible, et qu'on peut calculer un indice de performance $J(\mathbf{x},\mathbf{p})$. L'information a priori sur $\mathbf{p}$ peut prendre une des deux formes suivantes :

— une distribution de probabilités $\pi(\mathbf{p})$ connue pour $\mathbf{p}$ (on peut par exemple supposer que $\mathbf{p}$ est un vecteur aléatoire gaussien, de moyenne $\mathbf{0}$ et de matrice de covariance $\sigma^2\mathbf{I}$, avec $\sigma^2$ connue),

— un ensemble admissible $\mathbb{P}$ connu auquel $\mathbf{p}$ appartient (défini par exemple par des bornes inférieures et supérieures pour chacune des composantes de $\mathbf{p}$).

Dans ces deux cas, on veut choisir $\mathbf{x}$ de façon optimale tout en tenant compte de l'effet de $\mathbf{p}$. C'est de l'*optimisation robuste*, qui reçoit une attention considérable [10, 15]. Les deux sections qui suivent présentent deux méthodes utilisables dans ce contexte, une pour chacun des types d'information a priori sur $\mathbf{p}$.

#### 9.4.1.1 Optimisation en moyenne

Quand une distribution de probabilité $\pi(\mathbf{p})$ est disponible pour le vecteur des perturbations $\mathbf{p}$, on peut *éliminer* $\mathbf{p}$ *par moyennage*, en recherchant

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}} E_{\mathbf{p}}\{J(\mathbf{x},\mathbf{p})\}, \tag{9.213}$$

où $E_{\mathbf{p}}\{\cdot\}$ est l'opérateur espérance mathématique par rapport à $\mathbf{p}$. La méthode du gradient pour le calcul itératif d'une approximation de $\widehat{\mathbf{x}}$ poserait alors

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \lambda_k \mathbf{g}(\mathbf{x}^k), \tag{9.214}$$

avec

$$\mathbf{g}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}}[E_{\mathbf{p}}\{J(\mathbf{x},\mathbf{p})\}]. \tag{9.215}$$

Chaque itération requerrait donc l'évaluation du gradient d'une espérance mathématique, ce qui pourrait se révéler fort coûteux puisque ça pourrait impliquer l'évaluation numérique d'intégrales multidimensionnelles.

Pour contourner cette difficulté, la *méthode du gradient stochastique*, un exemple particulièrement simple de technique d'approximation stochastique, calcule plutôt

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \lambda_k \mathbf{g}'(\mathbf{x}^k), \tag{9.216}$$

avec

$$\mathbf{g}'(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}}[J(\mathbf{x},\mathbf{p}^k)], \tag{9.217}$$

où $\mathbf{p}^k$ est tiré au hasard suivant la loi $\pi(\mathbf{p})$ et où $\lambda_k$ doit satisfaire les trois conditions suivantes :

- $\lambda_k > 0$ (pour que les pas soient dans la bonne direction),
- $\sum_{k=0}^{\infty} \lambda_k = \infty$ (pour que toutes les valeurs possibles de $\mathbf{x}$ soient atteignables),
- $\sum_{k=0}^{\infty} \lambda_k^2 < \infty$ (pour que $\mathbf{x}^k$ converge vers un vecteur constant quand $k$ tend vers l'infini).

On peut utiliser, par exemple

$$\lambda_k = \frac{\lambda_0}{k+1},$$

avec $\lambda_0 > 0$ à choisir par l'utilisateur. Il existe des options plus sophistiquées, voir par exemple [244]. La méthode du gradient stochastique permet de minimiser une

espérance mathématique sans que jamais ni elle ni son gradient ne soient évalués. Comme cette méthode demeure locale, sa convergence vers un minimiseur global de $E_{\mathbf{p}}\{J(\mathbf{x},\mathbf{p})\}$ n'est pas garantie, et une stratégie *multistart* reste conseillée.

Un cas particulier intéressant est celui où $\mathbf{p}$ ne peut prendre que les valeurs $\mathbf{p}^i$, $i = 1,\cdots,N$, avec $N$ fini (éventuellement très grand), et où chaque $\mathbf{p}^i$ a la même probabilité $1/N$. L'optimisation en moyenne revient alors au calcul de

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}} J'(\mathbf{x}), \tag{9.218}$$

avec

$$J'(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} J_i(\mathbf{x}), \tag{9.219}$$

où

$$J_i(\mathbf{x}) = J(\mathbf{x},\mathbf{p}^i). \tag{9.220}$$

Pourvu que chaque fonction $J_i(\cdot)$ soit lisse et que $J'(\cdot)$ soit fortement convexe (comme c'est souvent le cas en apprentissage automatique), l'algorithme de gradient moyen stochastique (*stochastic average gradient*) présenté dans [140] peut battre un algorithme de gradient stochastique conventionnel à plate couture en termes de vitesse de convergence.

#### 9.4.1.2 Optimisation dans le pire des cas

Quand on dispose d'un ensemble admissible $\mathbb{P}$ pour le vecteur des perturbations $\mathbf{p}$, on peut rechercher le vecteur de décision $\widehat{\mathbf{x}}$ qui est le meilleur dans les pires circonstances, c'est à dire

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}}[\max_{\mathbf{p}\in\mathbb{P}} J(\mathbf{x},\mathbf{p})]. \tag{9.221}$$

C'est de l'*optimisation minimax* [199], couramment rencontrée en *théorie des jeux* où $\mathbf{x}$ et $\mathbf{p}$ caractérisent les décisions prises par deux joueurs. Le fait que $\mathbb{P}$ soit ici un ensemble continu rend le problème particulièrement difficile à résoudre. La stratégie naïve connue sous le nom de *best replay*, qui alterne la minimisation de $J$ par rapport à $\mathbf{x}$ pour la valeur courante de $\mathbf{p}$ et la maximisation de $J$ par rapport à $\mathbf{p}$ pour la valeur courante de $\mathbf{x}$, peut cycler sans espoir. D'un autre côté, un recours à la force brutale avec deux optimisations imbriquées se révèle en général trop compliqué pour être utile, à moins que $\mathbb{P}$ ne soit approximé par un ensemble fini $\mathscr{P}$ comportant suffisamment peu d'éléments pour que la maximisation par rapport à $\mathbf{p}$ puisse être conduite par recherche exhaustive. La méthode de relaxation [219] construit $\mathscr{P}$ itérativement, comme suit :

1. Poser $\mathscr{P} = \{\mathbf{p}^1\}$, avec $\mathbf{p}^1$ tiré au hasard dans $\mathbb{P}$, et $k = 1$.
2. Trouver $\mathbf{x}^k = \arg\min_{\mathbf{x}\in\mathbb{X}}[\max_{\mathbf{p}\in\mathscr{P}} J(\mathbf{x},\mathbf{p})]$.
3. Trouver $\mathbf{p}^{k+1} = \arg\max_{\mathbf{p}\in\mathbb{P}} J(\mathbf{x}^k,\mathbf{p})$.

4. Si $J(\mathbf{x}^k, \mathbf{p}^{k+1}) \leqslant \max_{\mathbf{p}\in\mathscr{P}} J(\mathbf{x}^k, \mathbf{p}) + \delta$, où $\delta > 0$ est un paramètre de tolérance choisi par l'utilisateur, alors accepter $\mathbf{x}^k$ en tant qu'approximation de $\widehat{\mathbf{x}}$. Sinon, poser $\mathscr{P} := \mathscr{P} \cup \{\mathbf{p}^{k+1}\}$, incrémenter $k$ d'une unité et aller au pas 2.

Cette méthode laisse libre le choix des méthodes d'optimisation à utiliser aux pas 2 et 3. Sous des conditions techniques raisonnables, elle s'arrête après un nombre fini d'itérations.

### *9.4.2 Optimisation globale*

L'optimisation globale recherche l'optimum global de la fonction d'objectif, et les valeurs de tous optimiseurs globaux associés (ou au moins la valeur de l'un d'entre eux). Elle contourne ainsi les problèmes d'initialisation que posent les méthodes locales. Il existe deux approches complé-mentaires, qui diffèrent par le type de recherche conduit. La *recherche aléatoire* est facile à mettre en œuvre et peut être utilisée sur de vastes classes de problèmes mais ne garantit pas son succès, tandis que la *recherche déterministe* [111] est plus compliquée à mettre en œuvre et moins généralement applicable mais permet de garantir des affirmations concernant l'optimum et les optimiseurs globaux. Les deux sections qui suivent décrivent brièvement des exemples de ces deux stratégies. Dans les deux cas, la recherche est conduite dans un domaine $\mathbb{X}$ (éventuellement très large) qui prend la forme d'un hyper-rectangle aux côtés alignés sur les axes, ou *boîte*. Dans la mesure où aucun optimiseur n'est supposé appartenir à la frontière de $\mathbb{X}$, il s'agit bien d'optimisation sans contrainte.

*Remarque 9.27.* Quand on doit estimer un vecteur $\mathbf{x}$ de paramètres d'un modèle à partir de données expérimentales en minimisant la norme $l_p$ d'un vecteur d'erreur ($p = 1, 2, \infty$), des conditions expérimentales appropriées peuvent éliminer tous les minimiseurs locaux sous-optimaux, ce qui permet alors d'utiliser des méthodes locales pour évaluer un minimiseur global [188]. □

#### 9.4.2.1 Recherche aléatoire

Le *multistart* est un exemple particulièrement simple de recherche aléatoire. Nombre de stratégies plus sophistiquées ont été inspirées par la biologie (avec les algorithmes génétiques [248, 75] et l'évolution différentielle [228]), les sciences du comportement (avec l'optimisation par colonies de fourmis [57] et par essaims de particules [124]) et la métallurgie (avec le recuit simulé, voir la section 11.2). La plupart des algorithmes de recherche aléatoire ont des paramètres internes qu'il faut régler et qui ont un impact significatif sur leur comportement, et on ne devrait pas oublier le temps passé à régler ces paramètres quand on évalue les performances de ces algorithmes sur une application donnée. La *recherche aléatoire adaptative* (RAA) [9] a montré dans [189] sa capacité à résoudre des problèmes test et des ap-

plications réelles variés en gardant les mêmes réglages pour ses paramètres internes. La description de la RAA présentée ici correspond à des choix typiques, auxquels il existe des alternatives parfaitement valides. (On peut, par exemple, utiliser des distributions uniformes pour générer les déplacements aléatoires plutôt que des distributions gaussiennes.)

L'algorithme de base est excessivement simple :

1. Choisir $\mathbf{x}^0$, poser $k = 0$.
2. Calculer un point d'essai $\mathbf{x}^{k+} = \mathbf{x}^k + \boldsymbol{\delta}^k$, avec $\boldsymbol{\delta}^k$ tiré au hasard.
3. Si $J(\mathbf{x}^{k+}) < J(\mathbf{x}^k)$ alors $\mathbf{x}^{k+1} = \mathbf{x}^{k+}$, sinon $\mathbf{x}^{k+1} = \mathbf{x}^k$.
4. Incrémenter $k$ d'une unité et aller au pas 2.

Cinq versions de cet algorithme sont en compétition. Dans la $j$-ème version ($j = 1, \cdots, 5$), on utilise une distribution gaussienne $\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}({}^j\boldsymbol{\sigma}))$ pour générer $\boldsymbol{\delta}^k$, avec une matrice de covariance diagonale

$$\boldsymbol{\Sigma}({}^j\boldsymbol{\sigma}) = \operatorname{diag}\left\{{}^j\sigma_i^2, \quad i = 1, \cdots, \dim \mathbf{x}\right\}, \tag{9.222}$$

et on limite les déplacements pour assurer que $\mathbf{x}^{k+}$ reste dans $\mathbb{X}$. Les distributions diffèrent par la valeur donnée à ${}^j\boldsymbol{\sigma}$, $j = 1, \cdots, 5$. On peut prendre, par exemple,

$${}^1\sigma_i = x_i^{\max} - x_i^{\min}, \quad i = 1, \cdots, \dim \mathbf{x}, \tag{9.223}$$

pour promouvoir de grands déplacements dans $\mathbb{X}$, et

$${}^j\boldsymbol{\sigma} = {}^{j-1}\boldsymbol{\sigma}/10, \quad j = 2, \cdots, 5. \tag{9.224}$$

pour favoriser des explorations de plus en plus fines.

On alterne une phase de sélection de variance et une phase d'exploitation de variance. Dans la *phase de sélection de variance*, les cinq algorithmes de base en compétition sont exécutés à partir du même point initial (le meilleur $\mathbf{x}$ disponible en début de phase). Chaque algorithme dispose de $100/i$ itérations, pour donner plus d'essais aux variances les plus grandes. L'algorithme qui atteint les meilleurs résultats (en termes de valeur finale du coût) est sélectionné pour la *phase d'exploitation de variance* suivante, durant laquelle il est initialisé au meilleur $\mathbf{x}$ disponible et utilisé pour 100 itérations avant de recommencer une phase de sélection de variance.

On peut décider de basculer sur un programme d'optimisation locale chaque fois que ${}^5\boldsymbol{\sigma}$ est sélectionné, puisque ${}^5\boldsymbol{\sigma}$ correspond à de très petits déplacements. La recherche s'arrête quand le budget pour l'évaluation du coût est épuisé ou quand ${}^5\boldsymbol{\sigma}$ a été sélectionné un nombre donné de fois consécutives.

Cet algorithme est extrêmement simple à mettre en œuvre, et ne requiert pas que la fonction de coût soit différentiable. Son utilisation est si flexible qu'elle encourage la créativité dans le développement de fonctions de coût sur mesure. Il peut échapper à des minimiseurs locaux parasites, mais on ne peut fournir aucune garantie quant à sa capacité à localiser un minimiseur global en un nombre fini d'itérations.

#### 9.4.2.2 Optimisation garantie

Une idée clé permettant de prouver des affirmations sur les minimiseurs globaux de fonctions de coût non convexes est de *diviser pour régner*. Cette idée est mise en œuvre dans les algorithmes dits de séparation et évaluation (*branch and bound*). La séparation (*branching*) partitionne l'ensemble admissible initial $\mathbb{X}$ en sous-ensembles, tandis que l'évaluation (*bounding*) calcule des bornes pour les valeurs prises par des quantités d'intérêt sur chacun des sous-ensembles résultants. Ceci permet de *prouver* que certains sous-ensembles ne contiennent aucun minimiseur global de la fonction de coût sur $\mathbb{X}$ et peuvent donc être éliminés des recherches ultérieures. Voici deux exemples de telles preuves :

— si une borne inférieure de la valeur de la fonction de coût sur $\mathbb{X}^i \subset \mathbb{X}$ est plus grande qu'une borne supérieure du minimum de la fonction de coût sur $\mathbb{X}^j \subset \mathbb{X}$, alors $\mathbb{X}^i$ ne contient aucun minimiseur global,

— si au moins une composante du gradient de la fonction de coût est telle que sa borne supérieure sur $\mathbb{X}^i \subset \mathbb{X}$ est strictement négative (ou sa borne inférieure strictement positive), alors la condition nécessaire d'optimalité (9.6) n'est satisfaite nulle part sur $\mathbb{X}^i$, qui ne contient donc aucun minimiseur local ou global (sans contrainte).

Tout sous-ensemble de $\mathbb{X}$ qui ne peut pas être éliminé peut contenir un minimiseur global de la fonction de coût sur $\mathbb{X}$. La séparation peut alors être utilisée pour le couper en sous-ensembles plus petits sur lesquels une évaluation est conduite. Il est quelquefois possible de localiser tous les minimiseurs globaux de façon très précise avec ce type d'approche.

L'analyse par intervalles (voir la section 14.5.2.3 et [115, 171, 195, 196]) est un bon fournisseur de bornes pour les valeurs que prennent la fonction de coût et ses dérivées sur des sous-ensembles de $\mathbb{X}$, et des algorithmes d'optimisation globale à base de calcul sur les intervalles sont décrits dans [96, 121, 191].

### *9.4.3 Optimisation avec un budget serré*

Parfois, l'évaluation de la fonction de coût est si coûteuse que le nombre des évaluations autorisées est très restreint. C'est souvent le cas quand des modèles fondés sur les lois de la physique sont simulés dans des conditions réalistes, par exemple pour concevoir des voitures plus sûres en simulant des accidents.

L'évaluation de $J(\mathbf{x})$ pour une valeur numérique donnée du vecteur de décision $\mathbf{x}$ peut être vue comme une expérience sur ordinateur, ou *computer experiment* [205], pour laquelle on peut construire des *modèles de substitution*. Un modèle de substitution prédit la valeur de la fonction de coût sur la base de ses évaluations passées. Il peut ainsi être utilisé pour trouver des valeurs prometteuses du vecteur de décision où la vraie fonction de coût est alors évaluée. Parmi toutes les méthodes disponibles pour construire des modèles de substitution, le krigeage, décrit brièvement en section 5.4.3, a l'avantage de fournir non seulement une prédiction $\widehat{J}(\mathbf{x})$ du coût

$J(\mathbf{x})$, mais aussi une évaluation de la qualité de cette prédiction, sous la forme d'une variance estimée $\widehat{\sigma}^2(\mathbf{x})$. La méthode EGO (pour *efficient global optimization*) [118], qui peut être interprétée dans le contexte de l'optimisation bayésienne [157], recherche la valeur de $\mathbf{x}$ qui maximise l'espérance de l'amélioration (ou *expected improvement*, EI) par rapport à la meilleure valeur du coût obtenue jusqu'ici. Maximiser $\mathrm{EI}(\mathbf{x})$ est de nouveau un problème d'optimisation, mais beaucoup moins coûteux à résoudre que le problème de départ. En exploitant le fait que, pour n'importe quelle valeur donnée de $\mathbf{x}$, la prédiction par krigeage de $J(\mathbf{x})$ est gaussienne, de moyenne $\widehat{J}(\mathbf{x})$ et de variance $\widehat{\sigma}^2(\mathbf{x})$ connues, on peut montrer que

$$\mathrm{EI}(\mathbf{x}) = \widehat{\sigma}(\mathbf{x})[u\Phi(u) + \varphi(u)], \tag{9.225}$$

où $\varphi(\cdot)$ et $\Phi(\cdot)$ sont la densité de probabilité et la distribution cumulative d'une variable gaussienne de moyenne nulle et de variance unité, et où

$$u = \frac{J_{\mathrm{best}}^{\mathrm{sofar}} - \widehat{J}(\mathbf{x})}{\widehat{\sigma}(\mathbf{x})}, \tag{9.226}$$

avec $J_{\mathrm{best}}^{\mathrm{sofar}}$ la plus petite valeur du coût sur toutes les évaluations effectuées jusqu'ici.

$\mathrm{EI}(\mathbf{x})$ sera grand si $\widehat{J}(\mathbf{x})$ est petit ou si $\widehat{\sigma}^2(\mathbf{x})$ est grand, ce qui donne à EGO une certaine capacité à échapper à l'attraction de minimiseurs locaux et à explorer des régions inconnues. La figure 9.9 illustre un pas d'EGO sur un problème à une variable de décision. La prédiction par krigeage de la fonction de coût $J(x)$ est en haut, et l'espérance de l'amélioration $\mathrm{EI}(x)$ en bas (avec une échelle logarithmique). Le graphe de la fonction de coût à minimiser est en pointillé. Le graphe de la moyenne de la prédiction par krigeage est en trait plein, avec les coûts précédemment évalués indiqués par des carrés. La ligne horizontale en pointillé indique la valeur de $J_{\mathrm{best}}^{\mathrm{sofar}}$. La région de confiance à 95% pour la prédiction est en gris. La prochaine évaluation de $J(x)$ devrait être conduite là où $\mathrm{EI}(x)$ atteint sa valeur maximale, c'est à dire vers $x = -0.62$. C'est loin de là où la meilleure valeur du coût a été atteinte, parce que l'incertitude sur $J(x)$ rend d'autres régions potentiellement intéressantes.

Une fois que

$$\widehat{\mathbf{x}} = \arg\max_{\mathbf{x}\in\mathbb{X}} \mathrm{EI}(\mathbf{x}) \tag{9.227}$$

a été trouvé, le vrai coût $J(\widehat{\mathbf{x}})$ est évalué. S'il diffère nettement de sa prédiction $\widehat{J}(\widehat{\mathbf{x}})$, alors $\widehat{\mathbf{x}}$ et $J(\widehat{\mathbf{x}})$ sont ajoutés aux données d'entraînement, un nouveau modèle de substitution par krigeage est construit et le processus est itéré. Sinon, $\widehat{\mathbf{x}}$ est pris comme approximation d'un minimiseur (global).

Comme toutes les approches utilisant une surface de réponse pour de l'optimisation, celle-ci peut échouer sur des fonctions trompeuses [117].

*Remarque 9.28.* En combinant la méthode de relaxation de [219] et la méthode EGO de [118], on peut calculer des optimiseurs minimax approximatifs avec un nombre limité d'évaluations du coût [151]. □

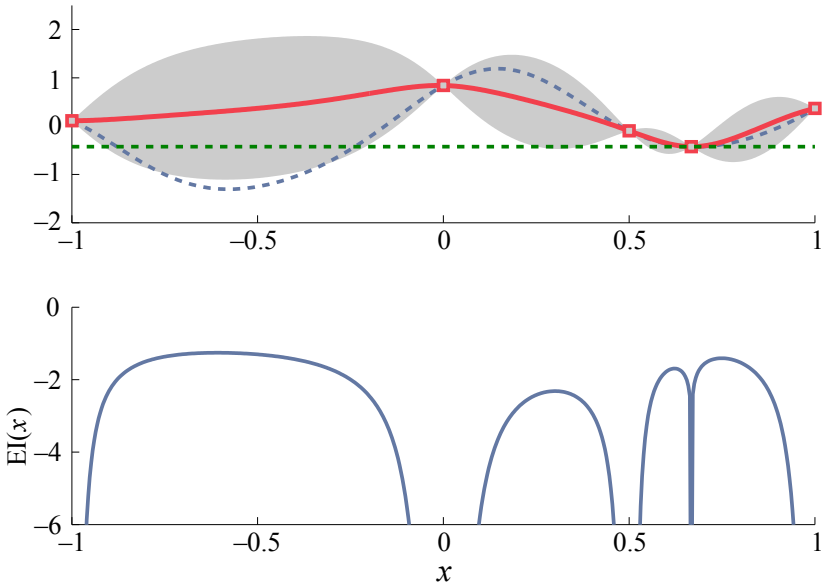


**Fig. 9.9** Prédiction par krigeage (en haut) et espérance de l'amélioration sur une échelle logarithmique (en bas) (merci à Emmanuel Vazquez, de Supélec)

### *9.4.4 Optimisation multi-objectif*

Jusqu'ici, nous avons supposé qu'il fallait minimiser une seule fonction de coût scalaire $J(\cdot)$. Tel n'est pas toujours le cas, et on peut souhaiter minimiser simultanément plusieurs fonctions de coût $J_i(\mathbf{x})$ $(i = 1, \cdots, n_J)$. Ceci ne poserait aucun problème si ces dernières avaient les mêmes minimiseurs, mais en général il y a des objectifs en conflit et des compromis sont inévitables.

Plusieurs stratégies permettent de se ramener à une minimisation conventionnelle. On peut, par exemple, définir une fonction de coût scalaire composite par combinaison linéaire des fonctions de coût individuelles

$$J(\mathbf{x}) = \sum_{i=1}^{n_J} w_i J_i(\mathbf{x}), \tag{9.228}$$

avec des poids $w_i$ positifs à choisir par l'utilisateur. On peut aussi donner la priorité à l'une des fonctions de coût et minimiser celle-ci sous des contraintes sur les valeurs autorisées aux autres (voir le chapitre 10).

Ces deux stratégies restreignent le choix, cependant, et on peut préférer rechercher le *front de Pareto*, c'est à dire l'ensemble des $\mathbf{x}$ dans $\mathbb{X}$ tels que tout déplacement local qui fait décroître une fonction de coût $J_i$ donnée fasse croître au moins l'une des autres fonctions de coût. Le front de Pareto front est ainsi un ensemble de solutions de compromis. Il est bien sûr beaucoup plus compliqué de calculer un front de

Pareto que de minimiser une seule fonction de coût [43]. Dans une phase ultérieure, il faudra en général prendre une seule décision $\widehat{\mathbf{x}}$ parmi cet ensemble, ce qui correspond à minimiser (9.228) pour un choix spécifique des poids $w_i$. Un examen de la forme du front de Pareto peut aider l'utilisateur à choisir le compromis le plus approprié.

## 9.5 Exemples MATLAB

Ces exemples correspondent à l'estimation des paramètres d'un modèle à partir de données expérimentales. A chaque fois, ces données ont été générées en simulant le modèle pour une vraie valeur connue du vecteur de ses paramètres, mais cette connaissance n'a pas été utilisée dans la procédure d'estimation, bien sûr. Aucun bruit de mesure simulé n'a été ajouté à ces données artificielles. Même si des erreurs d'arrondi sont inévitables, la valeur de la norme de l'erreur entre les données et la sortie du meilleur modèle doit donc être proche de zéro, et les paramètres optimaux devraient, on l'espère, être proches de leurs vraies valeurs. (L'exemple de la section 12.4.3 montrera qu'un manque d'identifiabilité [243] peut se traduire par de grandes erreurs sur les valeurs des paramètres même quand la valeur de la fonction de coût est très faible.)

### *9.5.1 Moindres carrés sur un modèle polynomial multivarié*

Le vecteur $\mathbf{p}$ des paramètres du modèle polynomial à quatre entrées et une sortie

$$\begin{aligned} y_{\mathrm{M}}(\mathbf{x},\mathbf{p}) &= p_1 + p_2x_1 + p_3x_2 + p_4x_3 + p_5x_4 + p_6x_1x_2 + p_7x_1x_3 \\ &\quad + p_8x_1x_4 + p_9x_2x_3 + p_{10}x_2x_4 + p_{11}x_3x_4 \end{aligned} \tag{9.229}$$

doit être estimé à partir des données $(y_i,\mathbf{x}^i), i = 1,\cdots,N$. Pour n'importe quelle valeur $\mathbf{x}^i$ donnée du vecteur des entrées, la donnée correspondante est calculée suivant

$$y_i = y_{\mathrm{M}}(\mathbf{x}^i,\mathbf{p}^\star), \tag{9.230}$$

où $\mathbf{p}^\star$ est la vraie valeur du vecteur des paramètres, choisie arbitrairement comme

$$\mathbf{p}^\star = (10,-9,8,-7,6,-5,4,-3,2,-1,0)^{\mathrm{T}}. \tag{9.231}$$

L'estimée $\widehat{\mathbf{p}}$ est calculée comme

$$\widehat{\mathbf{p}} = \arg\min_{\mathbf{p}\in\mathbb{R}^{11}} J(\mathbf{p}), \tag{9.232}$$

où

$$J(\mathbf{p}) = \sum_{i=1}^{N} [y_i - y_{\mathrm{M}}(\mathbf{x}^i, \mathbf{p})]^2. \tag{9.233}$$

Comme $y_{\mathrm{M}}(\mathbf{x}^i, \mathbf{p})$ est linéaire en $\mathbf{p}$, les moindres carrés linéaires s'appliquent. Le domaine admissible $\mathbb{X}$ pour le vecteur $\mathbf{x}^i$ des entrées est défini comme le produit cartésien d'intervalles admissibles pour chacun des facteurs d'entrée. Le $j$-ème facteur d'entrée peut prendre n'importe quelle valeur dans [`min(j)`, `max(j)`], avec

```
min(1) = 0; max(1) = 0.05;
min(2) = 50; max(2) = 100;
min(3) = -1; max(3) = 7;
min(4) = 0; max(4) = 1.e5;
```

Les domaines admissibles pour les quatre facteurs d'entrée sont donc très différents, ce qui tend à rendre le problème mal conditionné.

Deux plans d'expériences D1 et D2 sont envisagés pour la collecte des données. Avec D1, chaque $\mathbf{x}^i$ est tiré au hasard dans $\mathbb{X}$, tandis que D2 est un plan factoriel complet à deux niveaux, dans lequel les données sont collectées à toutes les combinaisons possibles des bornes des intervalles autorisés pour les facteurs d'entrée. D2 comporte donc $2^4 = 16$ conditions expérimentales différentes $\mathbf{x}^i$. Dans ce qui suit, le nombre $N$ des paires $(y_i, \mathbf{x}^i)$ de données de D1 est pris égal à 32, de sorte que D2 est répété pour obtenir autant de données qu'avec D1.

Les données de sortie sont dans `Y` pour D1 et dans `Yfd` pour D2, tandis que les valeurs correspondantes des facteurs d'entrée sont dans `X` pour D1 et dans `Xfd` pour D2. La fonction qui suit est utilisée pour estimer les paramètres `P` à partir des données de sortie `Y` et de la matrice de régression `F` correspondante :

```
function[P,Cond] = LSforExample(F,Y,option)
% F (nExp,nPar) contient la matrice de régression.
% Y (nExp,1) contient les sorties mesurées.
% option spécifie comment l'estimée est calculée ;
% = 1 pour la résolution des équations normales,
% 2 pour la factorisation QR et 3 pour la SVD.
% P (nPar,1) contient les estimées des paramètres.
% Cond est le conditionnement du système résolu par
% l'approche choisie (pour la norme spectrale).
[nExp,nPar] = size(F);

if (option == 1)
    % Calcul de P via les équations normales
    P = (F'*F)\F'*Y;
    % ici, \ est par élimination gaussienne
    Cond = cond(F'*F);
end

if (option == 2)
    % Calcul de P par factorisation QR
```

```
    [Q,R] = qr(F);
    QTY = Q'*Y;
    opts_UT.UT = true;
    P = linsolve(R,QTY,opts_UT);
    Cond = cond(R);
end

if (option == 3)
    % Calcul de P par SVD
    [U,S,V] = svd(F,'econ');
    P = V*inv(S)*U'*Y;
    Cond = cond(S);
end
end
```

#### 9.5.1.1 Utilisation d'expériences générées de façon aléatoire

Traitons tout d'abord les données collectées suivant le plan D1, avec le script

```
% Remplissage de la matrice de régression
F = zeros(nExp,nPar);
for i=1:nExp,
    F(i,1) = 1;
    F(i,2) = X(i,1);
    F(i,3) = X(i,2);
    F(i,4) = X(i,3);
    F(i,5) = X(i,4);
    F(i,6) = X(i,1)*X(i,2);
    F(i,7) = X(i,1)*X(i,3);
    F(i,8) = X(i,1)*X(i,4);
    F(i,9) = X(i,2)*X(i,3);
    F(i,10) = X(i,2)*X(i,4);
    F(i,11) = X(i,3)*X(i,4);
end
% Conditionnement du problème initial
InitialCond = cond(F)

% Calcul du P optimal avec les équations normales
[PviaNE,CondViaNE] = LSforExample(F,Y,1)
OptimalCost = (norm(Y-F*PviaNE))^2
NormErrorP = norm(PviaNE-trueP)

% Calcul du P optimal via une factorisation QR
[PviaQR,CondViaQR] = LSforExample(F,Y,2)
OptimalCost = (norm(Y-F*PviaQR))^2
```

```
NormErrorP = norm(PviaQR-trueP)

% Calcul du P optimal via une SVD
[PviaSVD,CondViaSVD] = LSforExample(F,Y,3)
OptimalCost = (norm(Y-F*PviaSVD))^2
NormErrorP = norm(PviaSVD-trueP)
```

Le conditionnement du problème initial se révèle être

```
InitialCond =
     2.022687340567638e+09
```

Les résultats obtenus en résolvant les équations normales sont

```
PviaNE =
     9.999999744351953e+00
    -8.999994672834873e+00
     8.000000003536115e+00
    -6.999999981897417e+00
     6.000000000000670e+00
    -5.000000071944669e+00
     3.999999956693500e+00
    -2.999999999998153e+00
     1.999999999730790e+00
    -1.000000000000011e+00
     2.564615186884112e-14

CondViaNE =
     4.097361000068907e+18

OptimalCost =
     8.281275106847633e-15

NormErrorP =
     5.333988749555268e-06
```

Bien que le conditionnement des équations normales soit dangereusement élevé, cette approche fournit encore des estimées plutôt bonnes des paramètres.

Les résultats obtenus via une factorisation QR de la matrice de régression sont

```
PviaQR =
     9.999999994414727e+00
    -8.999999912908700e+00
     8.000000000067203e+00
    -6.999999999297954e+00
     6.000000000000007e+00
    -5.000000001454850e+00
     3.999999998642462e+00
```

```
     -2.999999999999567e+00
      1.999999999988517e+00
     -1.000000000000000e+00
      3.038548268619260e-15

CondViaQR =
      2.022687340567638e+09

OptimalCost =
      4.155967155703225e-17

NormErrorP =
      8.729574294487699e-08
```

Le conditionnement du problème initial est récupéré, et les estimées des paramètres sont plus précises qu'avec les équations normales.

Les résultats obtenus via une SVD de la matrice de régression sont

```
PviaSVD =
      9.999999993015081e+00
     -9.000000089406967e+00
      8.000000000036380e+00
     -7.000000000407454e+00
      6.000000000000076e+00
     -5.000000000232831e+00
      4.000000002793968e+00
     -2.999999999999460e+00
      2.000000000003638e+00
     -1.000000000000000e+00
     -4.674038933671909e-14

CondViaSVD =
      2.022687340567731e+09

OptimalCost =
      5.498236550294591e-15

NormErrorP =
      8.972414778806571e-08
```

Le conditionnement du problème résolu est légèrement plus élevé que pour le problème initial et l'approche QR, et les estimées sont légèrement moins précises qu'avec l'approche QR pourtant plus simple.

#### 9.5.1.2 Normalisation des facteurs d'entrée

On peut s'attendre à ce qu'une transformation affine imposant à chacun des facteurs d'entrée d'appartenir à l'intervalle $[-1,1]$ améliore le conditionnement du problème. Cette transformation est mise en œuvre par le script suivant, qui procède ensuite comme précédemment pour traiter les données résultantes.

```
% Déplacement des facteurs d'entrée dans [-1,1]
for i = 1:nExp
    for k = 1:nFact
        Xn(i,k) = (2*X(i,k)-min(k)-max(k))...
            /(max(k)-min(k));
    end
end
% Remplissage de la matrice de régression
% avec  des facteurs d'entrée dans [-1,1].
% ATTENTION, ceci change les paramètres !
Fn = zeros(nExp,nPar);
for i=1:nExp
    Fn(i,1) = 1;
    Fn(i,2) = Xn(i,1);
    Fn(i,3) = Xn(i,2);
    Fn(i,4) = Xn(i,3);
    Fn(i,5) = Xn(i,4);
    Fn(i,6) = Xn(i,1)*Xn(i,2);
    Fn(i,7) = Xn(i,1)*Xn(i,3);
    Fn(i,8) = Xn(i,1)*Xn(i,4);
    Fn(i,9) = Xn(i,2)*Xn(i,3);
    Fn(i,10) = Xn(i,2)*Xn(i,4);
    Fn(i,11) = Xn(i,3)*Xn(i,4);
end

% Conditionnement du nouveau problème initial
NewInitialCond = cond(Fn)

% Calcul des nouveaux paramètres optimaux
% avec les équations normales
[NewPviaNE,NewCondViaNE] = LSforExample(Fn,Y,1)
OptimalCost = (norm(Y-Fn*NewPviaNE))^2

% Calcul des nouveaux paramètres optimaux
% via une factorisation QR
[NewPviaQR,NewCondViaQR] = LSforExample(Fn,Y,2)
OptimalCost = (norm(Y-Fn*NewPviaQR))^2

% Calcul des nouveauxparamètres optimaux via une SVD
```

```
[NewPviaSVD,NewCondViaSVD] = LSforExample(Fn,Y,3)
OptimalCost = (norm(Y-Fn*NewPviaSVD))^2
```

Le conditionnement du problème tranformé se révèle être

```
NewInitialCond =
     5.633128746769874e+00
```

Il est donc bien meilleur que pour le problème initial.

Les résultats obtenus en résolvant les équations normales sont

```
NewPviaNE =
    -3.452720300000000e+06
    -3.759299999999603e+03
    -1.249653125000001e+06
     5.723999999996740e+02
    -3.453750000000000e+06
    -3.124999999708962e+00
     3.999999997322448e-01
    -3.750000000000291e+03
     2.000000000006985e+02
    -1.250000000000002e+06
     7.858034223318100e-10

NewCondViaNE =
     3.173213947768512e+01

OptimalCost =
     3.218047573208537e-17
```

Les résultats obtenus via une factorisation QR de la matrice de régression sont

```
NewPviaQR =
    -3.452720300000001e+06
    -3.759299999999284e+03
    -1.249653125000001e+06
     5.724000000002399e+02
    -3.453750000000001e+06
    -3.125000000827364e+00
     3.999999993921934e-01
    -3.750000000000560e+03
     2.000000000012406e+02
    -1.250000000000000e+06
     2.126983788033481e-09

NewCondViaQR =
     5.633128746769874e+00
```

```
OptimalCost =
     7.951945308823372e-17
```

Bien que le conditionnement du problème initial transformé soit recouvré, la solution est en fait légèrement moins précise qu'avec les équations normales.

Les résultats obtenus via une SVD de la matrice de régression sont

```
NewPviaSVD =
    -3.452720300000001e+06
    -3.759299999998882e+03
    -1.249653125000000e+06
     5.724000000012747e+02
    -3.453749999999998e+06
    -3.125000001688022e+00
     3.999999996158294e-01
    -3.750000000000931e+03
     2.000000000023283e+02
    -1.250000000000001e+06
     1.280568540096283e-09

NewCondViaSVD =
     5.633128746769864e+00

OptimalCost =
     1.847488972244773e-16
```

Une fois de plus, la solution obtenue via une SVD est légèrement moins précise que celle obtenue via une factorisation QR. L'approche par résolution des équations normales sort donc la grande gagnante de cette version du problème, puisqu'elle est la moins coûteuse et la plus précise.

#### 9.5.1.3 Utilisation d'un plan factoriel complet à deux niveaux

Traitons enfin les données collectées suivant le plan D2, défini comme suit.

```
% Plan factoriel complet à deux niveaux
% pour le cas où nFact = 4
FD = [-1, -1, -1, -1;
      -1, -1, -1, +1;
      -1, -1, +1, -1;
      -1, -1, +1, +1;
      -1, +1, -1, -1;
      -1, +1, -1, +1;
      -1, +1, +1, -1;
      -1, +1, +1, +1;
      +1, -1, -1, -1;
```

```
        +1, -1, -1, +1;
        +1, -1, +1, -1;
        +1, -1, +1, +1;
        +1, +1, -1, -1;
        +1, +1, -1, +1;
        +1, +1, +1, -1;
        +1, +1, +1, +1;];
```

Les intervalles autorisés pour les facteurs restent normalisés à $[-1,1]$, mais chaque facteur est maintenant toujours égal à $\pm 1$. La résolution des équations normales devient alors particulièrement facile, puisque la matrice de régression `Ffd` résultante est maintenant telle que `Ffd'*Ffd` est un multiple de la matrice identité. Nous pouvons ainsi utiliser le script

```
% remplissage de la matrice de régression
Ffd = zeros(nExp,nPar);
for j=1:nRep,
    for i=1:16,
        Ffd(16*(j-1)+i,1) = 1;
        Ffd(16*(j-1)+i,2) = FD(i,1);
        Ffd(16*(j-1)+i,3) = FD(i,2);
        Ffd(16*(j-1)+i,4) = FD(i,3);
        Ffd(16*(j-1)+i,5) = FD(i,4);
        Ffd(16*(j-1)+i,6) = FD(i,1)*FD(i,2);
        Ffd(16*(j-1)+i,7) = FD(i,1)*FD(i,3);
        Ffd(16*(j-1)+i,8) = FD(i,1)*FD(i,4);
        Ffd(16*(j-1)+i,9) = FD(i,2)*FD(i,3);
        Ffd(16*(j-1)+i,10) = FD(i,2)*FD(i,4);
        Ffd(16*(j-1)+i,11) = FD(i,3)*FD(i,4);
    end
end

% Résolution (ici triviale) des équations normales
NewPviaNEandFD = Ffd'*Yfd/(16*nRep)
NewCondviaNEandFD = cond(Ffd)
OptimalCost = (norm(Yfd-Ffd*NewPviaNEandFD))^2
```

Ceci produit

```
NewPviaNEandFD =
    -3.452720300000000e+06
    -3.759299999999965e+03
    -1.249653125000000e+06
     5.723999999999535e+02
    -3.453750000000000e+06
    -3.125000000058222e+00
     3.999999999534225e-01
```

```
     -3.749999999999965e+03
      2.000000000000000e+02
     -1.250000000000000e+06
     -4.661160346586257e-11

NewCondviaNEandFD =
     1.000000000000000e+00

OptimalCost =
     1.134469775459169e-17
```

Ces résultats sont les plus précis, et ils ont été obtenus avec le moins de calcul.

Pour le même problème, une normalisation de l'intervalle autorisé pour les facteurs d'entrée et l'utilisation d'un plan factoriel approprié ont ainsi réduit le conditionnement des équations normales, d'à peu près $4.1 \cdot 10^{18}$ à un.

### *9.5.2 Estimation non linéaire*

Nous voulons estimer les trois paramètres du modèle

$$y_{\mathrm{M}}(t_i, \mathbf{p}) = p_1[\exp(-p_2 t_i) - \exp(-p_3 t_i)], \tag{9.234}$$

mis en œuvre dans la fonction

```
function [y] = ExpMod(p,t)
ntimes = length(t);
y = zeros(ntimes,1);
for i=1:ntimes,
    y(i) = p(1)*(exp(-p(2)*t(i))-exp(-p(3)*t(i)));
end
end
```

Des données sans bruit sont générées pour $\mathbf{p}^\star = (2, 0.1, 0.3)^{\mathrm{T}}$ par

```
truep = [2.;0.1;0.3];
t = [0;1;2;3;4;5;7;10;15;20;25;30;40;50;75;100];
data = ExpMod(truep,t);
plot(t,data,'o','MarkerEdgeColor','k',...
    'MarkerSize',7)
xlabel('Time')
ylabel('Output data')
hold on
```

Elles sont décrites par la figure 9.10.

Le vecteur $\widehat{\mathbf{p}}$ des paramètres du modèle sera estimé en minimisant soit la fonction de coût quadratique

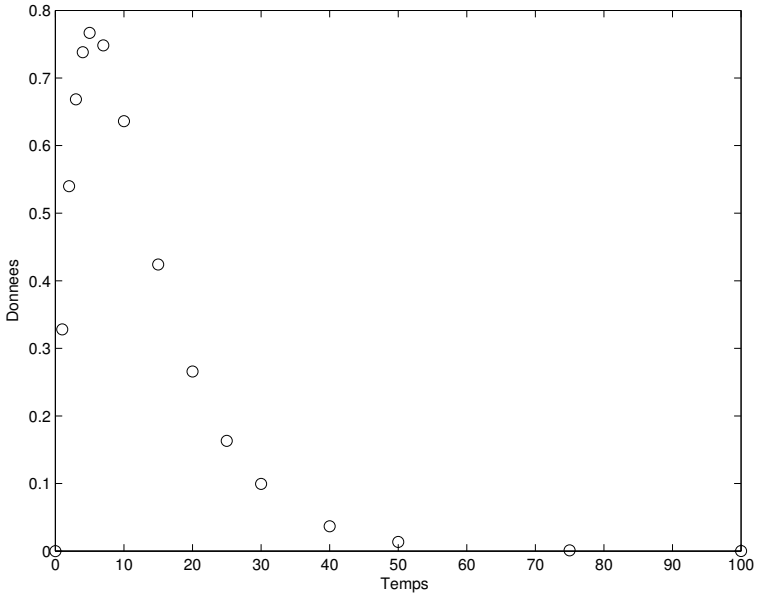


**Fig. 9.10** Données à utiliser pour l'exemple d'estimation de paramètres non linéaire

$$J(\mathbf{p}) = \sum_{i=1}^{16} [y_{\mathrm{M}}(t_i, \mathbf{p}^\star) - y_{\mathrm{M}}(t_i, \mathbf{p})]^2 \tag{9.235}$$

soit la fonction de coût $l_1$

$$J(\mathbf{p}) = \sum_{i=1}^{16} |y_{\mathrm{M}}(t_i, \mathbf{p}^\star) - y_{\mathrm{M}}(t_i, \mathbf{p})|. \tag{9.236}$$

Dans les deux cas, on s'attend à ce que $\widehat{\mathbf{p}}$ soit proche de $\mathbf{p}^\star$, et $J(\widehat{\mathbf{p}})$ de zéro. Tous les algorithmes sont initialisés à $\mathbf{p} = (1,1,1)^{\mathrm{T}}$.

#### 9.5.2.1 Utilisation du simplexe de Nelder et Mead sur un coût quadratique

La méthode du simplexe de Nelder et Mead est implémentée dans `fminsearch`, une fonction de l'*Optimization Toolbox*. La liste des options par défaut de cette fonction est obtenue grâce à l'instruction `optimset('fminsearch')`. Elle est plutôt longue, et commence par

```
Display: 'notify'
MaxFunEvals: '200*numberofvariables'
```

```
MaxIter: '200*numberofvariables'
TolFun: 1.000000000000000e-04
TolX: 1.000000000000000e-04
```

Ces options peuvent être changées via `optimset`. Ainsi, par exemple,

```
optionsFMS = optimset('Display','iter','TolX',1.e-8);
```

requiert que des informations soient fournies sur les itérations et change la tolérance sur les variables de décision de sa valeur standard $10^{-4}$ à $10^{-8}$ (voir la documentation pour les détails). Le script

```
p0 = [1;1;1]; % valeur initiale de pHat
optionsFMS = optimset('Display',...
    'iter','TolX',1.e-8);
[pHat,Jhat] = fminsearch(@(p) ...
    L2costExpMod(p,data,t),p0,optionsFMS)
finegridt = (0:100);
bestModel = ExpMod(pHat,finegridt);
plot(finegridt,bestModel)
ylabel('Data and best model output')
xlabel('Time')
```

appelle la fonction

```
function [J] = L2costExpMod(p,measured,times)
% Calcul du coût L2
modeled = ExpMod(p,times);
J = norm(measured-modeled)^2;
end
```

et produit

```
pHat =
     2.000000001386514e+00
     9.999999999868020e-02
     2.999999997322276e-01

Jhat =
     2.543904180521509e-19
```

après 393 évaluations de la fonction de coût et tous les types d'actions dont un algorithme du simplexe de Nelder et Mead est capable.

Les résultats de la simulation du meilleur modèle trouvé sont sur la figure 9.11, avec les données. Comme on pouvait s'y attendre, l'accord est visuellement parfait. Il en sera de même avec toutes les autres méthodes utilisées pour traiter ces données, de sorte qu'aucune autre figure de ce type ne sera présentée.

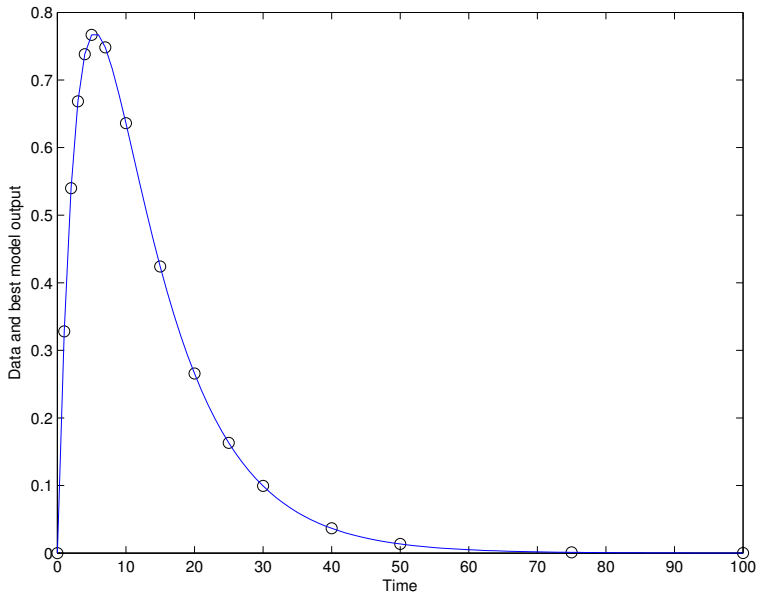


**Fig. 9.11** Ajustement aux moindres carrés des données de la figure 9.10, obtenu avec le simplexe de Nelder et Mead

#### 9.5.2.2 Utilisation du simplexe de Nelder et Mead sur un coût $l_1$

Le simplexe de Nelder et Mead peut aussi traiter des fonctions de coût non différentiables. Pour changer le coût de $l_2$ en $l_1$, il suffit de remplacer l'appel à la fonction `L2costExpMod` par un appel à

```
function [J] = L1costExpMod(p,measured,times)
% Calcul du coût L1
modeled = ExpMod(p,times);
J = norm(measured-modeled,1)
end
```

L'optimisation se révèle tout de même un peu plus difficile. Après avoir modifié les options de `fminsearch` suivant

```
p0 = [1;1;1];
optionsFMS = optimset('Display','iter',...
    'TolX',1.e-8,'MaxFunEvals',1000,'MaxIter',1000);
[pHat,Jhat] = fminsearch(@(p) L1costExpMod...
    (p,data,t),p0,optionsFMS)
```

on obtient

```
pHat =
     1.999999999761701e+00
     1.000000000015123e-01
     2.999999999356928e-01

Jhat =
     1.628779979759212e-09
```

après 753 évaluations de la fonction de coût.

#### 9.5.2.3 Utilisation d'une méthode de quasi-Newton

Il est très simple de remplacer l'utilisation du simplexe de Nelder et Mead par celle de la méthode BFGS. Il suffit d'utiliser `fminunc`, aussi fournie avec l'*Optimization Toolbox*, et de spécifier que le problème à traiter n'est pas de grande taille.

Le script

```
p0 = [1;1;1];
optionsFMU = optimset('LargeScale','off',...
    'Display','iter','TolX',1.e-8,'TolFun',1.e-10);
[pHat,Jhat] = fminunc(@(p) ...
    L2costExpMod(p,data,t),p0,optionsFMU)
```

produit

```
pHat =
     1.999990965496236e+00
     9.999973863180953e-02
     3.000007838651897e-01

Jhat =
     5.388400409913042e-13
```

après 178 évaluations de la fonction de coût, avec des gradients évalués par différences finies.

#### 9.5.2.4 Utilisation de la méthode de Levenberg et Marquardt

La méthode de Levenberg et Marquardt est mise en œuvre dans `lsqnonlin`, aussi fournie avec l'*Optimization Toolbox*. Au lieu d'une fonction évaluant le coût, `lsqnonlin`, requiert une fonction définie par l'utilisateur pour calculer chacun des résidus qu'il faut élever au carré et sommer pour obtenir le coût. Cette fonction peut s'écrire

```
function [residual] = ResExpModForLM(p,measured,times)
```

```
% calcule ce qu'il faut pour lsqnonlin pour L&M
[modeled] = ExpMod(p,times);
residual = measured - modeled;
end
```

et on peut l'utiliser dans le script

```
p0 = [1;1;1];
optionsLM = optimset('Display','iter',...
    'Algorithm','levenberg-marquardt')
% il faut fournir des bornes inf. et sup.
% pour ne pas déclencher un message d'erreur,
% bien que ces bornes ne soient pas utilisées...
lb = zeros(3,1);
lb(:) = -Inf;
ub = zeros(3,1);
ub(:) = Inf;
[pHat,Jhat] = lsqnonlin(@(p) ...
    ResExpModForLM(p,data,t),p0,lb,ub,optionsLM)
```

pour obtenir

```
pHat =
     1.999999999999992e+00
     9.999999999999978e-02
     3.000000000000007e-01

Jhat =
     7.167892101111147e-31
```

après seulement 51 évaluations du vecteur des résidus, avec des fonctions de sensibilité évaluées par différences finies.

Les options de chaque méthode mériteraient d'être réglées avec plus de soin qu'ici, ce qui rend les comparaisons difficiles. La méthode de Levengerg et Marquardt semble cependant gagner haut la main cette petite compétition. Ceci n'est guère surprenant car elle est particulièrement bien adaptée à des fonctions de coût quadratiques à valeur optimale proche de zéro et à un espace de recherche de dimension faible, comme ici.

## 9.6 En résumé

— Il est important de reconnaître quand la méthode des moindres carrés linéaires s'applique ou quand le problème est convexe, car il existe alors des algorithmes dédiés extrêmement puissants.
— Quand la méthode des moindres carrés linéaires s'applique, éviter de résoudre les équations normales, qui peuvent être numériquement désastreuses à cause

du calcul de $\mathbf{F}^T\mathbf{F}$ à moins que des conditions très particulières ne soient réunies. Préférer, si possible, l'approche fondée sur une factorisation QR ou une SVD de $\mathbf{F}$. La SVD fournit le conditionnement du problème pour la norme spectrale en sous-produit et permet la régularisation de problèmes mal conditionnés. Elle est par contre plus complexe que la factorisation QR et ne donne pas nécessairement des résultats plus précis.

— Quand la méthode des moindres carrés linéaires ne s'applique pas, la plupart des méthodes présentées sont itératives et *locales*. Elles convergent au mieux vers un minimiseur local, sans garantie qu'il soit global et unique (à moins que des propriétés complémentaires de la fonction de coût ne soient connues, comme sa convexité). Quand le temps nécessaire à une seule optimisation locale le permet, une stratégie *multistart* peut être utilisée dans une tentative d'échapper à l'attraction éventuelle de minimiseurs locaux parasites. C'est un exemple particulièrement simple d'optimisation globale par recherche aléatoire, sans garantie de succès non plus.
— La combinaison de recherches unidimensionnelles doit être faite avec précaution, car le fait de limiter les directions de recherche à des sous-espaces fixés peut interdire la convergence vers un minimiseur.
— Toutes les méthodes itératives fondées sur un développement de Taylor ne sont pas égales. Les meilleures démarrent comme des méthodes de gradient et terminent comme des méthodes de Newton. Tel est le cas des méthodes de quasi-Newton et de gradients conjugués.
— Quand la fonction de coût est quadratique en une erreur, la méthode de Gauss-Newton présente des avantages significatifs par rapport à la méthode de Newton. Elle est particulièrement efficace quand le minimum de la fonction de coût est proche de zéro.
— Les méthodes de gradients conjugués peuvent être préférées aux méthodes de quasi-Newton quand il y a beaucoup de variables de décision. Le prix à payer pour ce choix est qu'on ne disposera pas d'une estimée de l'inverse de la hessienne au minimiseur.
— A moins que la fonction de coût ne soit différentiable partout, toutes les méthodes locales à base de développement de Taylor sont vouées à échouer. La méthode de Nelder et Mead, qui n'utilise que des évaluations de la fonction de coût, est donc particulièrement intéressante pour les problèmes non-différentiables comme la minimisation d'une somme de valeurs absolues.
— L'optimisation robuste permet de se protéger contre l'effet de facteurs qu'on ne peut pas contrôler.
— Les techniques d'exploration aléatoire sont simples à mettre en œuvre mais ne peuvent garantir la localisation d'un minimiseur global de la fonction de coût.
— Les méthodes de *branch and bound* permettent de prouver des affirmations sur le minimum global et les minimiseurs globaux.
— Quand le budget pour évaluer la fonction de coût est sévèrement limité, on peut essayer l'*Efficient Global Optimization* (EGO), fondée sur un modèle de substitution obtenu par krigeage.

— La forme du front de Pareto peut aider à choisir le compromis le plus approprié quand des objectifs sont en conflit.

# Chapitre 10
# Optimiser sous contraintes

## 10.1 Introduction

De nombreux problèmes d'optimisation perdent tout sens si on néglige l'existence des contraintes qui contribuent à les définir. C'est pourquoi ce chapitre présente des techniques utilisables pour prendre ces contraintes en compte. On peut trouver plus d'informations dans des monographies comme [13, 28, 176]. La *révolution des points intérieurs* fournit un point de vue unificateur, agréablement documenté dans [255].

### *10.1.1 Analogie topographique*

Supposons que quelqu'un veuille minimiser son altitude $J(\mathbf{x})$, où $\mathbf{x}$ spécifie sa longitude $x_1$ et sa latitude $x_2$. Le fait de devoir marcher sur un chemin donné de largeur nulle se traduit par la contrainte d'égalité

$$c^{\mathrm{e}}(\mathbf{x}) = 0, \tag{10.1}$$

tandis que le fait de devoir rester sur un terrain donné se traduit par un ensemble de contraintes d'inégalité

$$\mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant \mathbf{0}. \tag{10.2}$$

Dans les deux cas, le voisinage de l'endroit d'altitude minimale peut ne pas être horizontal, ce qui revient à dire que le gradient de $J(\cdot)$ peut ne pas être nul en $\widehat{\mathbf{x}}$. Les conditions d'optimalité (et les méthodes d'optimisation résultantes) diffèrent donc de celles du cas sans contrainte.

### *10.1.2 Motivations*

Une première motivation pour introduire des contraintes sur $\mathbf{x}$ est d'*interdire des valeurs non réalistes des variables de décision*. Si, par exemple, le $i$-ème paramètre d'un modèle à estimer à partir de données expérimentales est la masse d'un être humain, on peut prendre

$$0 \leqslant x_i \leqslant 300 \text{ Kg}. \tag{10.3}$$

Ici, le minimiseur de la fonction de coût ne devrait pas être sur la frontière du domaine admissible, de sorte qu'aucune de ces deux contraintes d'inégalité ne devrait être active, sauf peut-être temporairement pendant la recherche. Elles ne jouent donc aucun rôle fondamental, et sont principalement utilisées a posteriori pour vérifier que les estimées obtenues pour les paramètres ne sont pas absurdes. Si $\widehat{x}_i$ obtenu par minimisation sans contrainte se révèle ne pas appartenir à $[0,300]$ Kg, alors le contraindre à le faire peut se traduire par $\widehat{x}_i = 0$ Kg ou $\widehat{x}_i = 300$ Kg, qui sont insatisfaisants l'un et l'autre.

Une seconde motivation est la prise en compte des *spécifications d'un cahier des charges, qui prennent souvent la forme de contraintes d'inégalité*, par exemple pour la conception assistée par ordinateur de produits industriels ou pour la conduite de processus. Certaines de ces contraintes d'inégalité sont en général saturées à l'optimum, et seraient violées si elles n'étaient pas prises en compte explicitement. Les contraintes peuvent porter sur des quantités qui dépendent de $\mathbf{x}$, de sorte que vérifier qu'un $\mathbf{x}$ donné appartient à $\mathbb{X}$ peut passer par la simulation d'un modèle numérique.

Une troisième motivation est le *traitement d'objectifs en conflit*, par l'optimisation de l'un d'entre eux sous des contraintes sur les autres. On peut, par exemple, minimiser le coût d'un lanceur spatial sous des contraintes sur sa charge utile, ou maximiser sa charge utile sous des contraintes sur son coût.

*Remarque 10.1.* En conception, les contraintes sont si cruciales que le rôle de la fonction de coût peut même devenir secondaire, comme de permettre le choix d'une solution ponctuelle $\widehat{\mathbf{x}}$ dans $\mathbb{X}$ tel que défini par les contraintes. On peut, par exemple, maximiser la distance euclidienne entre $\mathbf{x}$ dans $\mathbb{X}$ et le point le plus proche de la frontière $\partial\mathbb{X}$ de $\mathbb{X}$. Ceci assure une certaine robustesse à des fluctuations en production de masse des caractéristiques de composants du système en cours de conception. □

*Remarque 10.2.* Même si un minimiseur sans contrainte est strictement à l'intérieur de $\mathbb{X}$, il peut ne pas être optimal pour le problème sous contraintes, comme illustré par la figure 10.1. □

### *10.1.3 Propriétés souhaitables de l'ensemble admissible*

L'ensemble admissible $\mathbb{X}$ défini par les contraintes doit bien sûr contenir plusieurs éléments pour qu'une optimisation soit possible. S'il est facile de vérifier que tel est le cas sur de petits exemples académiques, cela devient un défi dans les problèmes industriels de grande taille, où $\mathbb{X}$ peut se révéler vide et où l'on peut devoir relaxer certaines contraintes.

$\mathbb{X}$ est supposé ici *compact*, c'est à dire *fermé* et *borné*. Il est fermé s'il contient sa frontière, ce qui interdit les contraintes d'inégalité strictes ; il est borné s'il est impossible de faire tendre la norme d'un vecteur $\mathbf{x}$ de $\mathbb{X}$ vers l'infini. Si la fonction de coût $J(\cdot)$ est continue sur $\mathbb{X}$ et si $\mathbb{X}$ est compact, alors le théorème de Weierstrass garantit l'existence d'un minimiseur global de $J(\cdot)$ sur $\mathbb{X}$. La figure 10.2 illustre le fait que la non-compacité de $\mathbb{X}$ peut entraîner l'absence de minimiseur.

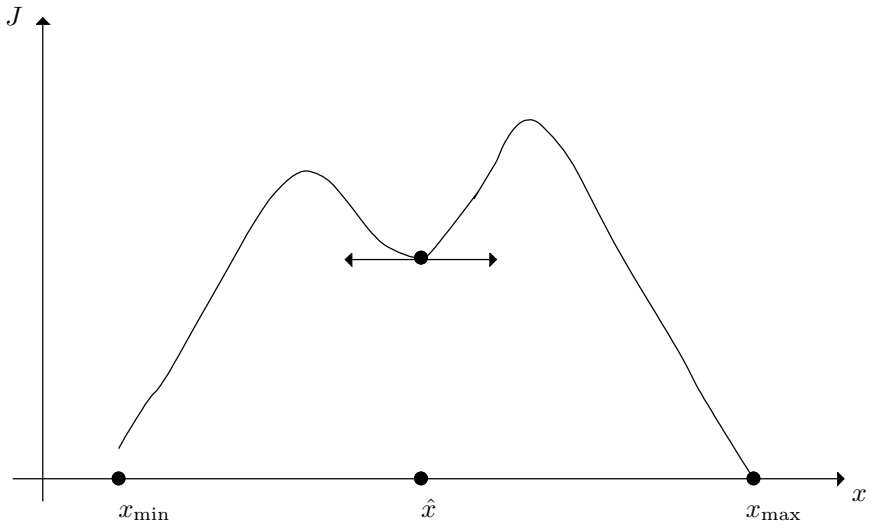


**Fig. 10.1** Bien qu'admissible, le minimiseur sans contrainte $\widehat{x}$ n'est pas optimal

### *10.1.4 Se débarasser de contraintes*

Il est parfois possible de transformer le problème pour éliminer des contraintes. Si, par exemple, $\mathbf{x}$ peut être partitionné en deux sous-vecteurs $\mathbf{x}_1$ et $\mathbf{x}_2$ liés par la contrainte

$$\mathbf{A}\mathbf{x}_1 + \mathbf{B}\mathbf{x}_2 = \mathbf{c}, \tag{10.4}$$

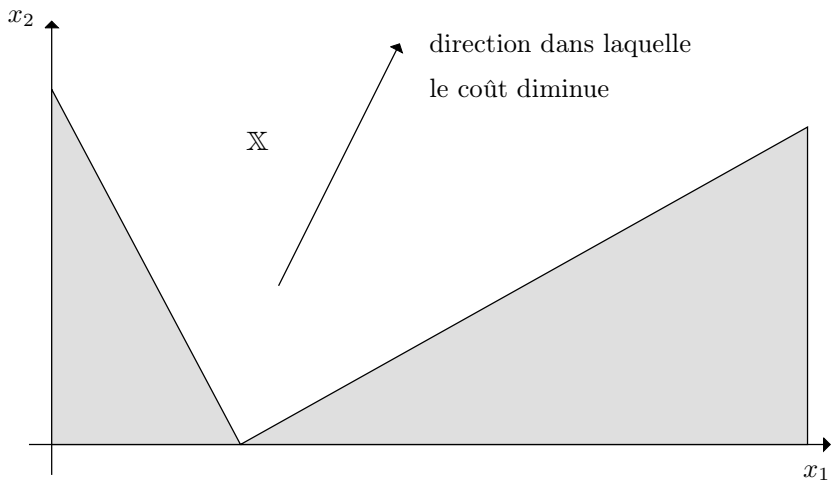


**Fig. 10.2** $\mathbb{X}$, la partie du premier quadrant en blanc, n'est pas compact et il n'y a pas de minimiseur

avec $\mathbf{A}$ inversible, alors on peut exprimer $\mathbf{x}_1$ en fonction de $\mathbf{x}_2$ :

$$\mathbf{x}_1 = \mathbf{A}^{-1}(\mathbf{c} - \mathbf{B}\mathbf{x}_2). \tag{10.5}$$

Ceci diminue la dimension de l'espace de recherche et élimine le besoin de prendre la contrainte (10.4) en considération. Cette stratégie peut cependant avoir des conséquences négatives sur la structure de certaines des équations à résoudre, en les rendant moins creuses.

Un changement de variable peut permettre d'éliminer des contraintes d'inégalité. Pour imposer la contrainte $x_i > 0$, par exemple, il suffit de remplacer $x_i$ par $\exp q_i$, et les contraintes $a < x_i < b$ peuvent être imposées en posant

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \tanh q_i. \tag{10.6}$$

Quand de telles transformations sont impossibles ou pas souhaitables, les algorithmes et les conditions théoriques d'optimalité doivent tenir compte des contraintes.

*Remarque 10.3.* Quand il y a un mélange de contraintes linéaires et non linéaires, c'est souvent une bonne idée de traiter les contraintes linéaires à part, pour tirer profit de l'algèbre linéaire (voir le chapitre 5 de [73]). □

## 10.2 Conditions théoriques d'optimalité

Tout comme dans le cas sans contrainte, les conditions théoriques d'optimalité sont utilisées pour concevoir des algorithmes d'optimisation et des critères d'arrêt. Insistons sur la différence qui suit.

> Contrairement à ce qui se passe en optimisation sans contrainte, le gradient de la fonction de coût peut ne pas être nul en un minimiseur. Il faut donc établir des conditions d'optimalité spécifiques.

### *10.2.1 Contraintes d'égalité*

Supposons, pour commencer, que

$$\mathbb{X} = \{\mathbf{x} : \mathbf{c}^{\mathrm{e}}(\mathbf{x}) = \mathbf{0}\}, \tag{10.7}$$

où le nombre des contraintes d'égalité scalaires est $n_{\mathrm{e}} = \dim \mathbf{c}^{\mathrm{e}}(\mathbf{x})$. Il est important de noter que les contraintes d'égalité doivent être écrites sous la forme standard prescrite par (10.7) pour que les résultats qui vont être établis soient corrects. La contrainte

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{10.8}$$

se traduit par exemple par

$$\mathbf{c}^{\mathrm{e}}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}. \tag{10.9}$$

Supposons de plus que

— les $n_{\mathrm{e}}$ contraintes d'égalité définissant $\mathbb{X}$ sont indépendantes (aucune ne peut être retirée sans changer $\mathbb{X}$) et compatibles ($\mathbb{X}$ n'est pas vide),

— le nombre $n = \dim \mathbf{x}$ des variables de décision est strictement supérieur à $n_{\mathrm{e}}$ (on peut effectuer des mouvements infinitésimaux $\boldsymbol{\delta}\mathbf{x}$ sans quitter $\mathbb{X}$),

— les contraintes et la fonction de coût sont différentiables.

La condition nécessaire (9.4) pour que $\widehat{\mathbf{x}}$ soit un minimiseur (au moins localement) devient

$$\widehat{\mathbf{x}} \in \mathbb{X} \quad \text{et} \quad \mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\boldsymbol{\delta}\mathbf{x} \geqslant 0 \quad \forall \boldsymbol{\delta}\mathbf{x} : \widehat{\mathbf{x}} + \boldsymbol{\delta}\mathbf{x} \in \mathbb{X}. \tag{10.10}$$

La condition (10.10) doit encore être satisfaite quand $\boldsymbol{\delta}\mathbf{x}$ est remplacé par $-\boldsymbol{\delta}\mathbf{x}$, de sorte qu'on peut la remplacer par

$$\widehat{\mathbf{x}} \in \mathbb{X} \quad \text{et} \quad \mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})\boldsymbol{\delta}\mathbf{x} = 0 \quad \forall \boldsymbol{\delta}\mathbf{x} : \widehat{\mathbf{x}} + \boldsymbol{\delta}\mathbf{x} \in \mathbb{X}. \tag{10.11}$$

Le gradient $\mathbf{g}(\widehat{\mathbf{x}})$ du coût en un minimiseur $\widehat{\mathbf{x}}$ sous contraintes doit donc être orthogonal à tout déplacement $\boldsymbol{\delta}\mathbf{x}$ localement autorisé. Comme $\mathbb{X}$ diffère maintenant de

$\mathbb{R}^n$, ceci n'implique plus que $\mathbf{g}(\widehat{\mathbf{x}}) = \mathbf{0}$. Au premier ordre,

$$c_i^{\mathrm{e}}(\widehat{\mathbf{x}} + \delta\mathbf{x}) \approx c_i^{\mathrm{e}}(\widehat{\mathbf{x}}) + \left(\frac{\partial c_i^{\mathrm{e}}}{\partial \mathbf{x}}(\widehat{\mathbf{x}})\right)^{\mathrm{T}} \delta\mathbf{x}, \quad i = 1, \cdots, n_{\mathrm{e}}. \tag{10.12}$$

Puisque $c_i^{\mathrm{e}}(\widehat{\mathbf{x}} + \delta\mathbf{x}) = c_i^{\mathrm{e}}(\widehat{\mathbf{x}}) = 0$, ceci implique que

$$\left(\frac{\partial c_i^{\mathrm{e}}}{\partial \mathbf{x}}(\widehat{\mathbf{x}})\right)^{\mathrm{T}} \delta\mathbf{x} = 0, \quad i = 1, \cdots, n_{\mathrm{e}}. \tag{10.13}$$

Le déplacement $\delta\mathbf{x}$ doit donc être orthogonal aux vecteurs

$$\mathbf{v}_i = \frac{\partial c_i^{\mathrm{e}}}{\partial \mathbf{x}}(\widehat{\mathbf{x}}), \quad i = 1, \cdots, n_{\mathrm{e}}, \tag{10.14}$$

qui correspondent à des directions localement interdites. Puisque $\delta\mathbf{x}$ est orthogonal aux directions localement interdites et à $\mathbf{g}(\widehat{\mathbf{x}})$, $\mathbf{g}(\widehat{\mathbf{x}})$ est une combinaison linéaire de directions localement interdites, de sorte que

$$\frac{\partial J}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) + \sum_{i=1}^{n_{\mathrm{e}}} \lambda_i \frac{\partial c_i^{\mathrm{e}}}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) = 0. \tag{10.15}$$

Définissons le *lagrangien* comme

$$L(\mathbf{x}, \boldsymbol{\lambda}) = J(\mathbf{x}) + \sum_{i=1}^{n_{\mathrm{e}}} \lambda_i c_i^{\mathrm{e}}(\mathbf{x}), \tag{10.16}$$

où $\boldsymbol{\lambda}$ est le vecteur des *multiplicateurs de Lagrange* $\lambda_i$, $i = 1, \cdots, n_{\mathrm{e}}$. De façon équivalente,

$$L(\mathbf{x}, \boldsymbol{\lambda}) = J(\mathbf{x}) + \boldsymbol{\lambda}^{\mathrm{T}} \mathbf{c}^{\mathrm{e}}(\mathbf{x}). \tag{10.17}$$

**Proposition 10.1.** *Si $\widehat{\mathbf{x}}$ et $\widehat{\boldsymbol{\lambda}}$ sont tels que*

$$L(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}) = \min_{\mathbf{x} \in \mathbb{R}^n} \max_{\boldsymbol{\lambda} \in \mathbb{R}^{n_{\mathrm{e}}}} L(\mathbf{x}, \boldsymbol{\lambda}), \tag{10.18}$$

*alors*

1. *les contraintes sont satisfaites :*

$$\mathbf{c}^{\mathrm{e}}(\widehat{\mathbf{x}}) = \mathbf{0}, \tag{10.19}$$

2. *$\widehat{\mathbf{x}}$ est un minimiseur global de la fonction de coût $J(\cdot)$ sur $\mathbb{X}$ tel que défini par les contraintes,*
3. *tout minimiseur global de $J(\cdot)$ sur $\mathbb{X}$ est tel que (10.18) soit satisfaite.* □

*Preuve.* 1. L'équation (10.18) équivaut à

$$L(\widehat{\mathbf{x}}, \boldsymbol{\lambda}) \leqslant L(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}) \leqslant L(\mathbf{x}, \widehat{\boldsymbol{\lambda}}). \tag{10.20}$$

S'il existait une contrainte violée $c_i^{\mathrm{e}}(\widehat{\mathbf{x}}) \neq 0$, alors il suffirait de remplacer $\widehat{\lambda}_i$ par $\widehat{\lambda}_i + \text{signe } c_i^{\mathrm{e}}(\widehat{\mathbf{x}})$ tout en laissant $\widehat{\mathbf{x}}$ et les autres composantes de $\widehat{\boldsymbol{\lambda}}$ inchangés pour faire croître la valeur du lagrangien de $|c_i^{\mathrm{e}}(\widehat{\mathbf{x}})|$, ce qui contredirait la première inégalité de (10.20).

2. Supposons qu'il existe $\mathbf{x}$ dans $\mathbb{X}$ tel que $J(\mathbf{x}) < J(\widehat{\mathbf{x}})$. Puisque

$$\mathbf{c}^{\mathrm{e}}(\mathbf{x}) = \mathbf{c}^{\mathrm{e}}(\widehat{\mathbf{x}}) = \mathbf{0}, \tag{10.21}$$

(10.17) implique que $J(\mathbf{x}) = L(\mathbf{x}, \widehat{\boldsymbol{\lambda}})$ et $J(\widehat{\mathbf{x}}) = L(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}})$. On aurait alors $L(\mathbf{x}, \widehat{\boldsymbol{\lambda}}) < L(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}})$, ce qui contredirait la seconde inégalité de (10.20).

3. Soit $\widehat{\mathbf{x}}$ un minimiseur global de $J(\cdot)$ sur $\mathbb{X}$. Pour tout $\boldsymbol{\lambda}$ dans $\mathbb{R}^{n_{\mathrm{e}}}$,

$$L(\widehat{\mathbf{x}}, \boldsymbol{\lambda}) = J(\widehat{\mathbf{x}}), \tag{10.22}$$

ce qui implique que

$$L(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}) = L(\widehat{\mathbf{x}}, \boldsymbol{\lambda}). \tag{10.23}$$

De plus, pour tout $\mathbf{x}$ dans $\mathbb{X}$, $J(\widehat{\mathbf{x}}) \leqslant J(\mathbf{x})$, de sorte que

$$L(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}) \leqslant L(\mathbf{x}, \widehat{\boldsymbol{\lambda}}). \tag{10.24}$$

Les inégalités (10.20) sont donc satisfaites, ce qui implique que (10.18) l'est aussi.

□

Ces résultats ont été établis sans supposer le lagrangien différentiable. Quand il l'est, les conditions nécessaires d'optimalité au premier ordre se traduisent par

$$\frac{\partial L}{\partial \mathbf{x}}(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}) = \mathbf{0}, \tag{10.25}$$

qui équivaut à (10.15), et

$$\frac{\partial L}{\partial \boldsymbol{\lambda}}(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}) = \mathbf{0}, \tag{10.26}$$

qui équivaut à $\mathbf{c}^{\mathrm{e}}(\widehat{\mathbf{x}}) = \mathbf{0}$.

> Le lagrangien permet ainsi d'éliminer formellement les contraintes du problème. La stationnarité du lagrangien garantit la satisfaction de ces contraintes.

On peut aussi définir des conditions d'optimalité du second ordre . Une condition nécessaire pour l'optimalité de $\widehat{\mathbf{x}}$ est que la hessienne du coût soit définie non négative sur l'espace tangent aux contraintes en $\widehat{\mathbf{x}}$. On obtient une condition suffisante d'optimalité (locale) en remplaçant *définie non négative* par *définie positive*, pourvu que les conditions d'optimalité du premier ordre soient aussi satisfaites.

*Exemple 10.1. Optimisation de forme*

On veut minimiser la surface de la feuille de métal à utiliser pour construire une boîte de conserve cylindrique de volume donné $V_0$. Les variables de conception sont la hauteur $h$ de la boîte et le rayon $r$ de sa base, de sorte que $\mathbf{x} = (h,r)^{\mathrm{T}}$. La surface à minimiser est

$$J(\mathbf{x}) = 2\pi r^2 + 2\pi rh, \tag{10.27}$$

et la contrainte sur le volume est

$$\pi r^2 h = V_0. \tag{10.28}$$

Sous la forme standard (10.7), cette contrainte devient

$$\pi r^2 h - V_0 = 0. \tag{10.29}$$

Le lagrangien peut donc s'écrire

$$L(\mathbf{x},\lambda) = 2\pi r^2 + 2\pi rh + \lambda(\pi r^2 h - V_0). \tag{10.30}$$

Une condition nécessaire pour que $(\widehat{h},\widehat{r},\widehat{\lambda})$ soit optimal est que

$$\frac{\partial L}{\partial h}(\widehat{h},\widehat{r},\widehat{\lambda}) = 2\pi\widehat{r} + \pi\widehat{r}^2\widehat{\lambda} = 0, \tag{10.31}$$

$$\frac{\partial L}{\partial r}(\widehat{h},\widehat{r},\widehat{\lambda}) = 4\pi\widehat{r} + 2\pi\widehat{h} + 2\pi\widehat{r}\widehat{h}\widehat{\lambda} = 0, \tag{10.32}$$

$$\frac{\partial L}{\partial \lambda}(\widehat{h},\widehat{r},\widehat{\lambda}) = \pi\widehat{r}^2\widehat{h} - V_0 = 0. \tag{10.33}$$

L'équation (10.31) implique que

$$\widehat{\lambda} = -\frac{2}{\widehat{r}}. \tag{10.34}$$

Avec (10.32), ceci implique que

$$\widehat{h} = 2\widehat{r}. \tag{10.35}$$

La hauteur de la boîte doit donc être égale à son diamètre. Portons (10.35) dans (10.33) pour obtenir

$$2\pi\widehat{r}^3 = V_0, \tag{10.36}$$

de sorte que

$$\widehat{r} = \left(\frac{V_0}{2\pi}\right)^{\frac{1}{3}} \quad \text{et} \quad \widehat{h} = 2\left(\frac{V_0}{2\pi}\right)^{\frac{1}{3}}. \tag{10.37}$$

□

### *10.2.2 Contraintes d'inégalité*

Rappelons que s'il y a des contraintes d'inégalité strictes il peut ne pas y avoir de minimiseur (voir, par exemple, la minimisation de $J(x) = -x$ sous la contrainte $x < 1$). C'est pourquoi nous supposons que les contraintes d'inégalité peuvent s'écrire sous la forme standard

$$\mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant \mathbf{0}, \tag{10.38}$$

à comprendre composante par composante, c'est à dire comme signifiant que

$$c^{\mathrm{i}}_j(\mathbf{x}) \leqslant 0, \quad j = 1, \cdots, n_{\mathrm{i}}, \tag{10.39}$$

où le nombre $n_{\mathrm{i}} = \dim \mathbf{c}^{\mathrm{i}}(\mathbf{x})$ des contraintes d'inégalité scalaires peut être plus grand que $\dim \mathbf{x}$. Il est important de noter que les contraintes d'inégalité doivent être écrites sous la forme standard prescrite par (10.39) pour que les résultats qui vont être établis soient corrects.

Les contraintes d'inégalité peuvent être transformées en contraintes d'égalité en écrivant

$$c^{\mathrm{i}}_j(\mathbf{x}) + y_j^2 = 0, \quad j = 1, \cdots, n_{\mathrm{i}}, \tag{10.40}$$

où $y_j$ est une *variable d'écart*, qui prend la valeur zéro quand la $j$-ème contrainte d'inégalité scalaire est *active* (c'est à dire qu'elle se comporte comme une contrainte d'égalité). Quand $c^{\mathrm{i}}_j(\mathbf{x}) = 0$ , on dit aussi que la $j$-ème contrainte d'inégalité est *saturée*. (Quand $c^{\mathrm{i}}_j(\mathbf{x}) > 0$, la $j$-ème contrainte d'inégalité est dite *violée*.)

Le lagrangien associé aux contraintes d'égalité (10.40) est

$$L(\mathbf{x}, \boldsymbol{\mu}, \mathbf{y}) = J(\mathbf{x}) + \sum_{j=1}^{n_{\mathrm{i}}} \mu_j \left[ c^{\mathrm{i}}_j(\mathbf{x}) + y_j^2 \right]. \tag{10.41}$$

Quand on traite des contraintes d'inégalités telles que (10.39), les multiplicateurs de Lagrange $\mu_j$ obtenus de cette manière sont souvent appelés *coefficients de Kuhn et Tucker*. Si les contraintes et la fonction de coût sont différentiables, alors les conditions du premier ordre pour la stationnarité du Lagrangien sont

$$\frac{\partial L}{\partial \mathbf{x}}(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\mu}}, \widehat{\mathbf{y}}) = \frac{\partial J}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) + \sum_{j=1}^{n_{\mathrm{i}}} \widehat{\mu}_j \frac{\partial c^{\mathrm{i}}_j}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) = \mathbf{0}, \tag{10.42}$$

$$\frac{\partial L}{\partial \mu_j}(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\mu}}, \widehat{\mathbf{y}}) = c^{\mathrm{i}}_j(\widehat{\mathbf{x}}) + \widehat{y}_j^2 = 0, \quad j = 1, \cdots, n_{\mathrm{i}}, \tag{10.43}$$

$$\frac{\partial L}{\partial y_j}(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\mu}}, \widehat{\mathbf{y}}) = 2\widehat{\mu}_j \widehat{y}_j = 0, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{10.44}$$

Quand la $j$-ème contrainte d'inégalité est inactive, $\widehat{y}_j \neq 0$ et (10.44) implique que la valeur optimale du multiplicateur de Lagrange associé $\widehat{\mu}_j$ est nulle. *C'est comme si cette contrainte n'existait pas*. La condition (10.42) traite les contraintes actives

comme si elles étaient des contraintes d'égalité. Quant à la condition (10.43), elle se borne à imposer les contraintes.

On peut aussi obtenir des conditions d'optimalité du second ordre impliquant la hessienne du lagrangien. Cette hessienne est diagonale par blocs. Le bloc qui correspond aux déplacements dans l'espace des variables d'écart est lui même diagonal, avec des éléments diagonaux donnés par

$$\frac{\partial^2 L}{\partial y_j^2} = 2\mu_j, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{10.45}$$

Pourvu que $J(\cdot)$ soit à minimiser, comme supposé ici, une condition nécessaire d'optimalité est que la hessienne soit définie non négative dans le sous-espace autorisé par les contraintes, ce qui implique que

$$\widehat{\mu}_j \geqslant 0, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{10.46}$$

*Remarque 10.4.* Ceci est à comparer avec le cas des contraintes d'égalité, pour lesquelles il n'y a pas de contrainte sur le signe des multiplicateurs de Lagrange. □

*Remarque 10.5.* Les conditions (10.46) correspondent à une *minimisation* avec des contraintes écrites sous la forme $\mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant \mathbf{0}$. Pour une maximisation ou si certaines contraintes étaient sous la forme $c_j^{\mathrm{i}}(\mathbf{x}) \geqslant 0$, les conditions différeraient. □

*Remarque 10.6.* On peut écrire le lagrangien sans introduire les variables d'écart $y_j$, pourvu qu'on n'oublie pas que (10.46) doit être satisfaite et que $\widehat{\mu}_j c_j^{\mathrm{i}}(\mathbf{x}) = 0$, pour $j = 1, \cdots, n_{\mathrm{i}}$. □

Il faut considérer toutes les combinaisons possibles de contraintes d'inégalité saturées, depuis le cas où aucune n'est saturée jusqu'à ceux où toutes les contraintes qui peuvent être saturées simultanément le sont.

*Exemple 10.2.* Pour trouver la ou les positions d'altitude minimale à l'intérieur d'un terrain carré $\mathbb{X}$ défini par quatre contraintes d'inégalité, il faut considérer neuf cas, à savoir

— aucune de ces contraintes n'est active (le minimiseur peut être à l'intérieur du terrain),
— l'une quelconque des quatre contraintes est active (le minimiseur peut être sur un des côtés du carré),
— l'une quelconque des quatre paires de contraintes compatibles est active (le minimiseur peut être sur un des sommets du carré).

Il faut enfin comparer tous les candidats au titre de minimiseur ainsi détectés en termes de valeur prise par la fonction de coût, après avoir vérifié qu'ils appartiennent bien à $\mathbb{X}$. Le ou les minimiseurs globaux peuvent être strictement à l'intérieur du terrain, mais l'existence d'un seul minimiseur strictement à l'intérieur du terrain n'impliquerait pas que ce minimiseur soit globalement optimal. □

*Exemple 10.3.* La fonction de coût $J(\mathbf{x}) = x_1^2 + x_2^2$ doit être minimisée sous la contrainte $x_1^2 + x_2^2 + x_1 x_2 \geqslant 1$. Le lagrangien du problème est donc

$$L(\mathbf{x},\mu) = x_1^2 + x_2^2 + \mu(1 - x_1^2 - x_2^2 - x_1x_2). \tag{10.47}$$

Des conditions nécessaires d'optimalité sont $\widehat{\mu} \geqslant 0$ et

$$\frac{\partial L}{\partial \mathbf{x}}(\widehat{\mathbf{x}}, \widehat{\mu}) = \mathbf{0}. \tag{10.48}$$

La condition (10.48) peut s'écrire

$$\mathbf{A}(\widehat{\mu})\widehat{\mathbf{x}} = \mathbf{0}, \tag{10.49}$$

avec

$$\mathbf{A}(\widehat{\mu}) = \begin{bmatrix} 2(1-\widehat{\mu}) & -\widehat{\mu} \\ -\widehat{\mu} & 2(1-\widehat{\mu}) \end{bmatrix}. \tag{10.50}$$

La solution triviale $\widehat{\mathbf{x}} = \mathbf{0}$ viole la contrainte, de sorte que $\widehat{\mu}$ est tel que $\det \mathbf{A}(\widehat{\mu}) = 0$, ce qui implique que $\widehat{\mu} = 2$ ou $\widehat{\mu} = 2/3$. Comme les deux valeurs possibles des coefficients de Kuhn et Tucker sont strictement positives, la contrainte d'inégalité est saturée et peut être traitée comme une contrainte d'égalité

$$x_1^2 + x_2^2 + x_1x_2 = 1. \tag{10.51}$$

Si $\widehat{\mu} = 2$, alors (10.49) implique que $\widehat{x}_1 = -\widehat{x}_2$ et les deux solutions de (10.51) sont $\widehat{\mathbf{x}}^1 = (1,-1)^{\mathrm{T}}$ et $\widehat{\mathbf{x}}^2 = (-1,1)^{\mathrm{T}}$, avec $J(\widehat{\mathbf{x}}^1) = J(\widehat{\mathbf{x}}^2) = 2$. Si $\widehat{\mu} = 2/3$, alors (10.49) implique que $\widehat{x}_1 = \widehat{x}_2$ et les deux solutions de (10.51) sont $\widehat{\mathbf{x}}^3 = (1/\sqrt{3}, 1/\sqrt{3})^{\mathrm{T}}$ et $\widehat{\mathbf{x}}^4 = (-1/\sqrt{3}, -1/\sqrt{3})^{\mathrm{T}}$, avec $J(\widehat{\mathbf{x}}^3) = J(\widehat{\mathbf{x}}^4) = 2/3$. Il y a donc deux minimiseurs globaux, $\widehat{\mathbf{x}}^3$ et $\widehat{\mathbf{x}}^4$. □

*Exemple 10.4. Projection sur une tranche*

Nous voulons projeter un vecteur $\mathbf{p} \in \mathbb{R}^n$ connu numériquement sur l'ensemble

$$\mathbb{S} = \{\mathbf{v} \in \mathbb{R}^n : -b \leqslant y - \mathbf{f}^{\mathrm{T}}\mathbf{v} \leqslant b\}, \tag{10.52}$$

où $y \in \mathbb{R}$, $b \in \mathbb{R}^+$ et $\mathbf{f} \in \mathbb{R}^n$ sont connus numériquement. $\mathbb{S}$ est la tranche située entre les hyperplans $\mathbb{H}^+$ et $\mathbb{H}^-$ dans $\mathbb{R}^n$, décrits par les équations

$$\mathbb{H}^+ = \{\mathbf{v} : y - \mathbf{f}^{\mathrm{T}}\mathbf{v} = b\}, \tag{10.53}$$

$$\mathbb{H}^- = \{\mathbf{v} : y - \mathbf{f}^{\mathrm{T}}\mathbf{v} = -b\}. \tag{10.54}$$

($\mathbb{H}^+$ et $\mathbb{H}^-$ sont tous les deux orthogonaux à $\mathbf{f}$, de sorte qu'ils sont parallèles.) Cette opération est au cœur de l'approche d'estimation creuse décrite dans [230].

Le résultat $\widehat{\mathbf{x}}$ de la projection sur $\mathbb{S}$ peut être calculé comme

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}\in\mathbb{S}} \|\mathbf{x} - \mathbf{p}\|_2^2. \tag{10.55}$$

Le lagrangien du problème est donc

$$L(\mathbf{x},\boldsymbol{\mu}) = (\mathbf{x}-\mathbf{p})^{\mathrm{T}}(\mathbf{x}-\mathbf{p}) + \mu_1(y - \mathbf{f}^{\mathrm{T}}\mathbf{x} - b) + \mu_2(-y + \mathbf{f}^{\mathrm{T}}\mathbf{x} - b). \tag{10.56}$$

Quand $\mathbf{p}$ est appartient à $\mathbb{S}$, la solution optimale est bien sûr $\widehat{\mathbf{x}} = \mathbf{p}$, et les deux coefficients de Kuhn et Tucker sont nuls. Quand $\mathbf{p}$ n'appartient pas à $\mathbb{S}$, une seule des contraintes d'inégalité est violée et la projection rendra cette contrainte active. Supposons, par exemple, que la contrainte

$$y - \mathbf{f}^{\mathrm{T}}\mathbf{p} \leqslant b \tag{10.57}$$

soit violée. Le lagrangien se simplifie alors en

$$L(\mathbf{x}, \mu_1) = (\mathbf{x}-\mathbf{p})^{\mathrm{T}}(\mathbf{x}-\mathbf{p}) + \mu_1(y - \mathbf{f}^{\mathrm{T}}\mathbf{x} - b). \tag{10.58}$$

Des conditions nécessaires d'optimalité au premier ordre sont

$$\frac{\partial L}{\partial \mathbf{x}}(\widehat{\mathbf{x}}, \widehat{\mu}_1) = \mathbf{0} = 2(\widehat{\mathbf{x}}-\mathbf{p}) - \widehat{\mu}_1\mathbf{f}, \tag{10.59}$$

$$\frac{\partial L}{\partial \mu_1}(\widehat{\mathbf{x}}, \widehat{\mu}_1) = 0 = y - \mathbf{f}^{\mathrm{T}}\widehat{\mathbf{x}} - b. \tag{10.60}$$

L'unique solution de ce système d'équations linéaires est

$$\widehat{\mu}_1 = 2\left(\frac{y - \mathbf{f}^{\mathrm{T}}\mathbf{p} - b}{\mathbf{f}^{\mathrm{T}}\mathbf{f}}\right), \tag{10.61}$$

$$\widehat{\mathbf{x}} = \mathbf{p} + \frac{\mathbf{f}}{\mathbf{f}^{\mathrm{T}}\mathbf{f}}(y - \mathbf{f}^{\mathrm{T}}\mathbf{p} - b), \tag{10.62}$$

et $\widehat{\mu}_1$ est positif, comme il convient. □

### *10.2.3 Cas général : conditions KKT*

Supposons maintenant que $J(\mathbf{x})$ doive être minimisé sous $\mathbf{c}^{\mathrm{e}}(\mathbf{x}) = \mathbf{0}$ et $\mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant \mathbf{0}$. Le lagrangien peut alors s'écrire

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = J(\mathbf{x}) + \boldsymbol{\lambda}^{\mathrm{T}}\mathbf{c}^{\mathrm{e}}(\mathbf{x}) + \boldsymbol{\mu}^{\mathrm{T}}\mathbf{c}^{\mathrm{i}}(\mathbf{x}), \tag{10.63}$$

et chaque coefficient de Kuhn et Tucker optimal $\widehat{\mu}_j$ doit satisfaire $\widehat{\mu}_j c^{\mathrm{i}}_j(\mathbf{x}) = 0$ et $\widehat{\mu}_j \geqslant 0$. Des conditions nécessaires d'optimalité peuvent être résumées en ce qui est connu comme les *conditions de Karush, Kuhn et Tucker* (KKT) :

$$\frac{\partial L}{\partial \mathbf{x}}(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{\mu}}) = \frac{\partial J}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) + \sum_{i=1}^{n_{\mathrm{e}}} \widehat{\lambda}_i \frac{\partial c^{\mathrm{e}}_j}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) + \sum_{j=1}^{n_{\mathrm{i}}} \widehat{\mu}_j \frac{\partial c^{\mathrm{i}}_j}{\partial \mathbf{x}}(\widehat{\mathbf{x}}) = \mathbf{0}, \tag{10.64}$$

$$\mathbf{c}^{\mathrm{e}}(\widehat{\mathbf{x}}) = \mathbf{0}, \quad \mathbf{c}^{\mathrm{i}}(\widehat{\mathbf{x}}) \leqslant \mathbf{0}, \tag{10.65}$$

$$\widehat{\boldsymbol{\mu}} \geqslant \mathbf{0}, \quad \widehat{\mu}_j c^{\mathrm{i}}_j(\widehat{\mathbf{x}}) = \mathbf{0}, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{10.66}$$

Il ne peut y avoir plus de $\dim \mathbf{x}$ contraintes indépendantes actives simultanément. (Les contraintes actives sont les contraintes d'inégalité saturées et les contraintes d'égalité.)

## 10.3 Résoudre les équations KKT avec la méthode de Newton

Une recherche formelle exhaustive de tous les points de l'espace de décision qui satisfont les conditions KKT n'est possible que pour des problèmes académiques relativement simples, de sorte qu'on la remplace en général par du calcul numérique. Pour chaque combinaison de contraintes actives possible, les conditions KKT se traduisent par un ensemble d'équations non linéaires, qui peuvent être résolues en utilisant la méthode de Newton (amortie) avant de vérifier si la solution ainsi calculée appartient à $\mathbb{X}$ et si les conditions de signe sur les coefficients de Kuhn et Tucker sont satisfaites. Rappelons toutefois que

— la satisfaction des conditions KKT ne garantit pas qu'un minimiseur a été atteint,

— même si un minimiseur a été trouvé, la recherche a seulement été locale, de sorte qu'une stratégie *multistart* peut rester à l'ordre du jour.

## 10.4 Utiliser des fonctions de pénalisation ou barrières

Au moins conceptuellement, la façon la plus simple de traiter des contraintes est via des fonctions de pénalisation ou des fonctions barrières.

Les *fonctions de pénalisation* modifient la fonction de coût $J(\cdot)$ de façon à traduire la violation de contraintes en une augmentation du coût. Il devient alors possible de retomber sur les méthodes classiques de minimisation sans contrainte. La fonction de coût initiale peut par exemple être remplacée par

$$J_\alpha(\mathbf{x}) = J(\mathbf{x}) + \alpha p(\mathbf{x}), \tag{10.67}$$

où $\alpha$ est un coefficient positif (à choisir par l'utilisateur) et où la pénalisation $p(\mathbf{x})$ croît avec la sévérité de la violation de la contrainte. On peut aussi utiliser plusieurs fonctions de pénalisation avec des coefficients multiplicateurs différents. Bien que $J_\alpha(\mathbf{x})$ ressemble un peu à un lagrangien, il n'y a pas ici d'optimisation de $\alpha$.

Les *fonctions barrières* utilisent aussi (10.67) ou une de ses variantes, mais avec $p(\mathbf{x})$ qui augmente dès que $\mathbf{x}$ s'approche de la frontière $\partial\mathbb{X}$ de $\mathbb{X}$ *de l'intérieur*, c'est à dire avant toute violation de contrainte (les fonctions barrières peuvent traiter des contraintes d'inégalité, pourvu que l'intérieur de $\mathbb{X}$ ne soit pas vide, mais pas des contraintes d'égalité).

### *10.4.1 Fonctions de pénalisation*

Avec les fonctions de pénalisation, $p(\mathbf{x})$ reste nul tant que $\mathbf{x}$ appartient à $\mathbb{X}$ mais croît avec la violation des contraintes. Pour $n_\mathrm{e}$ contraintes d'égalité, on peut prendre par exemple une fonction de pénalisation $l_2$

$$p_1(\mathbf{x}) = \sum_{i=1}^{n_\mathrm{e}} [c_i^\mathrm{e}(\mathbf{x})]^2, \tag{10.68}$$

ou une fonction de pénalisation $l_1$

$$p_2(\mathbf{x}) = \sum_{i=1}^{n_\mathrm{e}} |c_i^\mathrm{e}(\mathbf{x})|. \tag{10.69}$$

Pour $n_\mathrm{i}$ contraintes d'inégalité, ces fonctions de pénalisation deviendraient

$$p_3(\mathbf{x}) = \sum_{j=1}^{n_\mathrm{i}} [\max\{0, c_j^\mathrm{i}(\mathbf{x})\}]^2, \tag{10.70}$$

et

$$p_4(\mathbf{x}) = \sum_{i=1}^{n_\mathrm{i}} \max\{0, c_j^\mathrm{i}(\mathbf{x})\}. \tag{10.71}$$

On peut voir une fonction de pénalisation comme un mur autour de $\mathbb{X}$. Plus $\alpha$ est grand dans (10.67) et plus la pente du mur devient raide.

On conduit typiquement une série de minimisations sans contrainte

$$\widehat{\mathbf{x}}^k = \arg\min_{\mathbf{x}} J_{\alpha_k}(\mathbf{x}), \quad k = 1, 2, \cdots, \tag{10.72}$$

avec des valeurs positives *croissantes* de $\alpha_k$ pour s'approcher de $\partial\mathbb{X}$ de l'extérieur. L'estimée finale du minimiseur sous contraintes obtenue lors d'une minimisation sert de point initial pour la suivante.

*Remarque 10.7.* Le compteur d'itérations externes $k$ dans (10.72) ne doit pas être confondu avec le compteur d'itérations internes de l'algorithme en charge de chacune des minimisations. □

Sous des conditions techniques raisonnables [95, 259], il existe un $\bar{\alpha}$ *fini* tel que l'utilisation de $p_2(\cdot)$ ou de $p_4(\cdot)$ conduise à une solution $\widehat{\mathbf{x}}^k \in \mathbb{X}$ dès que $\alpha_k > \bar{\alpha}$. On parle alors de *pénalisation exacte* [13]. Avec $p_1(\cdot)$ ou $p_3(\cdot)$, $\alpha_k$ doit tendre vers l'infini pour qu'on obtienne le même résultat, ce qui pose d'évidents problèmes numériques. Le prix à payer pour une pénalisation exacte est que $p_2(\cdot)$ et $p_4(\cdot)$ ne sont pas différentiables, ce qui complique la minimisation de $J_{\alpha_k}(\mathbf{x})$.

*Exemple 10.5.* Considérons la minimisation de $J(x) = x^2$ sous la contrainte $x \geqslant 1$. Avec la fonction de pénalisation $p_3(\cdot)$, on est conduit à résoudre le problème de minimisation *sans contrainte*

$$\widehat{x} = \arg\min_x J_\alpha(x), \tag{10.73}$$

où

$$J_\alpha(x) = x^2 + \alpha[\max\{0,(1-x)\}]^2, \tag{10.74}$$

pour une valeur positive fixée de $\alpha$.

Puisque $x$ doit être positif pour que la contrainte soit satisfaite, il suffit de considérer deux cas. Si $x > 1$, alors $\max\{0,(1-x)\} = 0$ et $J_\alpha(x) = x^2$, de sorte que le minimiseur $\widehat{x}$ de $J_\alpha(x)$ est $\widehat{x} = 0$, ce qui est impossible. Si $0 \leqslant x \leqslant 1$, alors $\max\{0,(1-x)\} = 1-x$, de sorte que

$$J_\alpha(x) = x^2 + \alpha(1-x)^2. \tag{10.75}$$

La condition nécessaire d'optimalité au premier ordre (9.6) implique alors que

$$\widehat{x} = \frac{\alpha}{1+\alpha} < 1. \tag{10.76}$$

La contrainte est donc toujours violée, puisque $\alpha$ ne peut pas, en pratique, tendre vers l'infini.

Quand $p_3(\cdot)$ est remplacé par $p_4(\cdot)$, $J_\alpha(x)$ n'est plus différentiable, mais la figure 10.3 montre que le minimiseur sans contrainte de $J_\alpha$ satisfait la contrainte quand $\alpha = 3$. (Il ne le fait pas quand $\alpha = 1$.) □

### *10.4.2 Fonctions barrières*

La plus célèbre des fonctions barrières pour $n_\mathrm{i}$ contraintes d'inégalité est la *barrière logarithmique*

$$p_5(\mathbf{x}) = -\sum_{j=1}^{n_\mathrm{i}} \ln[-c_j^\mathrm{i}(\mathbf{x})]. \tag{10.77}$$

Celle-ci joue un rôle essentiel dans les méthodes à points intérieurs, voir par exemple la fonction `fmincon` de la *MATLAB Optimization Toolbox*.

Un autre exemple de fonction barrière est

$$p_6(\mathbf{x}) = -\sum_{j=1}^{n_\mathrm{i}} \frac{1}{c_j^\mathrm{i}(\mathbf{x})}. \tag{10.78}$$

Puisque $c_j^\mathrm{i}(\mathbf{x}) < 0$ à l'intérieur de $\mathbb{X}$, ces fonctions barrières sont bien définies.

On conduit typiquement une série de minimisations sans contrainte (10.72), avec des valeurs positives *décroissantes* de $\alpha_k$ afin de s'approcher de $\partial\mathbb{X}$ depuis l'intérieur. L'estimée du minimiseur sous contraintes obtenue lors d'une minimisation sert à nouveau de point initial pour la suivante. Cette approche fournit des solutions sous-optimales mais admissibles.

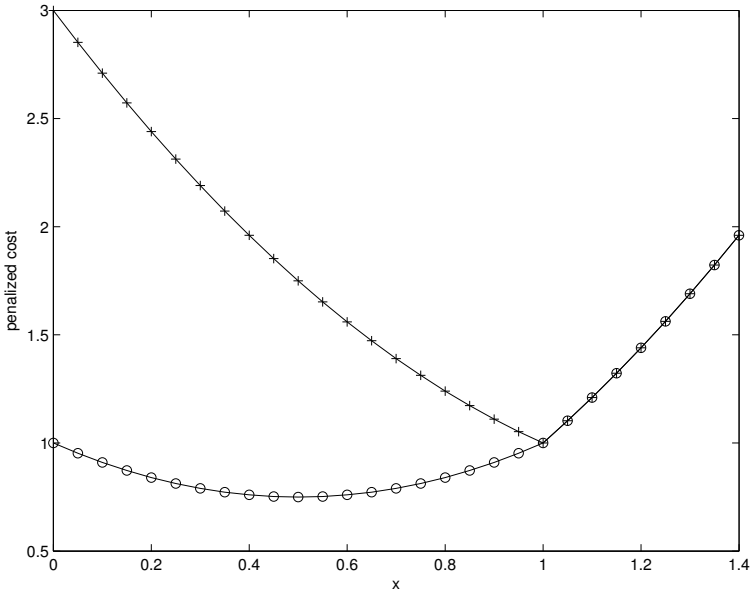


**Fig. 10.3** La fonction de pénalisation $p_4(\cdot)$ est utilisée pour mettre en œuvre un coût quadratique pénalisé par une fonction $l_1$ pour la contrainte $x \geqslant 1$ ; les cercles sont pour $\alpha = 1$, et les croix pour $\alpha = 3$

*Remarque 10.8.* Les modèles à base de connaissances ont souvent un domaine de validité limité. L'évaluation de fonctions de coût fondées sur de tels modèles peut donc n'avoir aucun sens à moins que certaines contraintes d'inégalité soient satisfaites. Les fonctions barrières sont alors plus utiles pour traiter ces contraintes que les fonctions de pénalisation. □

### *10.4.3 Lagrangiens augmentés*

Pour éviter des problèmes numériques résultants de l'emploi de valeurs de $\alpha$ trop grandes tout en gardant des fonctions de pénalisation différentiables, on peut ajouter la fonction de pénalisation au lagrangien

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = J(\mathbf{x}) + \boldsymbol{\lambda}^{\mathrm{T}} \mathbf{c}^{\mathrm{e}}(\mathbf{x}) + \boldsymbol{\mu}^{\mathrm{T}} \mathbf{c}^{\mathrm{i}}(\mathbf{x}) \tag{10.79}$$

pour obtenir le *lagrangien augmenté*

$$L_\alpha(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) + \alpha p(\mathbf{x}). \tag{10.80}$$

La fonction de pénalisation peut être

$$p(\mathbf{x}) = \sum_{i=1}^{n_\mathrm{e}} [c_i^\mathrm{e}(\mathbf{x})]^2 + \sum_{j=1}^{n_\mathrm{i}} [\max\{0, c_j^\mathrm{i}(\mathbf{x})]^2. \tag{10.81}$$

Plusieurs stratégies sont disponibles pour le réglage de $\mathbf{x}$, $\boldsymbol{\lambda}$ et $\boldsymbol{\mu}$ pour un $\alpha > 0$ donné. L'une d'entre elles [185] alterne

1. une *minimisation* du lagrangien augmenté par rapport à $\mathbf{x}$ pour $\boldsymbol{\lambda}$ et $\boldsymbol{\mu}$ fixés, par une méthode d'optimisation sans contrainte,
2. une itération d'un algorithme de gradient avec son coefficient $\lambda$ pris égal à $\alpha$ pour *maximiser* le lagrangien augmenté par rapport à $\boldsymbol{\lambda}$ et $\boldsymbol{\mu}$ pour $\mathbf{x}$ fixé,

$$\begin{aligned} \begin{bmatrix} \boldsymbol{\lambda}^{k+1} \\ \boldsymbol{\mu}^{k+1} \end{bmatrix} &= \begin{bmatrix} \boldsymbol{\lambda}^{k} \\ \boldsymbol{\mu}^{k} \end{bmatrix} + \alpha \begin{bmatrix} \frac{\partial L_\alpha}{\partial \boldsymbol{\lambda}}(\mathbf{x}^k, \boldsymbol{\lambda}^k, \boldsymbol{\mu}^k) \\ \frac{\partial L_\alpha}{\partial \boldsymbol{\mu}}(\mathbf{x}^k, \boldsymbol{\lambda}^k, \boldsymbol{\mu}^k) \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{\lambda}^{k} \\ \boldsymbol{\mu}^{k} \end{bmatrix} + \alpha \begin{bmatrix} \mathbf{c}^\mathrm{e}(\mathbf{x}^k) \\ \mathbf{c}^\mathrm{i}(\mathbf{x}^k) \end{bmatrix}. \end{aligned} \tag{10.82}$$

Il n'est plus nécessaire de faire tendre $\alpha$ vers l'infini pour imposer une satisfaction exacte des contraintes. Les contraintes d'inégalité demandent un soin particulier, car seules celles qui sont actives doivent être prises en considération. Ceci correspond aux *stratégies à ensemble actif* (*active-set strategies*) [176].

## 10.5 Programmation quadratique séquentielle

En programmation quadratique séquentielle, ou *sequential quadratic programming* (SQP) [22, 20, 21], le lagrangien est approximé par son développement de Taylor au second ordre par rapport à $\mathbf{x}$ en $(\mathbf{x}^k, \boldsymbol{\lambda}^k, \boldsymbol{\mu}^k)$, tandis que les contraintes sont approximées par leurs développements de Taylor au premier ordre en $\mathbf{x}^k$. Les équations KKT du problème d'optimisation quadratique résultant sont alors résolues pour obtenir $(\mathbf{x}^{k+1}, \boldsymbol{\lambda}^{k+1}, \boldsymbol{\mu}^{k+1})$, ce qui peut se faire efficacement. Des idées similaires à celles utilisées dans les méthodes de quasi-Newton peuvent être employées pour calculer des approximations de la hessienne du laplacien sur la base des valeurs successives de son gradient.

La mise en œuvre de SQP, l'une des approches les plus puissantes pour l'optimisation non linéaire sous contraintes, est une affaire complexe, qu'il est préférable de laisser au spécialiste puisque SQP est disponible dans nombre de progiciels. En MATLAB, on peut utiliser `sqp`, l'un des algorithmes implémentés dans la fonction `fmincon` de l'*Optimization Toolbox*, qui est inspiré de [173].

## 10.6 Programmation linéaire

Un programme (ou problème d'optimisation) est linéaire si la fonction d'objectif et les contraintes sont linéaires (ou affines) en les variables de décision. Bien que ce cas soit très particulier, il est extrêmement commun en pratique (en économie ou en logistique, par exemple), tout comme les moindres carrés linéaires dans le contexte de l'optimisation sans contrainte. Comme des algorithmes dédiés très puissants sont disponibles, il est important de reconnaître les programmes linéaires quand on les rencontre. Une introduction pédagogique à la programmation linéaire est [153].

*Exemple 10.6. Maximisation de valeur*

Cet exemple jouet n'a aucune prétention à la pertinence économique. Une compagnie fabrique $x_1$ tonnes d'un produit $P_1$ et $x_2$ tonnes d'un produit $P_2$. Une masse donnée de $P_1$ vaut le double de la valeur d'une même masse de $P_2$ et occupe un volume triple. Comment cette compagnie devrait-elle choisir $x_1$ et $x_2$ pour maximiser la valeur du stock dans son entrepôt, compte-tenu du fait que cet entrepôt est juste assez grand pour recevoir une tonne de $P_2$ (si aucun espace n'est pris par $P_1$) et qu'il est impossible de produire une plus grande masse de $P_1$ que de $P_2$ ?

Cette question se traduit par le programme linéaire

$$\text{Maximiser } U(\mathbf{x}) = 2x_1 + x_2 \tag{10.83}$$

sous les contraintes

$$\begin{aligned} x_1 &\geqslant 0, && (10.84)\\ x_2 &\geqslant 0, && (10.85)\\ 3x_1 + x_2 &\leqslant 1, && (10.86)\\ x_1 &\leqslant x_2. && (10.87) \end{aligned}$$

Ce programme est suffisamment simple pour qu'on puisse le résoudre graphiquement. Chacune des contraintes d'inégalité (10.84)–(10.87) coupe le plan $(x_1, x_2)$ en deux demi-plans, dont l'un doit être éliminé. L'intersection des demi-plans restants est le domaine admissible $\mathbb{X}$, qui est un polytope convexe (figure 10.4).

Puisque le gradient de la fonction d'utilité

$$\frac{\partial U}{\partial \mathbf{x}} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{10.88}$$

ne s'annule jamais, il n'y a pas de point stationnaire, et tout maximiseur de $U(\cdot)$ doit appartenir à $\partial\mathbb{X}$. Tous les $\mathbf{x}$ associés à la même valeur $a$ de la fonction d'utilité $U(\mathbf{x})$ sont sur la droite

$$2x_1 + x_2 = a. \tag{10.89}$$

Le maximiseur sous contraintes de la fonction d'utilité est donc le sommet de $\mathbb{X}$ situé sur la droite (10.89) associée à la plus grande valeur de $a$, c'est à dire

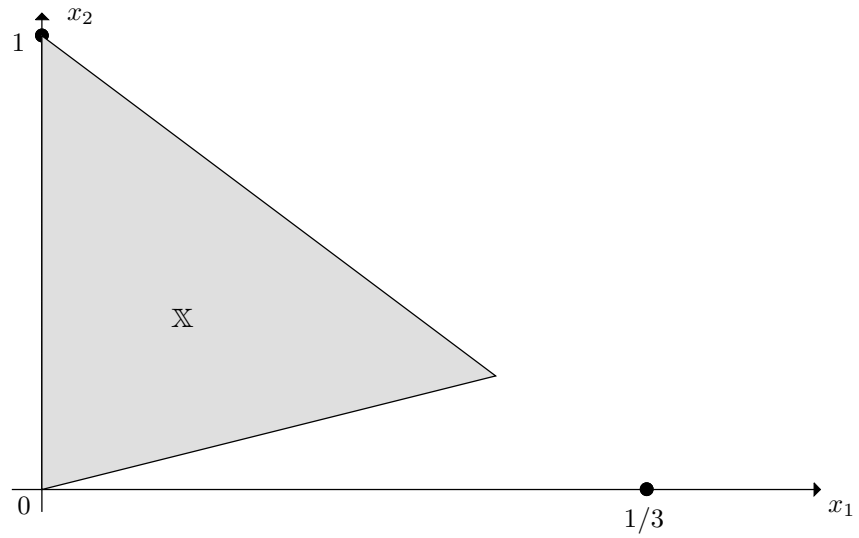


**Fig. 10.4** Domaine admissible $\mathbb{X}$ pour l'exemple 10.6

$$\widehat{\mathbf{x}} = [0 \quad 1]^{\mathrm{T}}. \tag{10.90}$$

La compagnie ne doit donc produire que du $P_2$. L'utilité résultante est $U(\widehat{\mathbf{x}}) = 1$. □

*Exemple 10.7. Estimation $l_p$ pour $p = 1$ ou $p = \infty$*

L'estimateur des moindres carrés (ou $l_2$) linéaire de la section 9.2 est

$$\widehat{\mathbf{x}}_{\mathrm{LS}} = \arg\min_{\mathbf{x}} \|\mathbf{e}(\mathbf{x})\|_2^2, \tag{10.91}$$

où l'erreur $\mathbf{e}(\mathbf{x})$ est le vecteur de dimension $N$ des résidus entre les données et les sorties du modèle

$$\mathbf{e}(\mathbf{x}) = \mathbf{y} - \mathbf{F}\mathbf{x}. \tag{10.92}$$

Quand certaines données $y_i$ sont complètement anormales, par exemple à cause d'une panne de capteur, ces *données aberrantes* (ou *outliers*) peuvent tant affecter la valeur numérique de $\widehat{\mathbf{x}}_{\mathrm{LS}}$ qu'elle en devient inutilisable. Les *estimateurs robustes* sont conçus pour être moins sensibles à des données aberrantes. L'un d'eux est l'*estimateur des moindres modules* (ou $l_1$)

$$\widehat{\mathbf{x}}_{\mathrm{LM}} = \arg\min_{\mathbf{x}} \|\mathbf{e}(\mathbf{x})\|_1. \tag{10.93}$$

Comme les composantes du vecteur d'erreur ne sont pas élevées au carré comme dans l'estimateur $l_2$, l'impact de quelques données aberrantes est bien moindre. L'estimateur des moindres modules peut être calculé [82, 128] comme

$$\widehat{\mathbf{x}}_{\mathrm{LM}} = \arg\min_{\mathbf{x}} \sum_{i=1}^{N} (u_i + v_i) \tag{10.94}$$

sous les contraintes

$$\begin{aligned} u_i - v_i &= y_i - \mathbf{f}_i^{\mathrm{T}}\mathbf{x}, \\ u_i &\geqslant 0, \\ v_i &\geqslant 0, \end{aligned} \tag{10.95}$$

pour $i = 1, \cdots, N$, où $\mathbf{f}_i^{\mathrm{T}}$ est la $i$-ème ligne de $\mathbf{F}$. Le calcul de $\widehat{\mathbf{x}}_{\mathrm{LM}}$ a ainsi été transformé en un programme linéaire, où les $n + 2N$ variables de décision sont les $n$ éléments de $\mathbf{x}$, et $u_i$ et $v_i$ $(i = 1, \cdots, N)$. On pourrait aussi calculer

$$\widehat{\mathbf{x}}_{\mathrm{LM}} = \arg\min_{\mathbf{x}} \sum_{i=1}^{N} \mathbf{1}^{\mathrm{T}}\mathbf{s}, \tag{10.96}$$

avec $\mathbf{1}$ un vecteur colonne dont tous les éléments sont égaux à un, sous les contraintes

$$\mathbf{y} - \mathbf{F}\mathbf{x} \leqslant \mathbf{s}, \tag{10.97}$$

$$-(\mathbf{y} - \mathbf{F}\mathbf{x}) \leqslant \mathbf{s}, \tag{10.98}$$

où les inégalités sont à interpréter composante par composante, comme d'habitude. Calculer $\widehat{\mathbf{x}}_{\mathrm{LM}}$ reste un programme linéaire, mais avec seulement $n + N$ variables de décision, à savoir les éléments de $\mathbf{x}$ et $\mathbf{s}$.

De façon similaire [82], l'évaluation d'un *estimateur minimax* (ou $l_\infty$)

$$\widehat{\mathbf{x}}_{\mathrm{MM}} = \arg\min_{\mathbf{x}} \|\mathbf{e}(\mathbf{x})\|_\infty \tag{10.99}$$

se traduit par le programme linéaire

$$\widehat{\mathbf{x}}_{\mathrm{MM}} = \arg\min_{\mathbf{x}} d_\infty, \tag{10.100}$$

sous les contraintes

$$\mathbf{y} - \mathbf{F}\mathbf{x} \leqslant \mathbf{1}d_\infty, \tag{10.101}$$

$$-(\mathbf{y} - \mathbf{F}\mathbf{x}) \leqslant \mathbf{1}d_\infty, \tag{10.102}$$

avec $n + 1$ variables de décision, à savoir les éléments de $\mathbf{x}$ et $d_\infty$.

L'estimateur minimax est encore moins robuste vis à vis de données aberrantes que l'estimateur $l_2$, puisqu'il minimise la plus grande déviation en valeur absolue entre une donnée et la sortie du modèle correspondante, sur toutes les données. L'optimisation minimax est principalement utilisée pour le réglage de variables de conception de façon à se protéger contre les effets de variables d'environnement incertaines (voir la section 9.4.1.2). □

*Remarque 10.9.* Dans l'exemple 10.7, des problèmes d'optimisation sans contrainte sont traités via la programmation linéaire comme des problèmes d'optimisation sous contraintes. □

Les problèmes réels peuvent impliquer de très nombreuses variables de décision (on peut maintenant traiter des problèmes avec des millions de variables), de sorte qu'il est impensable d'évaluer le coût à chaque sommet de $\mathbb{X}$ et qu'il faut une méthode d'exploration systématique.

La *méthode du simplexe de Dantzig*, à ne pas confondre avec celle de Nelder et Mead de la section 9.3.5, explore $\partial\mathbb{X}$ en se déplaçant le long des arrêtes de $\mathbb{X}$ d'un sommet vers le suivant tout en améliorant la valeur de la fonction d'objectif. Elle est considérée en premier. Les *méthodes à points intérieurs*, parfois plus efficaces, seront présentées en section 10.6.3.

### *10.6.1 Forme standard*

Pour éviter d'avoir à multiplier les sous-cas suivant que la fonction d'objectif est à minimiser ou à maximiser et suivant qu'il y a des contraintes d'égalité ou d'inégalité ou des deux types, il est commode de mettre le programme sous la *forme standard* suivante :

— $J(\cdot)$ est une fonction de coût, à minimiser,
— toutes les variables de décision $x_i$ sont non négatives, c'est à dire que $x_i \geqslant 0$,
— toutes les contraintes sont des contraintes d'égalité.

Au moins conceptuellement, il est facile de se ramener à cette forme standard. Quand il faut maximiser une fonction d'utilité $U(\cdot)$, il suffit de poser $J(\mathbf{x}) = -U(\mathbf{x})$ pour se ramener à une minimisation. Quand le signe d'une variable de décision $x_i$ est inconnu, on peut la remplacer par la différence entre deux variables de décision non négatives, en posant

$$x_i = x_i^+ - x_i^-, \quad \text{avec} \quad x_i^+ \geqslant 0 \quad \text{et} \quad x_i^- \geqslant 0. \tag{10.103}$$

Toute contrainte d'inégalité peut être transformée en une contrainte d'égalité via l'introduction d'une variable de décision (non négative) supplémentaire. Par exemple

$$3x_1 + x_2 \leqslant 1 \tag{10.104}$$

se traduit par

$$3x_1 + x_2 + x_3 = 1, \tag{10.105}$$
$$x_3 \geqslant 0, \tag{10.106}$$

où $x_3$ est une *variable d'écart*, et

$$3x_1 + x_2 \geqslant 1 \tag{10.107}$$

se traduit par

$$3x_1 + x_2 - x_3 = 1, \tag{10.108}$$
$$x_3 \geqslant 0, \tag{10.109}$$

où $x_3$ est une *variable de surplus*.

Le problème standard peut donc s'écrire, après l'introduction éventuelle de composantes supplémentaires dans le vecteur de décision $\mathbf{x}$, comme celui de trouver

$$\widehat{\mathbf{x}} = \arg\min_{\mathbf{x}} J(\mathbf{x}), \tag{10.110}$$

où la fonction de coût dans (10.110) est une combinaison linéaire des variables de décision :

$$J(\mathbf{x}) = \mathbf{c}^{\mathrm{T}}\mathbf{x}, \tag{10.111}$$

sous les contraintes

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{10.112}$$
$$\mathbf{x} \geqslant \mathbf{0}. \tag{10.113}$$

L'équation (10.112) exprime $m$ contraintes d'égalité affines entre les $n$ variables de décision

$$\sum_{k=1}^{n} a_{j,k}x_k = b_k, \quad j = 1, \cdots, m. \tag{10.114}$$

La matrice $\mathbf{A}$ a donc $m$ lignes (autant qu'il y a de contraintes) et $n$ colonnes (autant qu'il y a de variables).

Insistons, une fois de plus, sur le fait que le gradient du coût n'est jamais nul, puisque

$$\frac{\partial J}{\partial \mathbf{x}} = \mathbf{c}. \tag{10.115}$$

Minimiser un coût linéaire en l'absence de toute contrainte n'aurait donc aucun sens, car on pourrait faire tendre $J(\mathbf{x})$ vers $-\infty$ en faisant tendre $||\mathbf{x}||$ vers l'infini dans la direction $-\mathbf{c}$. La situation est donc très différente de celle rencontrée avec les fonctions de coût quadratiques.

### *10.6.2 Principe de la méthode du simplexe de Dantzig*

Nous supposons que

- les contraintes sont compatibles ($\mathbb{X}$ n'est pas vide),
- rang $\mathbf{A} = m$ (aucune contrainte ne peut être éliminée comme redondante),
- le nombre $n$ des variables est plus grand que le nombre $m$ des contraintes d'égalité (il reste un choix à faire),
- $\mathbb{X}$ tel que défini par (10.112) et (10.113) est borné (c'est un polytope convexe).

Ces hypothèses impliquent que le minimum global du coût est atteint sur un sommet de $\mathbb{X}$. Il peut y avoir plusieurs minimiseurs globaux, mais l'algorithme du simplexe se borne à rechercher l'un d'entre eux.

La proposition qui suit joue un rôle clé [30].

**Proposition 10.2.** *Si* $\mathbf{x} \in \mathbb{R}^n$ *est un sommet d'un polytope convexe* $\mathbb{X}$ *défini par* $m$ *contraintes d'égalité linéairement indépendantes* $\mathbf{A}\mathbf{x} = \mathbf{b}$*, alors* $\mathbf{x}$ *a* au moins $n-m$ *éléments nuls.* □

*Preuve.* $\mathbf{A}$ est $m \times n$. Si $\mathbf{a}_i \in \mathbb{R}^m$ est la $i$-ème colonne de $\mathbf{A}$, alors

$$\mathbf{A}\mathbf{x} = \mathbf{b} \Longleftrightarrow \sum_{i=1}^{n} \mathbf{a}_i x_i = \mathbf{b}. \tag{10.116}$$

Indexons les colonnes de $\mathbf{A}$ de telle façon que les éléments non nuls de $\mathbf{x}$ soient indexés de 1 à $r$. Alors

$$\sum_{i=1}^{r} \mathbf{a}_i x_i = \mathbf{b}. \tag{10.117}$$

Prouvons que les $r$ premiers vecteurs $\mathbf{a}_i$ sont linéairement indépendants. La preuve est par contradiction. S'ils étaient linéairement dépendants, alors on pourrait trouver un vecteur $\boldsymbol{\alpha} \in \mathbb{R}^n$ non nul tel que $\alpha_i = 0$ pour tout $i > r$ et que

$$\sum_{i=1}^{r} \mathbf{a}_i (x_i + \varepsilon \alpha_i) = \mathbf{b} \Longleftrightarrow \mathbf{A}(\mathbf{x} + \varepsilon \boldsymbol{\alpha}) = \mathbf{b}, \tag{10.118}$$

avec $\varepsilon = \pm\theta$. Soit $\theta$ un réel suffisamment petit pour assurer que $\mathbf{x} + \varepsilon\boldsymbol{\alpha} \geqslant \mathbf{0}$. On aurait alors

$$\mathbf{x}^1 = \mathbf{x} + \theta\boldsymbol{\alpha} \in \mathbb{X} \quad \text{et} \quad \mathbf{x}^2 = \mathbf{x} - \theta\boldsymbol{\alpha} \in \mathbb{X}, \tag{10.119}$$

de sorte que

$$\mathbf{x} = \frac{\mathbf{x}^1 + \mathbf{x}^2}{2} \tag{10.120}$$

ne pourrait pas être un sommet, puisqu'il serait strictement à l'intérieur d'une arrête. Les $r$ premiers vecteurs $\mathbf{a}_i$ sont donc linéairement indépendants. Maintenant, puisque $\mathbf{a}_i \in \mathbb{R}^m$, il y a au plus $m$ $\mathbf{a}_i$ linéairement indépendants, de sorte que $r \leqslant m$ et que $\mathbf{x} \in \mathbb{R}^n$ a au moins $n-m$ éléments nuls. □

N'importe quel vecteur $\mathbf{x}_\mathrm{b}$ de $\mathbb{X}$ ayant au moins $n-m$ éléments nuls est appelé *solution admissible de base*. Nous supposerons dans la description de la méthode du simplexe qu'un tel $\mathbf{x}_\mathrm{b}$ a déjà été trouvé.

*Remarque 10.10.* Quand on ne dispose pas d'une solution admissible de base, on peut en générer une (au prix d'une augmentation de la dimension de l'espace de recherche) par la procédure suivante [30] :

1. ajouter une *variable artificielle* différente au côté gauche de chaque contrainte qui ne contient pas de variable d'écart (même si elle contient une variable de surplus),

2. calculer les valeurs des $m$ variables artificielles et d'écart en résolvant l'ensemble de contraintes résultantes avec toutes les variables initiales et de surplus mises à zéro. Ceci est immédiat : la variable artificielle ou d'écart introduite dans la $j$-ème contrainte de (10.112) prend la valeur $b_j$ (qu'on peut toujours s'arranger pour choisir positive). Comme il y a maintenant au plus $m$ variables non nulles, une solution admissible de base a ainsi été obtenue, *mais pour un autre problème*.

En introduisant des variables artificielles, nous avons en effet changé le problème traité, à moins que toutes ces variables ne prennent la valeur zéro. C'est pourquoi la fonction de coût est modifiée en ajoutant à la fonction de coût précédente chacune des variables artificielles multipliée par un grand coefficient positif. A moins que $\mathbb{X}$ ne soit vide, toutes les variables artificielles devraient alors être conduites vers zéro par l'algorithme du simplexe, et la solution finalement fournie devrait correspondre à celle du problème initial. Cette procédure peut aussi être utilisée pour vérifier que $\mathbb{X}$ n'est pas vide. Supposons, par exemple, qu'on doive minimiser $J_1(x_1,x_2)$ sous les contraintes

$$\begin{aligned} x_1 - 2x_2 &= 0, && (10.121)\\ 3x_1 + 4x_2 &\geqslant 5, && (10.122)\\ 6x_1 + 7x_2 &\leqslant 8. && (10.123) \end{aligned}$$

Pour un problème aussi simple, il est trivial de montrer qu'il n'y a pas de solution en $x_1$ et $x_2$, mais supposons que ceci nous ait échappé. Pour mettre le problème sous forme standard, introduisons la variable de surplus $x_3$ dans (10.122) et la variable d'écart $x_4$ dans (10.123), pour obtenir

$$\begin{aligned} x_1 - 2x_2 &= 0, && (10.124)\\ 3x_1 + 4x_2 - x_3 &= 5, && (10.125)\\ 6x_1 + 7x_2 + x_4 &= 8. && (10.126) \end{aligned}$$

Ajoutons les variables artificielles $x_5$ à (10.124) et $x_6$ à (10.125), pour obtenir

$$\begin{aligned} x_1 - 2x_2 + x_5 &= 0, && (10.127)\\ 3x_1 + 4x_2 - x_3 + x_6 &= 5, && (10.128)\\ 6x_1 + 7x_2 + x_4 &= 8. && (10.129) \end{aligned}$$

Résolvons (10.127)–(10.129) en les variables artificielles et d'écart, avec toutes les autres variables mises à zéro, pour obtenir

$$\begin{aligned} x_5 &= 0, && (10.130)\\ x_6 &= 5, && (10.131)\\ x_4 &= 8. && (10.132) \end{aligned}$$

Pour le problème *modifié*, $\mathbf{x} = (0,0,0,8,0,5)^{\mathrm{T}}$ est une solution admissible de base, puisque quatre de ses six éléments sont nuls alors que $n-m=3$. Le fait de remplacer la fonction de coût initiale $J_1(x_1,x_2)$ par

$$J_2(\mathbf{x}) = J_1(x_1,x_2) + Mx_5 + Mx_6 \tag{10.133}$$

(avec $M$ un grand coefficient positif) ne convaincra cependant pas l'algorithme du simplexe de se débarrasser des variables artificielles, puisque nous savons que c'est mission impossible. □

Pourvu que $\mathbb{X}$ ne soit pas vide, l'une des solutions admissibles de base est un minimiseur global de la fonction de coût, et l'algorithme du simplexe se déplace d'une solution admissible de base vers la suivante tout en faisant décroître le coût.

Parmi les éléments nuls de $\mathbf{x}_{\mathrm{b}}$, $n-m$ éléments sont sélectionnés et appelés *hors-base*. Les $m$ éléments restants sont appelés *variables de base*. Les variables de base incluent donc tous les éléments non nuls de $\mathbf{x}_{\mathrm{b}}$.

L'équation (10.112) permet alors d'exprimer les variables de base et le coût $J(\mathbf{x})$ en fonction des variables hors base. Cette description sera utilisée pour décider quelle variable hors base doit devenir de base et quelle variable de base doit quitter la base pour lui laisser sa place. Pour simplifier la présentation de la méthode, nous utilisons l'exemple 10.6.

Considérons donc à nouveau le problème défini par (10.83)–(10.87), mis sous forme standard. La fonction de coût est

$$J(\mathbf{x}) = -2x_1 - x_2, \tag{10.134}$$

avec $x_1 \geqslant 0$ et $x_2 \geqslant 0$, et les contraintes d'inégalité (10.86) et (10.87) sont transformées en des contraintes d'égalité en introduisant les variables d'écart $x_3$ et $x_4$, de sorte que

$$\begin{aligned} 3x_1 + x_2 + x_3 &= 1, && (10.135)\\ x_3 &\geqslant 0, && (10.136)\\ x_1 - x_2 + x_4 &= 0, && (10.137)\\ x_4 &\geqslant 0. && (10.138) \end{aligned}$$

Le nombre des variables de décision (non négatives) est maintenant $n=4$ et le nombre des contraintes d'égalité est $m=2$. Toute solution admissible de base $\mathbf{x} \in \mathbb{R}^4$ a donc au moins deux éléments nuls.

Nous commençons par rechercher une solution admissible de base avec $x_1$ et $x_2$ de base, et $x_3$ et $x_4$ hors base. Les contraintes (10.135) et (10.137) se traduisent par

$$\begin{bmatrix} 3 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1-x_3 \\ -x_4 \end{bmatrix}. \tag{10.139}$$

Résolvons (10.139) en $x_1$ et $x_2$, pour obtenir

$$x_1 = \frac{1}{4} - \frac{1}{4}x_3 - \frac{1}{4}x_4, \tag{10.140}$$

et

$$x_2 = \frac{1}{4} - \frac{1}{4}x_3 + \frac{3}{4}x_4. \tag{10.141}$$

Il est trivial de vérifier que le vecteur $\mathbf{x}$ obtenu en mettant $x_3$ et $x_4$ à zéro et en choisissant $x_1$ et $x_2$ de façon à satisfaire (10.140) et (10.141), c'est à dire en posant

$$\mathbf{x} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 & 0 \end{bmatrix}^{\mathrm{T}}, \tag{10.142}$$

satisfait toutes les contraintes tout en ayant un nombre convenable d'éléments nuls. C'est donc une solution admissible de base.

Le coût peut lui aussi s'exprimer en fonction des variables hors base, puisque (10.134), (10.140) et (10.141) impliquent que

$$J(\mathbf{x}) = -\frac{3}{4} + \frac{3}{4}x_3 - \frac{1}{4}x_4. \tag{10.143}$$

La situation est résumée par le tableau 10.1.

**Tableau 10.1** Situation initiale dans l'exemple 10.6

| | coefficient constant | coefficient de $x_3$ | coefficient de $x_4$ |
|---|---|---|---|
| $J$ | $-3/4$ | $3/4$ | $-1/4$ |
| $x_1$ | $1/4$ | $-1/4$ | $-1/4$ |
| $x_2$ | $1/4$ | $-1/4$ | $3/4$ |

Les deux dernières lignes de la première colonne du tableau 10.1 listent les variables de base, tandis que les deux dernières colonnes de sa première ligne listent les variables hors base. L'algorithme du simplexe modifie ce tableau itérativement, en échangeant des variables de base et des variables hors base. La modification à effectuer lors d'une itération est décidée en trois étapes.

La *première étape* sélectionne, parmi les variables hors base, une variable telle que l'élément associé dans la ligne du coût est

- négatif (pour permettre au coût de décroître),
- de plus grande valeur absolue (pour que cette décroissance soit rapide).

Dans notre exemple, seule $x_4$ est associée à un coefficient négatif, de sorte qu'elle est sélectionnée pour devenir une variable de base. Quand il y a plusieurs variables hors base également prometteuses (coefficient négatif de valeur absolue maximale), ce qui est rare, il suffit d'en tirer une au sort. Si aucune variable hors base n'a un coefficient négatif, alors la solution admissible de base est globalement optimale et l'algorithme stoppe.

La *deuxième étape* augmente la variable hors base $x_i$ sélectionnée durant la première étape pour rejoindre la base (ici, $i = 4$), jusqu'à ce que l'une des variables de base devienne nulle et quitte ainsi la base pour faire de la place pour

$x_i$. Pour découvrir laquelle des variables de base précédentes se fera expulser, il faut considérer les signes des coefficients situés aux intersections entre la colonne associée à la nouvelle variable de base $x_i$ et les lignes associées aux variables de base précédentes. Quand ces coefficients sont positifs, faire croître $x_i$ fait aussi croître les variables correspondantes, qui restent donc dans la base. La variable qui va quitter la base a donc un coefficient négatif. La variable de base précédente à coefficient négatif qui atteint zéro la première quand $x_i$ augmente est celle qui quitte la base. Dans notre exemple, il y a un seul coefficient négatif, égal à $-1/4$ et associé à $x_1$. La variable $x_1$ devient nulle et quitte la base quand la nouvelle variable de base $x_4$ atteint 1.

La *troisième étape* met le tableau à jour. Dans notre exemple, les variables de base sont maintenant $x_2$ et $x_4$, et les variables hors base $x_1$ et $x_3$. Il faut donc exprimer $x_2$, $x_4$ et $J$ en fonction de $x_1$ et $x_3$. De (10.135) et (10.137) nous obtenons

$$x_2 = 1 - 3x_1 - x_3, \tag{10.144}$$
$$-x_2 + x_4 = -x_1, \tag{10.145}$$

ou de façon équivalente

$$x_2 = 1 - 3x_1 - x_3, \tag{10.146}$$
$$x_4 = 1 - 4x_1 - x_3. \tag{10.147}$$

Quant au coût, (10.134) et (10.146) impliquent que

$$J(\mathbf{x}) = -1 + x_1 + x_3. \tag{10.148}$$

Le tableau 10.1 devient donc le tableau 10.2.

**Tableau 10.2** Situation finale dans l'exemple 10.6

| | coefficient constant | coefficient de $x_1$ | coefficient de $x_3$ |
|---|---|---|---|
| $J$ | $-1$ | 1 | 1 |
| $x_2$ | 1 | $-3$ | $-1$ |
| $x_4$ | 1 | $-4$ | $-1$ |

Toutes les variables hors base ont maintenant des coefficients positifs dans la ligne des coûts. Il n'est donc plus possible d'améliorer la solution admissible de base courante

$$\widehat{\mathbf{x}} = [0 \quad 1 \quad 0 \quad 1]^{\mathrm{T}}, \tag{10.149}$$

qui est donc (globalement) optimale et associée avec le coût minimal

$$J(\widehat{\mathbf{x}}) = -1. \tag{10.150}$$

Ceci correspond à une utilité maximale de 1, cohérente avec les résultats obtenus graphiquement.

### *10.6.3 La révolution des points intérieurs*

Jusqu'en 1984, le simplexe de Dantzig a bénéficié d'un quasi monopole dans le contexte de la programmation linéaire, qu'on pensait fort différente de la programmation non linéaire. Le seul inconvénient de cet algorithme était que sa complexité *dans le pire des cas* ne pouvait pas être bornée par un polynôme en la dimension du problème (on pensait donc que la programmation linéaire était un problème NP). En dépit de ce défaut, la méthode du simplexe traitait des problèmes de grande taille avec entrain.

Un article publié par L. Khachiyan en 1979 [126] fit les gros titres (y compris sur la première page du *New York Times*) en montrant qu'on pouvait apporter une complexité polynomiale à la programmation linéaire en spécialisant une méthode ellipsoïdale déjà connue en programmation non linéaire. C'était une première brèche dans le dogme qui affirmait que la programmation linéaire et la programmation non linéaire n'avaient rien à voir. L'algorithme résultant se révéla cependant insuffisamment efficace en pratique pour défier la suprématie du simplexe de Dantzig. C'était ce que M. Wright appela *une anomalie intrigante et profondément insatisfaisante selon laquelle un algorithme à temps exponentiel était régulièrement et substantiellement plus rapide qu'un algorithme à temps polynomial* [255].

En 1984, N. Karmarkar présenta un autre algorithme à temps polynomial pour la programmation linéaire [120], avec des performances bien supérieurs à celles du simplexe de Dantzig sur certains cas tests. C'était un résultats si sensationnel qu'il fut, lui aussi, remarqué par la presse grand public. La *méthode à points intérieurs* de Karmarkar échappe à la complexité combinatoire qu'impose une exploration des arrêtes de $\mathbb{X}$ en se déplaçant vers un minimiseur de la fonction de coût le long d'un chemin qui reste à l'intérieur de $\mathbb{X}$ et n'atteint jamais sa frontière $\partial\mathbb{X}$, même si l'on sait que tout minimiseur appartient à $\partial\mathbb{X}$.

Après pas mal de controverses, dues pour partie au manque de détails dans [120], il est maintenant reconnu que les méthodes à points intérieurs sont beaucoup plus efficaces que la méthode du simplexe sur certains problèmes. Il n'en demeure pas moins que la méthode du simplexe reste plus efficace sur d'autres et qu'elle est encore très utilisée. L'algorithme de Karmarkar équivaut formellement à une méthode à barrière logarithmique appliquée à la programmation linéaire [72], ce qui confirme qu'il y a quelque chose à gagner à considérer la programmation linéaire comme un cas particulier de la programmation non linéaire.

Les méthodes à points intérieurs s'étendent facilement à l'optimisation convexe, dont la programmation linéaire est un cas particulier (voir la section 10.7.6). La séparation traditionnelle entre programmation linéaire et programmation non linéaire tend ainsi à se voir remplacée par une séparation entre optimisation convexe et optimisation non convexe.

Des méthodes à points intérieurs ont aussi été utilisées pour développer des solveurs d'usage général pour l'optimisation non linéaire sous contraintes dans des problèmes non convexes de grande taille [36].

## 10.7 Optimisation convexe

Minimiser $J(\mathbf{x})$ tout en imposant que $\mathbf{x}$ appartienne à $\mathbb{X}$ est un problème d'optimisation convexe si $\mathbb{X}$ et $J(\cdot)$ sont convexes. [28] et [169] forment une excellente introduction à ce sujet ; voir aussi [108] et [107].

### *10.7.1 Ensembles admissibles convexes*

L'ensemble $\mathbb{X}$ est convexe si, pour toute paire de points $(\mathbf{x}_1, \mathbf{x}_2)$ appartenant à $\mathbb{X}$, le segment qui les joint est inclus dans $\mathbb{X}$ :

$$\forall \lambda \in [0,1], \quad \lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in \mathbb{X}; \tag{10.151}$$

voir la figure 10.5.

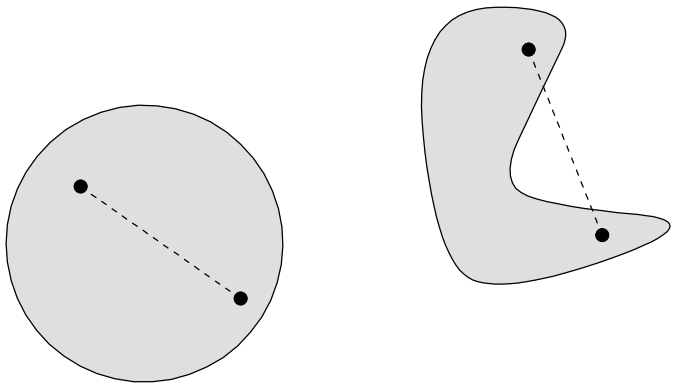

**Fig. 10.5** L'ensemble de gauche est convexe ; celui de droite ne l'est pas car le segment qui joint les deux points n'est pas inclus dans l'ensemble

*Exemple 10.8.* $\mathbb{R}^n$, les hyperplans, les demi-espaces, les ellipsoïdes et les boules unité pour des normes quelconques sont convexes, et l'intersection d'ensembles convexes est convexe. Les ensembles admissibles des programmes linéaires sont donc convexes. □

### *10.7.2 Fonctions de coût convexes*

La fonction $J(\cdot)$ est convexe sur $\mathbb{X}$ si $J(\mathbf{x})$ est défini pour tout $\mathbf{x}$ dans $\mathbb{X}$ et si, pour toute paire $(\mathbf{x}_1, \mathbf{x}_2)$ de points de $\mathbb{X}$, l'inégalité suivante est vérifiée :

$$\forall \lambda \in [0,1] \quad J(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leqslant \lambda J(\mathbf{x}_1) + (1-\lambda) J(\mathbf{x}_2) \; ; \tag{10.152}$$

voir la figure 10.6.

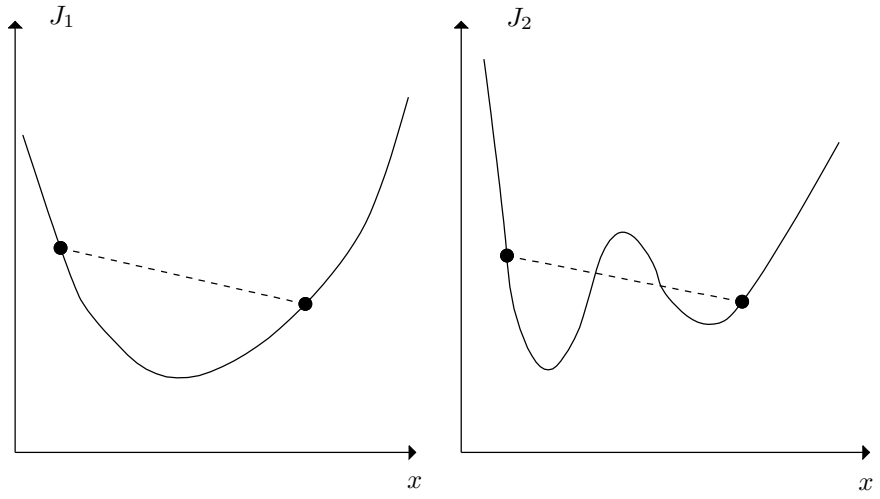


**Fig. 10.6** La fonction de gauche est convexe ; celle de droite ne l'est pas

*Exemple 10.9.* La fonction

$$J(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c \tag{10.153}$$

est convexe, pourvu que $\mathbf{A}$ soit symétrique définie non-négative. □

*Exemple 10.10.* La fonction de coût de la programmation linéaire

$$J(\mathbf{x}) = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{10.154}$$

est convexe. □

*Exemple 10.11.* La fonction

$$J(\mathbf{x}) = \sum_i w_i J_i(\mathbf{x}) \tag{10.155}$$

est convexe si chacune des fonctions $J_i(\mathbf{x})$ est convexe et si chaque poids $w_i$ est positif. □

*Exemple 10.12.* La fonction

$$J(\mathbf{x}) = \max_i J_i(\mathbf{x}) \tag{10.156}$$

est convexe si chacune des fonctions $J_i(\mathbf{x})$ est convexe. □

Si une fonction est convexe sur $\mathbb{X}$, alors elle est continue sur tout ensemble ouvert inclus dans $\mathbb{X}$. Une condition nécessaire et suffisante pour qu'une fonction $J(\cdot)$ différentiable une fois soit convexe est que

$$\forall \mathbf{x}_1 \in \mathbb{X}, \forall \mathbf{x}_2 \in \mathbb{X}, \quad J(\mathbf{x}_2) \geqslant J(\mathbf{x}_1) + \mathbf{g}^{\mathrm{T}}(\mathbf{x}_1)(\mathbf{x}_2 - \mathbf{x}_1), \tag{10.157}$$

où $\mathbf{g}(\cdot)$ est la fonction gradient de $J(\cdot)$. La connaissance de la valeur du gradient d'une fonction convexe en un point quelconque $\mathbf{x}_1$ fournit donc une borne inférieure *globale* pour cette fonction.

### *10.7.3 Conditions théoriques d'optimalité*

La convexité transforme les conditions nécessaires d'optimalité au premier ordre des sections 9.1 et 10.2 en des conditions nécessaires et suffisantes d'optimalité globale. Si $J(\cdot)$ est convexe et différentiable une fois, alors une condition nécessaire *et suffisante* pour que $\widehat{\mathbf{x}}$ soit un minimiseur *global* en l'absence de contrainte est

$$\mathbf{g}(\widehat{\mathbf{x}}) = \mathbf{0}. \tag{10.158}$$

Quand des contraintes définissent un ensemble admissible $\mathbb{X}$, cette condition devient

$$\mathbf{g}^{\mathrm{T}}(\widehat{\mathbf{x}})(\mathbf{x}_2 - \widehat{\mathbf{x}}) \geqslant 0 \quad \forall \mathbf{x}_2 \in \mathbb{X}, \tag{10.159}$$

qui est une conséquence directe de (10.157).

### *10.7.4 Formulation lagrangienne*

Reprenons la formulation lagrangienne de la section 10.2, mais en tirant maintenant avantage de la convexité. Le lagrangien associé à la minimisation de la fonction de coût $J(\mathbf{x})$ sous les contraintes d'inégalité $\mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant \mathbf{0}$ s'écrit

$$L(\mathbf{x}, \boldsymbol{\mu}) = J(\mathbf{x}) + \boldsymbol{\mu}^{\mathrm{T}} \mathbf{c}^{\mathrm{i}}(\mathbf{x}), \tag{10.160}$$

où le vecteur $\boldsymbol{\mu}$ des multiplicateurs de Lagrange (ou de Kuhn et Tucker) est aussi appelé *vecteur dual*. La *fonction duale* $D(\boldsymbol{\mu})$ est l'infimum du lagrangien sur $\mathbf{x}$

$$D(\boldsymbol{\mu}) = \inf_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\mu}). \tag{10.161}$$

Comme $J(\mathbf{x})$ et toutes les contraintes $c^{\mathrm{i}}_j(\mathbf{x})$ sont supposés convexes, $L(\mathbf{x},\boldsymbol{\mu})$ est une fonction convexe de $\mathbf{x}$ tant que $\boldsymbol{\mu} \geqslant \mathbf{0}$, ce qui doit être vrai de toute façon pour les contraintes d'inégalité. L'évaluation de $D(\boldsymbol{\mu})$ est donc un *problème convexe de minimisation sans contrainte*, qu'on peut résoudre avec une méthode locale comme celles de Newton et de quasi-Newton. Si l'infimum de $L(\mathbf{x},\boldsymbol{\mu})$ par rapport à $\mathbf{x}$ est atteint en $\widehat{\mathbf{x}}_\mu$, alors

$$D(\boldsymbol{\mu}) = J(\widehat{\mathbf{x}}_\mu) + \boldsymbol{\mu}^{\mathrm{T}} \mathbf{c}^{\mathrm{i}}(\widehat{\mathbf{x}}_\mu). \tag{10.162}$$

De plus, si $J(\mathbf{x})$ et les contraintes $c^{\mathrm{i}}_j(\mathbf{x})$ sont différentiables, alors $\widehat{\mathbf{x}}_\mu$ satisfait les conditions d'optimalité au premier ordre

$$\frac{\partial J}{\partial \mathbf{x}}(\widehat{\mathbf{x}}_\mu) + \sum_{j=1}^{n_{\mathrm{i}}} \widehat{\mu}_j \frac{\partial c^{\mathrm{i}}_j}{\partial \mathbf{x}}(\widehat{\mathbf{x}}_\mu) = \mathbf{0}. \tag{10.163}$$

Si $\boldsymbol{\mu}$ est *dual admissible*, c'est à dire tel que $\boldsymbol{\mu} \geqslant \mathbf{0}$ et $D(\boldsymbol{\mu}) > -\infty$, alors pour tout $\mathbf{x}$ admissible

$$D(\boldsymbol{\mu}) = \inf_{\mathbf{x}} L(\mathbf{x},\boldsymbol{\mu}) \leqslant L(\mathbf{x},\boldsymbol{\mu}) = J(\mathbf{x}) + \boldsymbol{\mu}^{\mathrm{T}} \mathbf{c}^{\mathrm{i}}(\mathbf{x}) \leqslant J(\mathbf{x}). \tag{10.164}$$

$D(\boldsymbol{\mu})$ est donc une *borne inférieure du coût minimal du problème sous contraintes*

$$D(\boldsymbol{\mu}) \leqslant J(\widehat{\mathbf{x}}). \tag{10.165}$$

Puisque cette borne est valable pour tout $\boldsymbol{\mu} \geqslant \mathbf{0}$, on peut l'améliorer en résolvant le *problème dual*, c'est à dire en calculant les multiplicateurs de Lagrange optimaux

$$\widehat{\boldsymbol{\mu}} = \arg\max_{\boldsymbol{\mu} \geqslant \mathbf{0}} D(\boldsymbol{\mu}), \tag{10.166}$$

pour rendre la borne inférieure de (10.165) aussi grande que possible. Même si le problème initial (aussi appelé *problème primal*) n'est pas convexe, on a toujours

$$D(\widehat{\boldsymbol{\mu}}) \leqslant J(\widehat{\mathbf{x}}), \tag{10.167}$$

ce qui correspond à une relation de *dualité faible*. Le *saut de dualité optimal* est

$$J(\widehat{\mathbf{x}}) - D(\widehat{\boldsymbol{\mu}}) \geqslant 0. \tag{10.168}$$

La dualité est *forte* si ce saut est nul, ce qui veut dire que l'on peut inverser l'ordre de la maximisation du lagrangien par rapport à $\boldsymbol{\mu}$ et de sa minimisation par rapport à $\mathbf{x}$.

Une condition suffisante de dualité forte (connue comme la *condition de Slater*) est que la fonction de coût $J(\cdot)$ et les fonctions de contraintes $c^{\mathrm{i}}_j(\cdot)$ soient convexes et que l'intérieur de $\mathbb{X}$ ne soit pas vide. Cette condition devrait être satisfaite dans le contexte présent d'optimisation convexe (il devrait exister $\mathbf{x}$ tel que $c^{\mathrm{i}}_j(\mathbf{x}) < 0$, $j = 1, \cdots, n_{\mathrm{i}}$).

Faible ou forte, la dualité peut être utilisée pour définir des critères d'arrêt. Si $\mathbf{x}^k$ et $\boldsymbol{\mu}^k$ sont admissibles pour les problèmes primal et dual obtenus à l'itération $k$, alors

$$J(\widehat{\mathbf{x}}) \in [D(\boldsymbol{\mu}^k), J(\mathbf{x}^k)], \tag{10.169}$$

$$D(\widehat{\boldsymbol{\mu}}) \in [D(\boldsymbol{\mu}^k), J(\mathbf{x}^k)], \tag{10.170}$$

et le saut de dualité est donné par la longueur de l'intervalle $[D(\boldsymbol{\mu}^k), J(\mathbf{x}^k)]$. On peut s'arrêter dès que ce saut est considéré comme acceptable (en termes absolus ou relatifs).

### *10.7.5 Méthodes à points intérieurs*

En résolvant une succession de problèmes d'optimisation sans contrainte, les méthodes à points intérieurs génèrent des suites de paires $(\mathbf{x}^k, \boldsymbol{\mu}^k)$ telles que

— $\mathbf{x}^k$ est strictement à l'intérieur de $\mathbb{X}$,
— $\boldsymbol{\mu}^k$ est strictement admissible pour le problème dual (chaque multiplicateur de Lagrange est strictement positif),
— la longueur de l'intervalle $[D(\boldsymbol{\mu}^k), J(\mathbf{x}^k)]$ décroît quand $k$ croît.

Quand la dualité est forte, $(\mathbf{x}^k, \boldsymbol{\mu}^k)$ converge vers la solution optimale $(\widehat{\mathbf{x}}, \widehat{\boldsymbol{\mu}})$ quand $k$ tend vers l'infini, et ceci est vrai même quand $\widehat{\mathbf{x}}$ appartient à $\partial\mathbb{X}$.

Pour obtenir un point de départ $\mathbf{x}^0$, on peut calculer

$$(\widehat{w}, \mathbf{x}^0) = \arg\min_{w,\mathbf{x}} w \tag{10.171}$$

sous les contraintes

$$c^{\mathrm{i}}_j(\mathbf{x}) \leqslant w, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{10.172}$$

Si $\widehat{w} < 0$, alors $\mathbf{x}^0$ est strictement à l'intérieur de $\mathbb{X}$. Si $\widehat{w} = 0$ alors $\mathbf{x}^0$ appartient à $\partial\mathbb{X}$ et ne peut pas être utilisé pour une méthode à points intérieurs. Si $\widehat{w} > 0$, alors le problème initial n'a pas de solution.

Pour rester strictement à l'intérieur de $\mathbb{X}$, on peut utiliser une fonction barrière, en général la *barrière logarithmique* définie par (10.77), ou plus précisément par

$$p_{\log}(\mathbf{x}) = \begin{cases} -\sum_{j=1}^{n_{\mathrm{i}}} \ln[-c^{\mathrm{i}}_j(\mathbf{x})] & \text{si} \quad \mathbf{c}^{\mathrm{i}}(\mathbf{x}) < \mathbf{0}, \\ +\infty & \text{autrement .} \end{cases} \tag{10.173}$$

Cette barrière est différentiable et convexe à l'intérieur de $\mathbb{X}$ ; elle tend vers l'infini quand $\mathbf{x}$ tend vers $\partial\mathbb{X}$ de l'intérieur. On résout alors le problème de minimisation convexe sans contrainte

$$\widehat{\mathbf{x}}_\alpha = \arg\min_{\mathbf{x}} [J(\mathbf{x}) + \alpha p_{\log}(\mathbf{x})], \tag{10.174}$$

où $\alpha$ est un coefficient réel positif à choisir. Le lieu des $\widehat{\mathbf{x}}_\alpha$ pour $\alpha > 0$ est appelé *chemin central*, et chaque $\widehat{\mathbf{x}}_\alpha$ est un *point central*. En prenant $\alpha_k = \frac{1}{\beta_k}$, où $\beta_k$ est une fonction croissante de $k$, on peut calculer une suite de points centraux en résolvant une succession de *problèmes de minimisation convexe sans contrainte* pour $k = 1, 2, \ldots$ Le point central $\mathbf{x}^k$ est donné par

$$\mathbf{x}^k = \arg\min_{\mathbf{x}} \left[ J(\mathbf{x}) + \alpha_k p_{\log}(\mathbf{x}) \right] = \arg\min_{\mathbf{x}} \left[ \beta_k J(\mathbf{x}) + p_{\log}(\mathbf{x}) \right]. \tag{10.175}$$

Ceci peut être fait très efficacement par une méthode de type Newton, initialisée à $\mathbf{x}^{k-1}$ pour la recherche de $\mathbf{x}^k$. Plus $\beta_k$ devient grand et plus $\mathbf{x}^k$ s'approche de $\partial\mathbb{X}$, puisque le poids relatif du coût par rapport à la barrière augmente. Si $J(\mathbf{x})$ et $\mathbf{c}^{\mathrm{i}}(\mathbf{x})$ sont tous les deux différentiables, alors $\mathbf{x}^k$ doit satisfaire la condition d'optimalité au premier ordre

$$\beta_k \frac{\partial J}{\partial \mathbf{x}}(\mathbf{x}^k) + \frac{\partial p_{\log}}{\partial \mathbf{x}}(\mathbf{x}^k) = \mathbf{0}, \tag{10.176}$$

qui est nécessaire *et suffisante* puisque le problème est convexe. Un résultat important [28] est que

— tout point central $\mathbf{x}^k$ est admissible pour le problème primal,

— un point admissible pour le problème dual est

$$\mu_j^k = -\frac{1}{\beta_k c_j^{\mathrm{i}}(\mathbf{x}^k)}, \quad j = 1, \cdots, n_{\mathrm{i}}, \tag{10.177}$$

— et le saut de dualité est

$$J(\mathbf{x}^k) - D(\boldsymbol{\mu}^k) = \frac{n_{\mathrm{i}}}{\beta_k}, \tag{10.178}$$

où $n_{\mathrm{i}}$ est le nombre des contraintes d'inégalité.

*Remarque 10.11.* Puisque $\mathbf{x}^k$ est strictement à l'intérieur de $\mathbb{X}$, $c_j^{\mathrm{i}}(\mathbf{x}^k) < 0$ et $\mu_j^k$ donné par (10.177) est strictement positif. □

Le saut de dualité tend donc vers zéro quand $\beta_k$ tend vers l'infini, ce qui assure (au moins mathématiquement) que $\mathbf{x}^k$ tend vers une solution optimale du problème primal quand $k$ tend vers l'infini.

On peut prendre, par exemple,

$$\beta_k = \gamma \beta_{k-1}, \tag{10.179}$$

avec $\gamma > 1$ et $\beta_0 > 0$ à choisir. Deux types de problèmes peuvent advenir :

— quand $\beta_0$ et tout particulièrement $\gamma$ sont trop petits, on perd son temps à ramper le long du chemin central,

— quand ils sont trop grands, la recherche de $\mathbf{x}^k$ peut être mal initialisée et la méthode de Newton peut perdre du temps dans des itérations qui auraient pu être évitées.

### *10.7.6 Retour à la programmation linéaire*

Minimiser

$$J(\mathbf{x}) = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{10.180}$$

sous les contraintes *d'inégalité*

$$\mathbf{A}\mathbf{x} \leqslant \mathbf{b} \tag{10.181}$$

est un problème convexe, puisque la fonction de coût et le domaine admissible sont convexes. Le lagrangien s'écrit

$$L(\mathbf{x}, \boldsymbol{\mu}) = \mathbf{c}^{\mathrm{T}}\mathbf{x} + \boldsymbol{\mu}^{\mathrm{T}}(\mathbf{A}\mathbf{x} - \mathbf{b}) = -\mathbf{b}^{\mathrm{T}}\boldsymbol{\mu} + (\mathbf{A}^{\mathrm{T}}\boldsymbol{\mu} + \mathbf{c})^{\mathrm{T}}\mathbf{x}. \tag{10.182}$$

La fonction duale est telle que

$$D(\boldsymbol{\mu}) = \inf_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\mu}). \tag{10.183}$$

Puisque le lagrangien est affine en $\mathbf{x}$, l'infimum est $-\infty$ à moins que $\frac{\partial L}{\partial \mathbf{x}}$ ne soit identiquement nul, de sorte que

$$D(\boldsymbol{\mu}) = \begin{cases} -\mathbf{b}^{\mathrm{T}}\boldsymbol{\mu} & \text{si} \quad \mathbf{A}^{\mathrm{T}}\boldsymbol{\mu} + \mathbf{c} = \mathbf{0} \\ -\infty & \text{autrement} \end{cases}$$

et $\boldsymbol{\mu}$ est dual admissible si $\boldsymbol{\mu} \geqslant \mathbf{0}$ et $\mathbf{A}^{\mathrm{T}}\boldsymbol{\mu} + \mathbf{c} = \mathbf{0}$.

L'utilisation d'une barrière logarithmique conduit à calculer les points centraux

$$\mathbf{x}^k = \arg\min_{\mathbf{x}} J_k(\mathbf{x}), \tag{10.184}$$

où

$$J_k(\mathbf{x}) = \beta_k \mathbf{c}^{\mathrm{T}}\mathbf{x} - \sum_{j=1}^{n_{\mathrm{i}}} \ln\left(b_j - \mathbf{a}_j^{\mathrm{T}}\mathbf{x}\right), \tag{10.185}$$

avec $\mathbf{a}_j^{\mathrm{T}}$ la $j$-ème ligne de $\mathbf{A}$. Ceci est un problème de minimisation convexe sans contrainte, et donc facile. Une condition nécessaire et suffisante pour que $\mathbf{x}^k$ soit une solution de (10.184) est que

$$\mathbf{g}^k(\mathbf{x}^k) = \mathbf{0}, \tag{10.186}$$

avec $\mathbf{g}^k(\cdot)$ le gradient de $J_k(\cdot)$, trivial à calculer comme

$$\mathbf{g}^k(\mathbf{x}) = \frac{\partial J_k}{\partial \mathbf{x}}(\mathbf{x}) = \beta_k \mathbf{c} + \sum_{j=1}^{n_{\mathrm{i}}} \frac{1}{b_j - \mathbf{a}_j^{\mathrm{T}}\mathbf{x}}\, \mathbf{a}_j. \tag{10.187}$$

Pour rechercher $\mathbf{x}^k$ avec une méthode de Newton (amortie), il faut aussi disposer de la hessienne de $J_k(\cdot)$, donnée par

$$\mathbf{H}_k(\mathbf{x}) = \frac{\partial^2 J_k}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}}(\mathbf{x}) = \sum_{j=1}^{n_{\mathrm{i}}} \frac{1}{(b_j - \mathbf{a}_j^{\mathrm{T}}\mathbf{x})^2}\, \mathbf{a}_j \mathbf{a}_j^{\mathrm{T}}. \tag{10.188}$$

$\mathbf{H}_k$ est évidemment symétrique. Pourvu qu'il y ait $\dim \mathbf{x}$ vecteurs $\mathbf{a}_j$ linéairement indépendants, elle est aussi définie positive, de sorte qu'une méthode de Newton amortie devrait converger vers l'unique minimiseur global de (10.180) sous (10.181). On peut utiliser à la place une méthode de quasi-Newton ou de gradients conjugués qui n'utilise que des évaluations du gradient.

*Remarque 10.12.* La méthode interne de Newton, de quasi-Newton ou des gradients conjugués aura son propre compteur d'itérations, à ne pas confondre avec celui des itérations externes dénoté ici par $k$. □

L'équation (10.177) suggère de prendre comme vecteur dual associé à $\mathbf{x}^k$ le vecteur $\boldsymbol{\mu}^k$ ayant pour éléments

$$\mu_j^k = -\frac{1}{\beta_k c_j^{\mathrm{i}}(\mathbf{x}^k)}, \quad j = 1, \cdots, n_{\mathrm{i}}, \tag{10.189}$$

c'est à dire

$$\mu_j^k = \frac{1}{\beta_k (b_j - \mathbf{a}_j^{\mathrm{T}} \mathbf{x}^k)}, \quad j = 1, \cdots, n_{\mathrm{i}}. \tag{10.190}$$

Le saut de dualité

$$J(\mathbf{x}^k) - D(\boldsymbol{\mu}^k) = \frac{n_{\mathrm{i}}}{\beta_k} \tag{10.191}$$

peut servir à décider quand arrêter.

## 10.8 Optimisation sous contraintes à petit budget

Pour les cas où l'évaluation de la fonction de coût et/ou des contraintes est si coûteuse que le nombre des évaluations autorisées est très limité, la philosophie qui sous-tend EGO peut être étendue à l'optimisation sous contraintes [207, 206].

Des fonctions de pénalisation peuvent être utilisées pour transformer l'optimisation sous contraintes en une optimisation sans contrainte, à laquelle EGO peut alors être appliqué. Quand l'évaluation de contraintes est coûteuse, cette approche a l'avantage de construire un modèle de substitution qui prend en compte le coût initial et les contraintes. Le réglage des coefficients multiplicatifs appliqués aux fonctions de pénalisation n'est cependant pas trivial dans ce contexte.

Une approche alternative consiste à conduire une maximisation *sous contraintes* de l'espérance de l'amélion de la fonction de coût initiale. Ceci est particulièrement intéressant quand l'évaluation des contraintes est beaucoup moins coûteuse que celle de la fonction de coût initiale, car la maximisation de l'amélioration espérée de la sortie pénalisée d'un modèle de substitution de la fonction de coût initial sera relativement bon marché, même si les coefficients multiplicateurs de fonctions de pénalisation doivent être ajustés.

## 10.9 Exemples MATLAB

### *10.9.1 Programmation linéaire*

Trois méthodes principales de programmation linéaire sont mises en œuvre dans `linprog`, une fonction de l'*Optimization Toolbox* :

— une méthode primale-duale à points intérieurs pour des problèmes de grande taille,

— une méthode à ensemble actif (une variante de la programmation quadratique séquentielle) pour des problèmes de taille moyenne,

— un simplexe de Dantzig pour des problèmes de taille moyenne.

L'instruction `optimset('linprog')` liste les options par défaut. On lit notamment

```
   Display: 'final'
Diagnostics: 'off'
 LargeScale: 'on'
   Simplex: 'off'
```

Employons le simplexe de Dantzig sur l'exemple 10.6. La fonction `linprog` suppose que

— un coût linéaire doit être minimisé, de sorte que nous utilisons la fonction de coût (10.111), avec

$$\mathbf{c} = (-2,-1)^{\mathrm{T}}; \tag{10.192}$$

— les contraintes d'inégalité ne sont *pas* transformées en contraintes d'égalité, mais écrites comme

$$\mathbf{A}\mathbf{x} \leqslant \mathbf{b}\,; \tag{10.193}$$

nous posons donc

$$\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{et} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \tag{10.194}$$

— toute borne inférieure ou supérieure d'une variable de décision est donnée explicitement, de sorte que nous devons indiquer que la borne inférieure de chacune des deux variables de décision est nulle.

Ceci est mis en œuvre dans le script

```
clear all
c = [-2;
    -1];
A = [3  1;
    1  -1];
b = [1;
    0];
LowerBound = zeros(2,1);
```

```
% Imposons l'utilisation du simplexe
optionSIMPLEX = ...
    optimset('LargeScale','off','Simplex','on')
[OptimalX, OptimalCost] = ...
    linprog(c,A,b,[],[],LowerBound,...
    [],[],optionSIMPLEX)
```

Les crochets [] dans la liste des arguments d'entrée de `linprog` correspondent à des arguments non utilisés, tels que des bornes supérieures sur les variables de décision. Voir la documentation de la boîte à outils pour plus de détails. Ce script produit

```
Optimization terminated.

OptimalX =
     0
     1

OptimalCost =
    -1
```

ce qui ne doit pas nous surprendre.

### *10.9.2 Programmation non linéaire*

La fonction `patternsearch` de la *Global Optimization Toolbox* permet de traiter un mélange de contraintes d'égalité et d'inégalité, linéaires ou pas, en utilisant l'algorithme dit *Augmented Lagrangian Pattern Search* (ALPS) [44, 45, 143]. Les contraintes linéaires sont traitées séparément des contraintes non linéaires.

Considérons à nouveau l'exemple 10.5. La contrainte d'inégalité est si simple qu'on peut la mettre en œuvre en imposant une borne inférieure sur la variable de décision, comme dans le script qui suit, où tous les arguments non utilisés de `patternsearch` qui doivent être fournis sont remplacés par `[]` :

```
x0 = 0;
Cost = @(x) x.^2;
LowerBound = 1;
[xOpt,CostOpt] = patternsearch(Cost,x0,[],[],...
    [],[],LowerBound)
```

La solution trouvée est

```
xOpt = 1
CostOpt = 1
```

comme attendu.

Considérons maintenant l'exemple 10.3, où la fonction de coût $J(\mathbf{x}) = x_1^2 + x_2^2$ doit être minimisée sous la contrainte d'inégalité non linéaire $x_1^2 + x_2^2 + x_1 x_2 \geqslant 1$. Nous savons qu'il y a deux minimiseurs globaux

$$\widehat{\mathbf{x}}^3 = (1/\sqrt{3}, 1/\sqrt{3})^{\mathrm{T}}, \tag{10.195}$$

$$\widehat{\mathbf{x}}^4 = (-1/\sqrt{3}, -1/\sqrt{3})^{\mathrm{T}}, \tag{10.196}$$

où $1/\sqrt{3} \approx 0.57735026919$, et que $J(\widehat{\mathbf{x}}^3) = J(\widehat{\mathbf{x}}^4) = 2/3 \approx 0.66666666667$.

La fonction de coût est mise en œuvre dans la fonction

```
function Cost = L2cost(x)
Cost = norm(x)^2;
end
```

La contrainte d'inégalité non linéaire est écrite sous la forme $c(\mathbf{x}) \leqslant 0$, et mise en œuvre dans la fonction

```
function [c,ceq] = NLConst(x)
c = 1 - x(1)^2 - x(2)^2 - x(1)*x(2);
ceq = [];
end
```

Puisqu'il n'y a pas de contrainte d'égalité non linéaire, `ceq` est laissé vide, mais doit être présent. Finalement, `patternsearch` est appelé avec le script

```
clear all
format LONGE
x0 = [0;0];
x = zeros(2,1);
[xOpt,CostOpt] = patternsearch(@(x) ...
    L2cost(x),x0,[],[],...
    [],[],[],[], @(x) NLconst(x))
```

qui produit, après 4000 évaluations de la fonction de coût,

```
Optimization terminated: mesh size less
than options.TolMesh and constraint violation
is less than options.TolCon.

xOpt =
    -5.672302246093750e-01
    -5.882263183593750e-01

CostOpt =
     6.677603293210268e-01
```

La précision de cette solution peut être légèrement améliorée (au prix d'une forte augmentation du temps de calcul) en changeant les options de `patternsearch`, comme dans le script qui suit

```
clear all
format LONGE
x0 = [0;0];
x = zeros(2,1);
options = psoptimset('TolX',1e-10,'TolFun',...
    1e-10,'TolMesh',1e-12,'TolCon',1e-10,...
    'MaxFunEvals',1e5);
[xOpt,CostOpt] = patternsearch(@(x) ...
    L2cost(x),x0,[],[],...
    [],[],[],[], @(x) NLconst(x),options)
```

qui produit, après $10^5$ évaluations de la fonction de coût

```
Optimization terminated: mesh size less
than options.TolMesh and constraint violation
is less than options.TolCon.

xOpt =
    -5.757669508457184e-01
    -5.789321511983871e-01

CostOpt =
     6.666700173773681e-01
```

Voir la documentation de `patternsearch` pour plus de détails.

Ces résultats peu brillants suggèrent d'essayer d'autres approches. Avec la fonction de coût pénalisée

```
function Cost = L2costPenal(x)
Cost = x(1).^2+x(2).^2+1.e6*...
    max(0,1-x(1)^2-x(2)^2-x(1)*x(2));
end
```

le script

```
clear all
format LONGE
x0 = [1;1];
optionsFMS = optimset('Display',...
    'iter','TolX',1.e-10,'MaxFunEvals',1.e5);
[xHat,Jhat] = fminsearch(@(x) ...
    L2costPenal(x),x0,optionsFMS)
```

qui utilise tout simplement `fminsearch` produit

```
xHat =
     5.773502679858542e-01
     5.773502703933975e-01
```

```
Jhat =
     6.666666666666667e-01
```

après 284 évaluations de la fonction de coût pénalisée, sans même tenter de régler finement le coefficient multiplicatif de la fonction de pénalisation.

Quand on remplace sa première ligne par `x0 = [-1;-1];`, le même script produit

```
xHat =
    -5.773502679858542e-01
    -5.773502703933975e-01

Jhat =
     6.666666666666667e-01
```

ce qui suggère qu'il aurait été facile de trouver des approximations précises des deux solutions avec une stratégie *multistart*.

La programmation quadratique séquentielle telle que mise en œuvre dans la fonction `fmincon` de l'*Optimization Toolbox* est utilisée dans le script

```
clear all
format LONGE
x0 = [0;0];
x = zeros(2,1);
options = optimset('Algorithm','sqp');
[xOpt,CostOpt,exitflag,output] = fmincon(@(x) ...
    L2cost(x),x0,[],[],...
    [],[],[],[], @(x) NLconst(x),options)
```

qui produit

```
xOpt =
     5.773504749133580e-01
     5.773500634738818e-01

CostOpt =
     6.666666666759753e-01
```

en 94 évaluations de la fonction. Si l'on raffine les tolérances en remplaçant les options de `fmincon` dans le script précédent par

```
options = optimset('Algorithm','sqp',...
    'TolX',1.e-20, 'TolFun',1.e-20,'TolCon',1.e-20);
```

on obtient en 200 évaluations de la fonction

```
xOpt =
     5.773503628462886e-01
     5.773501755329579e-01
```

```
CostOpt =
     6.666666666666783e-01
```

qui est marginalement plus précis.

Pour utiliser l'algorithme à points intérieurs de `fmincon` au lieu de `sqp`, il suffit de remplacer les options de `fmincon` par

```
options = optimset('Algorithm','interior-point');
```

Le script résultant produit

```
xOpt =
     5.773510674737423e-01
     5.773494882274224e-01

CostOpt =
     6.666666866695364e-01
```

en 59 évaluations de la fonction. Si on raffine les tolérances en posant

```
options = optimset('Algorithm','interior-point',...
    'TolX',1.e-20, 'TolFun',1.e-20, 'TolCon',1.e-20);
```

on obtient, avec le même script,

```
xOpt =
     5.773502662973828e-01
     5.773502722550736e-01

CostOpt =
     6.666666668666664e-01
```

en 138 évaluation de la fonction.

*Remarque 10.13.* Les algorithmes `sqp` et `interior-point` satisfont les bornes éventuelles sur les variables de décision à chaque itération ; contrairement à `sqp`, `interior-point` peut traiter des problèmes creux de grande taille. □

## 10.10 En résumé

- Les contraintes jouent un rôle majeur dans la plupart des applications de l'optimisation à l'ingénierie.
- Même si une minimisation sans contrainte fournit un minimiseur admissible, ceci ne veut pas dire que les contraintes peuvent être négligées.
- Le domaine admissible $\mathbb{X}$ pour les variables de décision doit être non vide, et si possible fermé et borné.
- Le gradient du coût en un minimiseur sous contraintes n'est en général pas nul, et des conditions théoriques d'optimalité spécifiques doivent être considérées (les conditions KKT).

— La résolution formelle des équations KKT n'est possible que dans des problèmes simples, mais les conditions KKT jouent un rôle clé en programmation quadratique séquentielle.
— L'introduction de fonctions de pénalisation ou de fonctions barrières est l'approche la plus simple de l'optimisation sous contraintes (au moins du point de vue conceptuel), car elle permet d'utiliser des méthodes conçues pour l'optimisation sans contrainte. Il ne faut cependant pas sous-estimer ses difficultés numériques.
— L'approche par lagrangien augmenté facilite l'utilisation pratique des fonctions de pénalisation.
— Il est important de reconnaître un programme linéaire quand on en rencontre un, car des algorithmes spécifiques très puissants sont disponibles, tels que la méthode du simplexe de Dantzig.
— On peut dire la même chose de l'optimisation convexe, dont la programmation linéaire est un cas particulier.
— Les méthodes à points intérieurs peuvent traiter des problèmes de grande taille, qu'ils soient convexes ou pas.

# Chapitre 11
# Optimisation combinatoire

## 11.1 Introduction

Jusqu'ici, nous avons supposé que le domaine admissible $\mathbb{X}$ était tel qu'on pouvait y effectuer des déplacements infinitésimaux du vecteur de décision $\mathbf{x}$. Supposons maintenant que certaines variables de décision $x_i$ ne peuvent prendre que des valeurs discrètes, qui peuvent être codées avec des entiers. Il convient de distinguer deux situations.

Dans la première, les valeurs discrètes de $x_i$ ont un sens quantitatif. Une ordonnance peut par exemple recommander de prendre un nombre entier de pilules d'un type donné. On a alors $x_i \in \{0,1,2,\cdots\}$, et la prise de deux pilules implique l'ingestion de deux fois plus de principe actif qu'avec une seule pilule. On peut alors parler d'*optimisation en nombres entiers*. Une approche envisageable pour ce type de problème est d'introduire la contrainte

$$(x_i)(x_i-1)(x_i-2)(\cdots)=0, \tag{11.1}$$

au moyen d'une fonction de pénalisation, et de recourir ensuite à de l'optimisation sans contrainte.

Dans la seconde situation, celle considérée dans ce chapitre, les valeurs discrètes prises par les variables de décision n'ont pas de sens quantitatif, même si elles peuvent être codées avec des entiers. Considérons, par exemple, le fameux *problème du voyageur de commerce* (PVC), où toutes les villes d'une liste doivent être visitées en minimisant la distance totale à couvrir. Si la ville X est codée par 1 et la ville Y par 2, ceci ne signifie pas que la valeur de la ville Y est égale au double de celle de la ville X en un sens quelconque. La solution optimale est une liste ordonnée de noms de villes. Même si cette liste peut être décrite par une série ordonnée d'entiers (visiter la ville 45, puis la ville 12, puis...), on ne devrait pas confondre ceci avec de la programmation en nombres entiers, et on devrait plutôt parler d'*optimisation combinatoire*.

*Exemple 11.1.* Les problèmes d'optimisation combinatoire sont innombrables en ingénierie et en logistique. L'un d'entre eux est l'allocation de ressources (personnel, unités centrales, camions de livraison, etc.) à des tâches. On peut voir cette allocation comme le calcul d'un tableau optimal de noms de ressources en face de noms de tâches (la ressource $R_i$ doit traiter la tâche $T_j$, puis la tâche $T_k$, puis...). On peut souhaiter, par exemple, achever un travail au plus tôt sous des contraintes sur les ressources disponibles, ou minimiser les ressources nécessaires sous des contraintes sur la date d'achèvement. Des contraintes additionnelles peuvent s'imposer (la tâche $T_i$ ne peut pas commencer avant l'achèvement de la tâche $T_j$, par exemple), ce qui complique encore le problème. □

En optimisation combinatoire, la fonction de coût n'est pas différentiable par rapport aux variables de décision, et si l'on relaxait le problème pour le rendre différentiable en remplaçant les variables entières par des variables réelles, le gradient du coût n'aurait de toute façon aucun sens... Il faut donc des méthodes spécifiques. Nous ne ferons qu'effleurer le sujet dans la section suivante. On peut trouver beaucoup plus d'informations par exemple dans [178, 179, 180].

## 11.2 Recuit simulé

En métallurgie, le *recuit* est le procédé qui consiste à chauffer un matériau puis à le refroidir lentement. Ceci permet aux atomes d'atteindre un état d'énergie minimale et augmente la solidité. Le *recuit simulé* part de la même idée [134]. Bien qu'on puisse l'appliquer à des problèmes à variables continues, il est particulièrement utile pour les problèmes d'optimisation combinatoire, pourvu qu'on cherche une solution acceptable plutôt qu'une solution dont on puisse prouver l'optimalité.

La méthode, attribuée à Metropolis (1953), peut être résumée comme suit :

1. Tirer au hasard une solution candidate $\mathbf{x}^0$ (pour le PVC, une liste de villes à l'ordre aléatoire, par exemple), choisir une température initiale $\theta_0 > 0$ et poser $k = 0$.
2. Effectuer une transformation élémentaire de la solution candidate (pour le PVC, ce pourrait être l'échange de deux villes tirées au hasard dans la liste de la solution candidate) pour obtenir $\mathbf{x}^{k+}$.
3. Évaluer la variation de coût résultante $\Delta J_k = J(\mathbf{x}^{k+}) - J(\mathbf{x}^k)$ (pour le PVC, la variation de la distance à couvrir par le voyageur de commerce).
4. Si $\Delta J_k < 0$, alors *toujours* accepter la transformation et prendre $\mathbf{x}^{k+1} = \mathbf{x}^{k+}$
5. Si $\Delta J_k \geqslant 0$, alors *parfois* accepter la transformation et prendre $\mathbf{x}^{k+1} = \mathbf{x}^{k+}$, avec une probabilité $\pi_k$ qui décroît quand $\Delta J_k$ augmente mais qui croît quand la température $\theta_k$ augmente ; sinon, garder $\mathbf{x}^{k+1} = \mathbf{x}^k$.
6. Diminuer $\theta_k$, augmenter $k$ d'une unité et aller au pas 2.

En général, la probabilité d'accepter une modification qui détériore le coût est donnée par

$$\pi_k = \exp\left(-\frac{\Delta J_k}{\theta_k}\right), \tag{11.2}$$

par analogie avec la distribution de Boltzmann, avec la constante de Boltzmann prise égale à un. Ceci peut permettre d'échapper à des minimiseurs locaux parasites, pourvu que $\theta_0$ soit suffisamment grand et que la température décroisse assez lentement quand le compteur d'itérations $k$ est incrémenté.

On peut, par exemple, prendre $\theta_0$ grand par rapport à un $\Delta J$ typique évalué par quelques essais, puis faire décroître la température suivant $\theta_{k+1} = 0.9999\theta_k$. Une analyse théorique du recuit simulé vu comme une chaîne de Markov aide à voir la façon dont on doit faire décroître la température [156].

Bien qu'aucune garantie d'obtenir un résultat final optimal ne puisse être fournie, de nombreuses applications satisfaisantes de cette technique ont été rapportées. Un avantage significatif du recuit simulé par rapport à des techniques plus sophistiquées est la facilité avec laquelle on peut modifier la fonction de coût pour l'adapter au problème qu'il s'agit de résoudre. *Numerical Recipes* [186] présente ainsi d'amusantes variations du PVC, suivant que le passage d'un pays à un autre est considéré comme un inconvénient (parce qu'il faut passer par des ponts à péage) ou comme un avantage (parce que ça facilite la contrebande).

L'analogie avec la métallurgie peut être rendue encore plus convaincante en ayant un grand nombre de particules qui suivent la loi de Boltzmann. L'algorithme résultant est facile à paralléliser, et permet de détecter plusieurs minimiseurs. Si, pour n'importe quelle particule donnée, on refusait toute transformation qui accroîtrait le coût, on obtiendrait alors un simple algorithme de descente avec *multistart*, et la question de savoir si le recuit simulé fait mieux semble ouverte [8].

*Remarque 11.1.* Des techniques de *branch and bound* peuvent *trouver des solutions optimales certifiées* pour des PVC comportant des dizaines de milliers de villes, pour un coût de calcul énorme [4]. Il est plus simple de *certifier* qu'une solution candidate obtenue autrement est optimale, même pour des problèmes de très grande taille [5]. □

*Remarque 11.2.* Des méthodes à points intérieurs peuvent aussi être utilisées pour trouver des solutions approchées à des problèmes combinatoires dont on pense qu'ils sont de complexité NP, c'est à dire des problèmes pour lesquels on ne connaît pas d'algorithme dont la complexité dans le pire des cas soit bornée par un polynôme en la taille de l'entrée. Ceci montre une fois de plus le rôle unificateur de ces méthodes [255]. □

## 11.3 Exemple MATLAB

Considérons dix villes régulièrement espacées sur un cercle. Supposons qu'un voyageur de commerce habitant l'une d'entre elles vole en hélicoptère et puisse aller en ligne droite du centre de n'importe quelle ville vers le centre de n'importe quelle autre. Il y a alors $9! = 362880$ circuits possibles pour partir de la ville où habite ce

voyageur de commerce et y retourner après avoir visité une seule fois chacune des neuf autres villes. La longueur d'un circuit donné est calculée par la fonction

```
function [TripLength] = ...
    TravelGuide(X,Y,iOrder,NumCities)
TripLength = 0;
for i=1:NumCities-1,
    iStart=iOrder(i);
    iFinish=iOrder(i+1);
    TripLength = TripLength +...
        sqrt((X(iStart)-X(iFinish))^2+...
        (Y(iStart)-Y(iFinish))^2);
    % Retour à la maison
    TripLength=TripLength +...
        sqrt((X(iFinish)-X(iOrder(1)))^2+...
        (Y(iFinish)-Y(iOrder(1)))^2);
end
```

Le script qui suit explore $10^5$ circuits au hasard pour produire celui tracé sur la figure 11.2 en partant de celui tracé sur la figure 11.1. Ce résultat est clairement sous-optimal.

```
% X = table des longitudes des villes
% Y = table des latitudes des villes
% NumCities = nombre de villes
% InitialOrder = circuit
% utilisé comme point de départ
% FinalOrder = circuit finalement suggéré
NumCities = 10;
NumIterations = 100000;
for i=1:NumCities,
X(i)=cos(2*pi*(i-1)/NumCities);
Y(i)=sin(2*pi*(i-1)/NumCities);
end

% Tirage au hasard d'un ordre initial
% et tracé du circuit résultant
InitialOrder=randperm(NumCities);
for i=1:NumCities,
    InitialX(i)=X(InitialOrder(i));
    InitialY(i)=Y(InitialOrder(i));
end
% Retour à la maison
InitialX(NumCities+1)=X(InitialOrder(1));
InitialY(NumCities+1)=Y(InitialOrder(1));
figure;
plot(InitialX,InitialY)
```

```
% Début du recuit simulé
Temp = 1000; % température initiale
Alpha=0.9999; % taux de décroissance de la température
OldOrder = InitialOrder
for i=1:NumIterations,
    OldLength=TravelGuide(X,Y,OldOrder,NumCities);
    % Modification aléatoire du circuit
    NewOrder=randperm(NumCities);
    % Calcul de la longueur du circuit résultant
    NewLength=TravelGuide(X,Y,NewOrder,NumCities);
    r=random('Uniform',0,1);
    if (NewLength<OldLength)||...
            (r < exp(-(NewLength-OldLength)/Temp))
        OldOrder=NewOrder;
    end
    Temp=Alpha*Temp;
end

% Acceptation de la suggestion finale
% et retour à la maison
FinalOrder=OldOrder;
for i=1:NumCities,
    FinalX(i)=X(FinalOrder(i));
    FinalY(i)=Y(FinalOrder(i));
end
FinalX(NumCities+1)=X(FinalOrder(1));
FinalY(NumCities+1)=Y(FinalOrder(1));

% Tracé du circuit suggéré
figure;
plot(FinalX,FinalY)
```

Il suffirait d'échanger les positions de deux villes spécifiques du circuit décrit par la figure 11.2 pour le rendre optimal, mais cet échange ne peut arriver avec le script précédent (à moins que `randperm` ne se révèle échanger directement ces deux villes tout en gardant le même ordre pour toutes les autres, ce qui est fort peu probable). Il faut donc rendre moins drastiques les modifications du circuit à chaque itération. Ceci peut être fait en remplaçant dans le script précédent

```
NewOrder=randperm(NumCities);
```

par

```
NewOrder=OldOrder;
Tempo=randperm(NumCities);
NewOrder(Tempo(1))=OldOrder(Tempo(2));
NewOrder(Tempo(2))=OldOrder(Tempo(1));
```

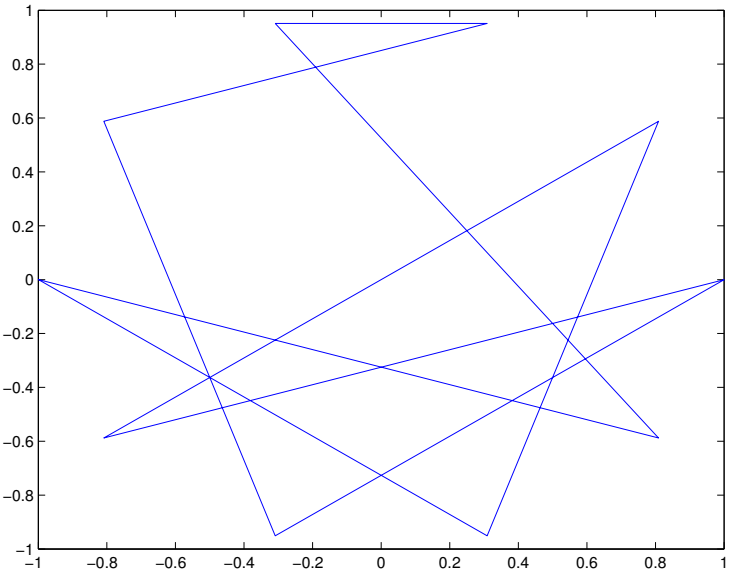


**Fig. 11.1** Circuit initial pour dix villes

A chaque itération, deux villes tirées au hasard sont ainsi échangées, tandis que toutes les autres sont laissées à leur place. Le script ainsi modifié produit en $10^5$ itérations le circuit optimal représenté en figure 11.3 (mais il n'y a pas de garantie qu'il y parvienne). Avec vingt villes (et $19! \approx 1.2 \cdot 10^{17}$ circuits partant de la résidence du voyageur de commerce et y retournant), le même algorithme produit aussi un solution optimale après $10^5$ échanges de villes.

Il n'est pas clair que la décroissance de la température soit utile dans cet exemple particulier. Le script qui suit refuse toute modification du circuit qui augmenterait la distance à couvrir, et il produit pourtant le circuit optimal de la figure 11.5 à partir du circuit de la figure 11.4 pour un problème à vingt villes.

```
NumCities = 20;
NumIterations = 100000;
for i=1:NumCities,
X(i)=cos(2*pi*(i-1)/NumCities);
Y(i)=sin(2*pi*(i-1)/NumCities);
end
InitialOrder=randperm(NumCities);
for i=1:NumCities,
    InitialX(i)=X(InitialOrder(i));
    InitialY(i)=Y(InitialOrder(i));
```

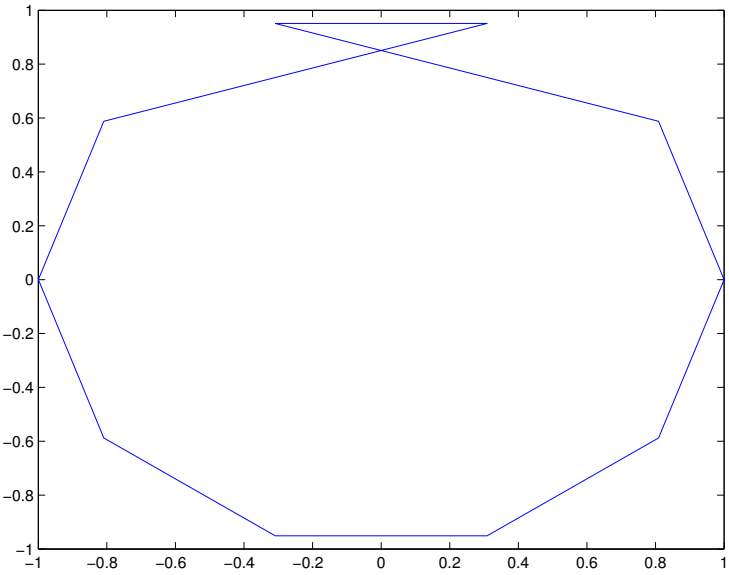


**Fig. 11.2** Circuit sous-optimal suggéré par le recuit simulé pour le problème à dix villes après la génération aléatoire de $10^5$ circuits

```
end
InitialX(NumCities+1)=X(InitialOrder(1));
InitialY(NumCities+1)=Y(InitialOrder(1));

% Tracé du circuit initial
figure;
plot(InitialX,InitialY)
OldOrder = InitialOrder
for i=1:NumIterations,
    OldLength=TravelGuide(X,Y,OldOrder,NumCities);

    % Echange de deux villes tirées au hasard
    NewOrder = OldOrder;
    Tempo=randperm(NumCities);
    NewOrder(Tempo(1)) = OldOrder(Tempo(2));
    NewOrder(Tempo(2)) = OldOrder(Tempo(1));

    % Calcul de la longueur du meilleur circuit
    NewLength=TravelGuide(X,Y,NewOrder,NumCities);
    if(NewLength<OldLength)
```

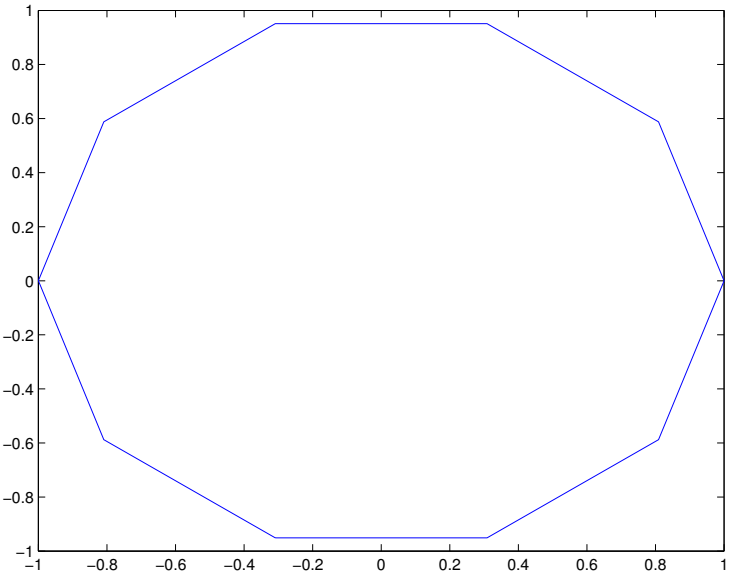


**Fig. 11.3** Circuit optimal suggéré par le recuit simulé pour le problème à dix villes après la génération de $10^5$ échanges de deux villes tirées au hasard

```
        OldOrder=NewOrder;
    end
end

% Acceptation de la suggestion finale
% et retour à la maison
FinalOrder=OldOrder;
for i=1:NumCities,
    FinalX(i)=X(FinalOrder(i));
    FinalY(i)=Y(FinalOrder(i));
end
FinalX(NumCities+1)=X(FinalOrder(1));
FinalY(NumCities+1)=Y(FinalOrder(1));

% Tracé du circuit suggéré
figure;
plot(FinalX,FinalY)
end
```

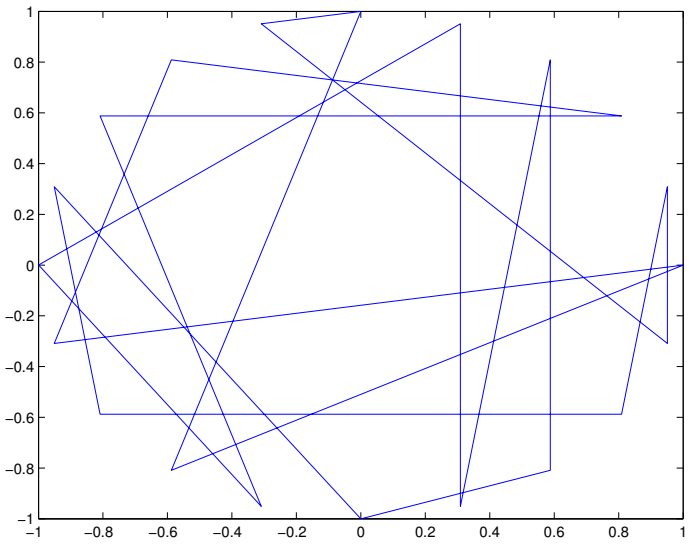


**Fig. 11.4** Circuit initial pour vingt villes

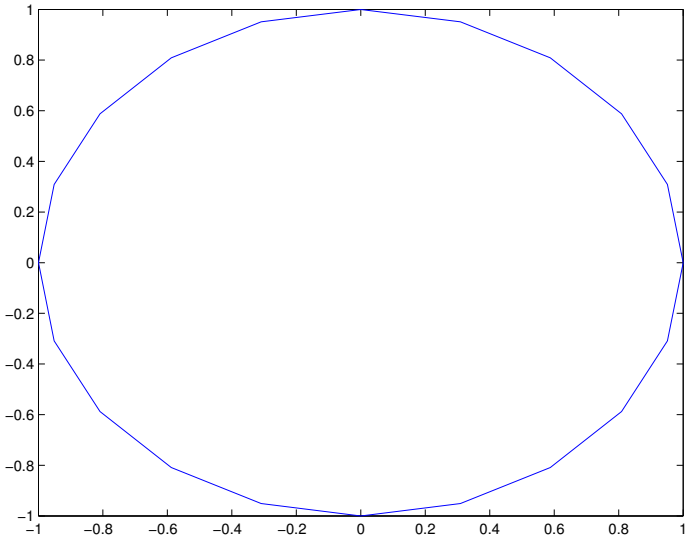


**Fig. 11.5** Circuit optimal pour le problème à vingt villes, obtenue après la génération de $10^5$ échanges de deux villes tirées au hasard ; aucune augmentation de la longueur du circuit n'a été acceptée

# Chapitre 12
# Simuler des équations différentielles ordinaires

## 12.1 Introduction

Les équations différentielles jouent un rôle crucial dans la simulation de systèmes physiques, et les solutions de la plupart d'entre elles ne peuvent être calculées que numériquement. Nous ne considérons ici que le cas *déterministe* ; pour une introduction pratique à la simulation numérique d'équations différentielles stochastiques, voir [98]. Les *équations différentielles ordinaires* (EDO), qui n'ont qu'une variable indépendante, sont traitées en premier, car c'est le cas le plus simple. Les *équations aux dérivées partielles* (EDP) sont pour le chapitre 13. Des références classiques sur la résolution des EDO sont [68] et [227]. On trouvera dans [87] et [210] des informations sur des codes populaires pour résoudre des EDO. Des compléments utiles pour qui a l'intention d'utiliser les solveurs d'EDO de MATLAB sont dans [216, 212, 7, 215, 214] et dans le chapitre 7 de [158].

La plupart des méthodes de résolution numérique d'EDO supposent ces EDO sous la forme

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t), \tag{12.1}$$

où $\mathbf{x}$ est un vecteur de $\mathbb{R}^n$, avec $n$ l'*ordre* de l'EDO, et où $t$ est la variable indépendante. Cette variable est souvent associée au temps, et c'est comme ça que nous l'appellerons, mais elle peut tout aussi bien correspondre à une autre quantité, comme dans l'exemple de la section 12.4.4. L'équation (12.1) définit un système de $n$ équations différentielles scalaires du premier ordre. Pour une valeur de $t$ donnée, la valeur de $\mathbf{x}(t)$ est l'*état* de ce système, et (12.1) est une *équation d'état*.

*Remarque 12.1.* Le fait que la fonction vectorielle $\mathbf{f}$ dans (12.1) dépende explicitement de $t$ permet de considérer des EDO forcées par un signal d'entrée $\mathbf{u}(t)$, pourvu que $\mathbf{u}(t)$ puisse être évalué en tout $t$ pour lequel $\mathbf{f}$ doit l'être. □

*Exemple 12.1.* Les équations de la cinétique chimique dans les réacteurs continus parfaitement agités sont naturellement sous forme d'état, avec les concentrations des espèces chimiques comme variables d'état. Considérons, par exemple, les deux réactions élémentaires

$$A + 2B \longrightarrow 3C \quad \text{et} \quad A + C \longrightarrow 2D. \tag{12.2}$$

Les équations cinétiques correspondantes sont

$$\begin{aligned} [\dot{A}] &= -k_1[A]\cdot[B]^2 - k_2[A]\cdot[C], \\ [\dot{B}] &= -2k_1[A]\cdot[B]^2, \\ [\dot{C}] &= 3k_1[A]\cdot[B]^2 - k_2[A]\cdot[C], \\ [\dot{D}] &= 2k_2[A]\cdot[C], \end{aligned} \tag{12.3}$$

où $[X]$ correspond à la concentration de l'espèce $X$. (Les constantes de vitesse $k_1$ et $k_2$ des deux réactions élémentaires sont en fait des fonctions de la température, qui peut être maintenue constante ou autrement contrôlée.) □

*Exemple 12.2.* Les modèles à compartiments [114], très utilisés en biologie et en pharmacocinétique, sont constitués de réservoirs (représentés par des disques) qui échangent des quantités de matière comme indiqué par des flèches (figure 12.1). Leur équation d'état est obtenue par bilan matière. Le modèle à deux compartiments

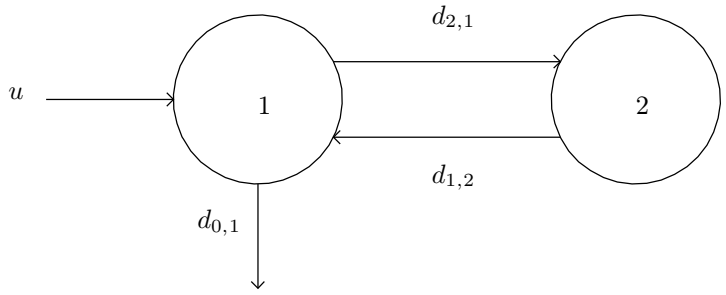


**Fig. 12.1** Exemple de modèle à compartiments

de la figure 12.1 correspond à

$$\begin{aligned} \dot{x}_1 &= -(d_{0,1} + d_{2,1}) + d_{1,2} + u, \\ \dot{x}_2 &= d_{2,1} - d_{1,2}, \end{aligned} \tag{12.4}$$

avec $u$ un débit entrant, $x_i$ la quantité de matière dans le compartiment $i$ et $d_{i,j}$ le débit de matière du compartiment $j$ vers le compartiment $i$, qui est une fonction du vecteur d'état $\mathbf{x}$. (L'extérieur est considéré comme un compartiment spécial

supplémentaire indexé par 0.) Si, comme on le suppose souvent, chaque débit de matière est proportionnel à la quantité de matière dans le compartiment donneur :

$$d_{i,j} = \theta_{i,j} x_j, \tag{12.5}$$

alors l'équation d'état devient

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}, \tag{12.6}$$

qui est linéaire en le vecteur $\mathbf{u}$ des débits entrants, avec $\mathbf{A}$ une fonction des $\theta_{i,j}$. Pour le modèle de la figure 12.1,

$$\mathbf{A} = \begin{bmatrix} -(\theta_{0,1} + \theta_{2,1}) & \theta_{1,2} \\ \theta_{2,1} & -\theta_{1,2} \end{bmatrix} \tag{12.7}$$

et $\mathbf{B}$ devient

$$\mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{12.8}$$

car l'entrée est scalaire. □

*Remarque 12.2.* Bien que (12.6) soit linéaire en ses entrées, sa solution est fortement non linéaire en $\mathbf{A}$. Ceci a des conséquences si les paramètres inconnus $\theta_{i,j}$ doivent être estimés à partir de résultats de mesures

$$\mathbf{y}(t_i) = \mathbf{C}\mathbf{x}(t_i), \quad i = 1, \cdots, N, \tag{12.9}$$

en minimisant une fonction de coût. Même si cette fonction de coût est quadratique en l'erreur, la méthode des moindres carrés linéaires ne pourra pas être appliquée, car la fonction de coût ne sera pas quadratique en les paramètres. □

*Remarque 12.3.* Quand la fonction vectorielle $\mathbf{f}$ dans (12.1) dépend non seulement de $\mathbf{x}(t)$ mais aussi de $t$, on peut se débarrasser formellement de la dépendance en $t$ en considérant le vecteur d'état étendu

$$\mathbf{x}^{\mathrm{e}}(t) = \begin{bmatrix} \mathbf{x} \\ t \end{bmatrix}. \tag{12.10}$$

Ce vecteur satisfait l'équation d'état étendue

$$\dot{\mathbf{x}}^{\mathrm{e}}(t) = \begin{bmatrix} \dot{\mathbf{x}}(t) \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{f}(\mathbf{x}, t) \\ 1 \end{bmatrix} = \mathbf{f}^{\mathrm{e}}\left(\mathbf{x}^{\mathrm{e}}(t)\right), \tag{12.11}$$

où la fonction vectorielle $\mathbf{f}^{\mathrm{e}}$ ne dépend que de l'état étendu. □

Parfois, la mise sous forme d'état demande un peu de travail, comme dans l'exemple suivant, qui correspond à une vaste classe d'EDO.

*Exemple 12.3.* Toute EDO scalaire d'ordre $n$ qui peut s'écrire

$$y^{(n)} = f(y, \dot{y}, \ldots, y^{(n-1)}, t) \tag{12.12}$$

peut être mise sous la forme (12.1) en posant

$$\mathbf{x} = \begin{bmatrix} y \\ \dot{y} \\ \vdots \\ y^{(n-1)} \end{bmatrix}. \tag{12.13}$$

En effet,

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \vdots \\ y^{(n)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & 0 & 1 & 0 & \cdots \\ \vdots & 0 & \ddots & \ddots & \ddots \\ 0 & \cdots & \cdots & 0 & 1 \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix} g(\mathbf{x}, t) = \mathbf{f}(\mathbf{x}, t), \tag{12.14}$$

avec

$$g(\mathbf{x}, t) = f(y, \dot{y}, \ldots, y^{(n-1)}, t). \tag{12.15}$$

La solution $y(t)$ de l'EDO scalaire initiale est alors dans la première composante de $\mathbf{x}(t)$. □

*Remarque 12.4.* Ce qui précède n'est qu'une des méthodes utilisables pour obtenir une équation d'état à partir d'une EDO scalaire. Tout changement de base $\mathbf{z} = \mathbf{T}\mathbf{x}$ dans l'espace d'état, où $\mathbf{T}$ est inversible et indépendante de $t$, conduit à une autre équation d'état :

$$\dot{\mathbf{z}} = \mathbf{T}\mathbf{f}(\mathbf{T}^{-1}\mathbf{z}, t). \tag{12.16}$$

La solution $y(t)$ de l'EDO scalaire initiale est alors obtenue comme

$$y(t) = \mathbf{c}^{\mathrm{T}}\mathbf{T}^{-1}\mathbf{z}(t), \tag{12.17}$$

avec

$$\mathbf{c}^{\mathrm{T}} = (1\ 0 \cdots 0). \tag{12.18}$$

□

Pour spécifier complètement la solution de (12.1), il faut lui imposer des contraintes. Nous distinguons

— les *problèmes aux valeurs initiales*, où ces contraintes fixent la valeur de $\mathbf{x}$ pour une seule valeur $t_0$ de $t$ et où la solution $\mathbf{x}(t)$ est à calculer pour $t \geqslant t_0$,

— les *problèmes aux limites*, et en particulier les problèmes aux deux bouts où ces contraintes fournissent une information partielle sur $\mathbf{x}(t_{\min})$ et $\mathbf{x}(t_{\max})$ et où la solution $\mathbf{x}(t)$ est à calculer pour $t_{\min} \leqslant t \leqslant t_{\max}$.

Idéalement, le solveur d'EDO devrait choisir, à partir des spécifications du problème,

— une famille d'algorithmes d'intégration,
— un membre de cette famille,
— un pas d'intégration.

Il devrait aussi adapter ces choix pendant la simulation quand c'est utile. Ceci explique pourquoi les algorithmes d'intégration ne représentent qu'une petite portion du code de solveurs d'EDO de classe professionnelle. Nous nous limitons ici à une brève description des principales familles de méthodes d'intégration (avec leurs avantages et leurs limites) et des techniques d'adaptation du pas d'intégration. Nous commençons par les problèmes aux valeurs initiales (section 12.2), car ils sont plus simples que les problèmes aux limites traités en section 12.3.

## 12.2 Problèmes aux valeurs initiales

Le type de problème considéré ici est le calcul numérique, pour $t \geqslant t_0$, de la solution $\mathbf{x}(t)$ du système

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t), \tag{12.19}$$

pour la condition initiale

$$\mathbf{x}(t_0) = \mathbf{x}_0, \tag{12.20}$$

où la valeur numérique de $\mathbf{x}_0$ est connue. L'équation (12.20) est une *condition de Cauchy*, et ceci est un *problème de Cauchy*.

Nous supposerons que la solution de (12.19) pour la condition initiale (12.20) existe et est unique. Quand $\mathbf{f}(\cdot,\cdot)$ est défini sur un ensemble ouvert $\mathbb{U} \subset \mathbb{R}^n \times \mathbb{R}$, une condition *suffisante* pour que tel soit le cas dans $\mathbb{U}$ est que $\mathbf{f}$ vérifie une *condition de Lipschitz* par rapport à $\mathbf{x}$, uniformément par rapport à $t$. Ceci signifie qu'il existe une constante $L \in \mathbb{R}$ telle que

$$\forall (\mathbf{x},\mathbf{y},t) : (\mathbf{x},t) \in \mathbb{U} \text{ et } (\mathbf{y},t) \in \mathbb{U}, ||\mathbf{f}(\mathbf{x},t) - \mathbf{f}(\mathbf{y},t)|| \leqslant L \cdot ||\mathbf{x} - \mathbf{y}||. \tag{12.21}$$

*Remarque 12.5.* Des phénomènes étranges peuvent apparaître quand cette condition de Lipschitz n'est pas satisfaite, comme pour les problèmes de Cauchy

$$\dot{x} = -px^2, \quad x(0) = 1, \tag{12.22}$$

et

$$\dot{x} = -x + x^2, \quad x(0) = p. \tag{12.23}$$

Il est facile de vérifier que (12.22) admet la solution

$$x(t) = \frac{1}{1+pt}. \tag{12.24}$$

Quand $p > 0$, cette solution est valide pour tout $t \geqslant 0$, mais quand $p < 0$, elle s'échappe en temps fini : elle tend vers l'infini quand $t$ tend vers $-1/p$ et n'est valide que pour $t \in [0, -1/p)$.

La nature de la solution de (12.23) dépend de la magnitude de $p$. Quand $|p|$ est suffisamment petit, l'effet du terme quadratique est négligeable et la solution est approximativement égale à $p\exp(-t)$, tandis que quand $|p|$ est suffisamment large le terme quadratique domine et la solution s'échappe en temps fini. □

*Remarque 12.6.* L'instant final $t_\mathrm{f}$ du calcul de la solution peut ne pas être connu à l'avance, et être défini comme le premier instant tel que

$$h\left(\mathbf{x}\left(t_\mathrm{f}\right),t_\mathrm{f}\right)=0, \tag{12.25}$$

où $h(\mathbf{x},t)$ est une *fonction d'événement* qui dépend du problème considéré. Un exemple typique est la simulation d'un système hybride qui commute entre des comportements continus décrits par des EDO, et où l'on change d'EDO quand l'état traverse une frontière. Un nouveau problème de Cauchy avec une autre EDO et un autre temps initial $t_\mathrm{f}$ doit alors être considéré. Une balle qui tombe au sol avant de rebondir est un exemple très simple d'un tel système hybride, où l'EDO à utiliser une fois que la balle a commencé à heurter le sol diffère de celle utilisée pour décrire sa chute libre. Certains solveurs peuvent localiser des événements et redémarrer l'intégration de façon à tenir compte du changement d'EDO [74, 217]. □

### *12.2.1 Cas linéaire stationnaire*

Un cas particulier important est quand $\mathbf{f}(\mathbf{x},t)$ est linéaire en $\mathbf{x}$ et ne dépend pas explicitement de $t$, de sorte qu'elle peut s'écrire comme

$$\mathbf{f}(\mathbf{x},t)\equiv\mathbf{A}\mathbf{x}(t), \tag{12.26}$$

où $\mathbf{A}$ est une matrice carrée constante et connue numériquement. La solution du problème de Cauchy est alors

$$\mathbf{x}(t)=\exp[\mathbf{A}(t-t_0)]\cdot\mathbf{x}(t_0), \tag{12.27}$$

où $\exp[\mathbf{A}(t-t_0)]$ est une *exponentielle de matrice*, qui peut être calculée de multiples façons [159].

Pourvu que la norme de $\mathbf{M}=\mathbf{A}(t-t_0)$ soit suffisamment petite, on peut utiliser un développement de Taylor tronqué à l'ordre $q$

$$\exp\,\mathbf{M}\approx\mathbf{I}+\mathbf{M}+\frac{1}{2}\mathbf{M}^2+\cdots+\frac{1}{q!}\mathbf{M}^q, \tag{12.28}$$

ou une approximation de Padé $(p,p)$

$$\exp\,\mathbf{M}\approx[D_p(\mathbf{M})]^{-1}N_p(\mathbf{M}), \tag{12.29}$$

avec $N_p(\mathbf{M})$ et $D_p(\mathbf{M})$ des polynômes d'ordre $p$ en $\mathbf{M}$. Les coefficients de ces polynômes sont choisis pour que le développement de Taylor de l'approximation de Padé soit identique à celui de $\exp\mathbf{M}$ jusqu'à l'ordre $q = 2p$. On a ainsi

$$N_p(\mathbf{M}) = \sum_{j=0}^{p} c_j \, \mathbf{M}^j \tag{12.30}$$

et

$$D_p(\mathbf{M}) = \sum_{j=0}^{p} c_j \, (-\mathbf{M})^j, \tag{12.31}$$

avec

$$c_j = \frac{(2p-j)!\; p!}{(2p)!\; j!\; (p-j)!}. \tag{12.32}$$

Quand $\mathbf{A}$ peut être diagonalisée par une matrice $\mathbf{T}$ de changement de base dans l'espace des états, telle que

$$\boldsymbol{\Lambda} = \mathbf{T}^{-1}\mathbf{A}\mathbf{T}, \tag{12.33}$$

les éléments diagonaux de $\boldsymbol{\Lambda}$ sont les valeurs propres $\lambda_i$ de $\mathbf{A}$ $(i = 1, \cdots, n)$, et

$$\exp[\mathbf{A}(t-t_0)] = \mathbf{T} \cdot \exp[\boldsymbol{\Lambda}(t-t_0)] \cdot \mathbf{T}^{-1}, \tag{12.34}$$

où l'élément en position $(i,i)$ de la matrice diagonale $\exp[\boldsymbol{\Lambda}(t-t_0)]$ est égal à $\exp[\lambda_i(t-t_0)]$. Cette approche par diagonalisation facilite l'évaluation de $\mathbf{x}(t_i)$ en des instants $t_i$ arbitrairement placés.

La méthode dite de *scaling and squaring* [159, 2, 103], fondée sur la relation

$$\exp\,\mathbf{M} = \left[\exp\left(\frac{\mathbf{M}}{m}\right)\right]^m, \tag{12.35}$$

est l'une des plus populaires pour le calcul d'exponentielles de matrices. Elle est mise en œuvre en MATLAB par la fonction `expm`. Lors de la mise à l'échelle (*scaling*), $m$ est la plus petite puissance de deux telle que $\|\mathbf{M}/m\| < 1$. Une approximation de Taylor ou de Padé est alors utilisée pour évaluer $\exp(\mathbf{M}/m)$, avant d'évaluer $\exp\mathbf{M}$ par des élévations au carré répétées (*squaring*).

### *12.2.2 Cas général*

Toutes les méthodes présentées dans cette section impliquent une taille de pas $h$ positive sur la variable indépendante $t$, et $h$ est supposé constant pour le moment. Pour simplifier les notations, nous écrivons

$$\mathbf{x}_l = \mathbf{x}(t_l) = \mathbf{x}(t_0 + lh) \tag{12.36}$$

et

$$\mathbf{f}_l = \mathbf{f}(\mathbf{x}(t_l), t_l). \tag{12.37}$$

Les méthodes les plus simples pour la résolution des problèmes aux valeurs initiales sont celles d'Euler.

#### 12.2.2.1 Méthodes d'Euler

Partant de $\mathbf{x}_l$, la *méthode d'Euler explicite* évalue $\mathbf{x}_{l+1}$ via un développement de Taylor au premier ordre de $\mathbf{x}(t)$ au voisinage de $t = t_l$

$$\mathbf{x}(t_l + h) = \mathbf{x}(t_l) + h\dot{\mathbf{x}}(t_l) + \mathrm{o}(h). \tag{12.38}$$

Elle pose donc

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h\mathbf{f}_l. \tag{12.39}$$

C'est une méthode à *un pas*, car l'évaluation de $\mathbf{x}(t_{l+1})$ n'utilise que la valeur de $\mathbf{x}(t_l)$. L'erreur de méthode sur un pas (ou *erreur de méthode locale*) est génériquement en $\mathrm{O}(h^2)$ (sauf si $\ddot{\mathbf{x}}(t_l) = \mathbf{0}$).

L'équation (12.39) revient à remplacer $\dot{\mathbf{x}}$ dans (12.1) par l'approximation par différence finie *avant*

$$\dot{\mathbf{x}}(t_l) \approx \frac{\mathbf{x}_{l+1} - \mathbf{x}_l}{h}. \tag{12.40}$$

Comme l'évaluation de $\mathbf{x}_{l+1}$ par (12.39) n'utilise que la valeur passée $\mathbf{x}_l$ de $\mathbf{x}$, la méthode d'Euler explicite est une *méthode de prédiction*.

On peut aussi remplacer $\dot{\mathbf{x}}$ dans (12.1) par l'approximation par différence finie *arrière*

$$\dot{\mathbf{x}}(t_{l+1}) \approx \frac{\mathbf{x}_{l+1} - \mathbf{x}_l}{h}, \tag{12.41}$$

pour obtenir

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h\mathbf{f}_{l+1}. \tag{12.42}$$

Puisque $\mathbf{f}_{l+1}$ dépend de $\mathbf{x}_{l+1}$, $\mathbf{x}_{l+1}$ est maintenant obtenu en résolvant une équation implicite, et ceci est la *méthode d'Euler implicite*. Cette méthode a de meilleures propriétés de stabilité que sa contrepartie explicite, comme illustré dans l'exemple qui suit.

*Exemple 12.4.* Considérons l'équation différentielle scalaire du premier ordre

$$\dot{x} = \lambda x, \tag{12.43}$$

où $\lambda$ est une constante réelle négative, de sorte que (12.43) est asymptotiquement stable. Ceci signifie que $x(t)$ tend vers zéro quand $t$ tend vers l'infini. La méthode d'Euler explicite calcule

$$x_{l+1} = x_l + h(\lambda x_l) = (1 + \lambda h)x_l. \tag{12.44}$$

Cette récurrence est asymptotiquement stable si et seulement si $|1+\lambda h|<1$, ou encore si $0<-\lambda h<2$. Ceci est à comparer avec la méthode d'Euler implicite, qui calcule

$$x_{l+1}=x_l+h(\lambda x_{l+1}). \tag{12.45}$$

L'équation implicite (12.45) peut être explicitée comme

$$x_{l+1}=\frac{1}{1-\lambda h}x_l. \tag{12.46}$$

Cette récurrence est asymptotiquement stable pour toute taille de pas $h$ puisque $\lambda<0$ et $0<\frac{1}{1-\lambda h}<1$. □

Sauf quand (12.42) peut être explicitée (comme dans l'exemple 12.4), la méthode d'Euler implicite est plus compliquée à mettre en œuvre que sa contrepartie explicite, et ceci est vrai de toutes les autres méthodes implicites qui seront présentées, voir la section 12.2.2.4.

#### 12.2.2.2 Méthodes de Runge-Kutta

Il semble naturel d'aller au delà des méthodes d'Euler en utilisant un développement de Taylor de $\mathbf{x}(t)$ au voisinage de $t_l$ à un ordre plus élevé

$$\mathbf{x}_{l+1}=\mathbf{x}_l+h\dot{\mathbf{x}}(t_l)+\cdots+\frac{h^k}{k!}\mathbf{x}^{(k)}(t_l)+\mathrm{o}(h^k). \tag{12.47}$$

Les calculs se compliquent cependant quand $k$ augmente, car il faut évaluer des dérivées d'ordre plus élevé de $\mathbf{x}$ par rapport à $t$. Ceci fut utilisé comme un argument en faveur des méthodes de Runge-Kutta, qui sont beaucoup plus communément utilisées [35]. Les équations d'une *méthode de Runge-Kutta* d'ordre $k$, notée $\mathrm{RK}(k)$, sont choisis pour que les coefficients du développement de Taylor de $\mathbf{x}_{l+1}$ tels que calculés avec $\mathrm{RK}(k)$ soient identiques à ceux de (12.47) jusqu'à l'ordre $k$.

*Remarque 12.7.* L'ordre $k$ d'une méthode numérique de résolution d'EDO est celui de l'erreur de méthode, à ne pas confondre avec l'ordre $n$ de l'EDO. □

La solution de (12.1) entre $t_l$ et $t_{l+1}=t_l+h$ satisfait

$$\mathbf{x}(t_{l+1})=\mathbf{x}(t_l)+\int_{t_l}^{t_{l+1}}\mathbf{f}(\mathbf{x}(\tau),\tau)\mathrm{d}\tau. \tag{12.48}$$

Ceci suggère d'utiliser une quadrature numérique, comme au chapitre 6, et d'écrire

$$\mathbf{x}_{l+1}=\mathbf{x}_l+h\sum_{i=1}^{q}b_i\mathbf{f}(\mathbf{x}(t_{l,i}),t_{l,i}), \tag{12.49}$$

où $\mathbf{x}_l$ est une approximation de $\mathbf{x}(t_l)$ supposée disponible, et où

$$t_{l,i} = t_l + \tau_i h, \tag{12.50}$$

avec $0 \leqslant \tau_i \leqslant 1$. Le problème est cependant plus difficile qu'au chapitre 6, parce que la valeur de $\mathbf{x}(t_{l,i})$ qui apparaît dans (12.49) est inconnue. Elle est remplacée par $\mathbf{x}_{l,i}$, aussi obtenu par quadrature numérique comme

$$\mathbf{x}_{l,i} = \mathbf{x}_l + h \sum_{j=1}^{q} a_{i,j} \mathbf{f}(\mathbf{x}_{l,j}, t_{l,j}). \tag{12.51}$$

Les $q(q+2)$ coefficients $a_{i,j}$, $b_i$ et $\tau_i$ d'une méthode de Runge-Kutta à $q$ étapes doivent être choisis pour assurer la stabilité et la plus grande précision possible. Ceci conduit à ce qui est appelé dans [3] *une jungle algébrique non linéaire, à laquelle la civilisation et l'ordre ont été apportés par le travail de pionnier de J.C. Butcher*.

Plusieurs ensembles d'équations de Runge-Kutta peuvent être obtenus pour un même ordre. Les formules classiques RK($k$) sont *explicites*, avec $a_{i,j} = 0$ pour $i \leqslant j$, ce qui rend triviale la résolution de (12.51). Pour $q = 1$ et $\tau_1 = 0$, on obtient RK(1), qui est la méthode d'Euler explicite. Un des choix possibles pour RK(2) est

$$\begin{aligned}
\mathbf{k}_1 &= h \cdot \mathbf{f}(\mathbf{x}_l, t_l), && (12.52)\\
\mathbf{k}_2 &= h \cdot \mathbf{f}\left(\mathbf{x}_l + \frac{\mathbf{k}_1}{2}, t_l + \frac{h}{2}\right), && (12.53)\\
\mathbf{x}_{l+1} &= \mathbf{x}_l + \mathbf{k}_2, && (12.54)\\
t_{l+1} &= t_l + h, && (12.55)
\end{aligned}$$

avec une erreur de méthode locale en $\mathrm{o}(h^2)$, génériquement en $\mathrm{O}(h^3)$. La figure 12.2 illustre la procédure pour un état scalaire $x$.

Bien que des calculs soient conduits au point milieu $t_l + h/2$, ça reste une méthode à un pas, puisque $\mathbf{x}_{l+1}$ est calculé en fonction de $\mathbf{x}_l$.

La méthode de Runge-Kutta la plus utilisée est RK(4), qui peut s'écrire

$$\begin{aligned}
\mathbf{k}_1 &= h \cdot \mathbf{f}(\mathbf{x}_l, t_l), && (12.56)\\
\mathbf{k}_2 &= h \cdot \mathbf{f}\left(\mathbf{x}_l + \frac{\mathbf{k}_1}{2}, t_l + \frac{h}{2}\right), && (12.57)\\
\mathbf{k}_3 &= h \cdot \mathbf{f}\left(\mathbf{x}_l + \frac{\mathbf{k}_2}{2}, t_l + \frac{h}{2}\right), && (12.58)\\
\mathbf{k}_4 &= h \cdot \mathbf{f}(\mathbf{x}_l + \mathbf{k}_3, t_l + h), && (12.59)\\
\mathbf{x}_{l+1} &= \mathbf{x}_l + \frac{\mathbf{k}_1}{6} + \frac{\mathbf{k}_2}{3} + \frac{\mathbf{k}_3}{3} + \frac{\mathbf{k}_4}{6}, && (12.60)\\
t_{l+1} &= t_l + h, && (12.61)
\end{aligned}$$

avec une erreur de méthode locale en $\mathrm{o}(h^4)$, génériquement en $\mathrm{O}(h^5)$. La dérivée première de l'état par rapport à $t$ est maintenant évaluée une fois en $t_l$, une fois en $t_{l+1}$ et deux fois en $t_l + h/2$. RK(4) n'en reste pas moins une méthode à un pas.

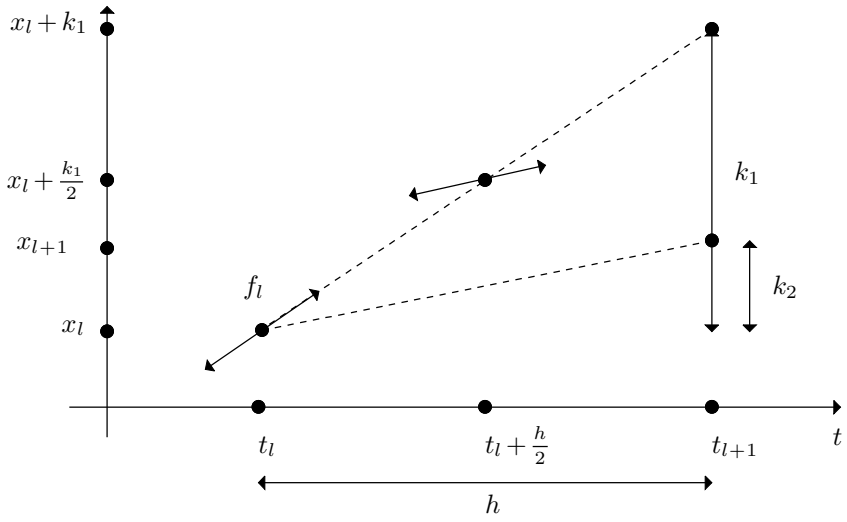


**Fig. 12.2** Un pas de RK(2)

*Remarque 12.8.* Comme les autres méthodes de Runge-Kutta explicites, RK(4) peut démarrer par elle-même. Si on lui fournit la condition initiale $\mathbf{x}_0$, elle calcule $\mathbf{x}_1$, qui est la condition initiale pour le calcul de $\mathbf{x}_2$, et ainsi de suite. Le prix à payer pour cette propriété bien utile est qu'aucune des quatre évaluations numériques de $\mathbf{f}$ menées pour calculer $\mathbf{x}_{l+1}$ ne peut être réutilisée dans le calcul de $\mathbf{x}_{l+2}$. Ceci peut être un inconvénient majeur par rapport aux méthodes à plusieurs pas de la section 12.2.2.3, si l'efficacité calculatoire est un facteur important. D'un autre côté, il est beaucoup plus facile d'adapter la taille du pas (voir la section 12.2.4), et les méthodes de Runge-Kutta sont plus robustes quand la solution présente des quasi-discontinuités. On peut comparer les méthodes de Runge-Kutta à des remorqueurs de haute mer, qui peuvent faire sortir les paquebots de croisière des ports encombrés et venir à leur secours quand la mer se démonte. □

Des méthodes de Runge-Kutta *implicites* [34, 3] ont aussi été développées. Ce sont les seules méthodes de Runge-Kutta qui restent utilisables sur les EDO raides, voir la section 12.2.5. Chacun de leurs pas requiert la résolution d'un ensemble d'équations implicites, et est donc plus complexe pour un ordre donné. Sur la base de [223] et [260], MATLAB a mis en œuvre sa propre version d'une méthode de Runge-Kutta implicite dans `ode23s`, où le calcul de $\mathbf{x}_{l+1}$ passe par la résolution d'un système d'équations linéaires [216].

*Remarque 12.9.* Il a en fait été montré dans [162], et discuté plus avant dans [163], que des relations de récurrence permettent souvent d'utiliser des développements de Taylor en faisant *moins* de calculs qu'avec une méthode de Runge-Kutta du même

ordre. L'approche par série de Taylor est effectivement utilisée (pour des valeurs de $k$ très grandes) dans le contexte de l'*intégration garantie*, où des ensembles contenant les solutions mathématiques exactes d'EDO sont calculés numériquement [16, 148, 149]. □

#### 12.2.2.3 Méthodes linéaires à plusieurs pas

Les méthodes linéaires à plusieurs pas expriment $\mathbf{x}_{l+1}$ comme une combinaison linéaire de valeurs de $\mathbf{x}$ et $\dot{\mathbf{x}}$, sous la forme générale

$$\mathbf{x}_{l+1} = \sum_{i=0}^{n_\mathrm{a}-1} a_i \mathbf{x}_{l-i} + h \sum_{j=j_0}^{n_\mathrm{b}+j_0-1} b_j \mathbf{f}_{l-j}. \tag{12.62}$$

Elles diffèrent par les valeurs données au nombre $n_\mathrm{a}$ des coefficients $a_i$, au nombre $n_\mathrm{b}$ des coefficients $b_j$ et à la valeur initiale $j_0$ de l'indice de la seconde somme de (12.62). Dès que $n_\mathrm{a} > 1$ ou que $n_\mathrm{b} > 1 - j_0$, (12.62) correspond à une *méthode à plusieurs pas*, parce que $\mathbf{x}_{l+1}$ est calculé à partir de plusieurs valeurs passées de $\mathbf{x}$ (ou de $\dot{\mathbf{x}}$, lui-même calculé à partir de valeurs de $\mathbf{x}$).

*Remarque 12.10.* L'équation (12.62) n'utilise que des évaluations conduites avec la taille de pas constante $h = t_{i+1} - t_i$. Les évaluations de $\mathbf{f}$ utilisées pour calculer $\mathbf{x}_{l+1}$ peuvent donc être réutilisées pour calculer $\mathbf{x}_{l+2}$, ce qui est un avantage considérable par rapport aux méthodes de Runge-Kutta. Il y a cependant des désavantages :

- adapter la taille du pas devient significativement plus compliqué qu'avec les méthodes de Runge-Kutta ;
- les méthodes à plusieurs pas ne peuvent démarrer par elles-mêmes ; si on ne leur fournit que la condition initiale $\mathbf{x}_0$, elles sont incapables de calculer $\mathbf{x}_1$, et doivent recevoir l'aide d'une méthode à un pas pour calculer assez de valeurs de $\mathbf{x}$ et $\dot{\mathbf{x}}$ pour initialiser la récurrence (12.62).

Si les méthodes de Runge-Kutta sont des remorqueurs de haute mer, alors les méthodes à plusieurs pas sont des paquebots de croisière, qui ne peuvent quitter le port des conditions initiales par eux-mêmes. Les méthodes à plusieurs pas peuvent aussi échouer plus tard, si les fonctions impliquées ne sont pas assez lisses, et les méthodes de Runge-Kutta (ou d'autres méthodes à un pas) peuvent alors être appelées à leur secours. □

Nous considérons trois familles de méthodes linéaires à plusieurs pas, à savoir les méthodes d'Adams-Bashforth, d'Adams-Moulton et de Gear. Le membre d'ordre $k$ de n'importe laquelle de ces familles a une erreur de méthode locale en $\mathrm{o}(h^k)$, génériquement en $\mathrm{O}(h^{k+1})$.

Les *méthodes d'Adams-Bashforth* sont explicites. Dans la méthode d'ordre $k$ AB($k$), $n_\mathrm{a} = 1$, $a_0 = 1$, $j_0 = 0$ et $n_\mathrm{b} = k$, de sorte que

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h \sum_{j=0}^{k-1} b_j \mathbf{f}_{l-j}. \tag{12.63}$$

Quand $k = 1$, il y a un seul coefficient $b_0 = 1$ et AB(1) est la méthode d'Euler explicite

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h\mathbf{f}_l. \tag{12.64}$$

C'est donc une méthode à un pas. AB(2) est telle que

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{2}(3\mathbf{f}_l - \mathbf{f}_{l-1}). \tag{12.65}$$

C'est donc une méthode à plusieurs pas, qui ne peut pas démarrer seule, tout comme AB(3), où

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{12}(23\mathbf{f}_l - 16\mathbf{f}_{l-1} + 5\mathbf{f}_{l-2}), \tag{12.66}$$

et AB(4), où

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{24}(55\mathbf{f}_l - 59\mathbf{f}_{l-1} + 37\mathbf{f}_{l-2} - 9\mathbf{f}_{l-3}). \tag{12.67}$$

Dans la *méthode d'Adams-Moulton* d'ordre $k$ AM($k$), $n_\mathrm{a} = 1$, $a_0 = 1$, $j_0 = -1$ et $n_\mathrm{b} = k$, de sorte que

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h\sum_{j=-1}^{k-2} b_j\mathbf{f}_{l-j}. \tag{12.68}$$

Puisque $j$ prend la valeur $-1$, toutes les méthodes d'Adams-Moulton sont implicites. Quand $k = 1$, il y a un seul coefficient $b_{-1} = 1$ et AM(1) est la méthode d'Euler implicite

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h\mathbf{f}_{l+1}. \tag{12.69}$$

AM(2) est une méthode trapézoïdale (voir NC(1) dans la section 6.2.1.1)

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{2}\left(\mathbf{f}_{l+1} + \mathbf{f}_l\right). \tag{12.70}$$

AM(3) satisfait

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{12}\left(5\mathbf{f}_{l+1} + 8\mathbf{f}_l - \mathbf{f}_{l-1}\right), \tag{12.71}$$

et est une méthode à plusieurs pas, comme AM(4), qui est telle que

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{24}\left(9\mathbf{f}_{l+1} + 19\mathbf{f}_l - 5\mathbf{f}_{l-1} + \mathbf{f}_{l-2}\right). \tag{12.72}$$

Finalement, dans la *méthode de Gear* d'ordre $k$ G($k$), $n_\mathrm{a} = k$, $n_\mathrm{b} = 1$ et $j_0 = -1$, de sorte que toutes les méthodes de Gear sont implicites et

$$\mathbf{x}_{l+1} = \sum_{i=0}^{k-1} a_i\mathbf{x}_{l-i} + hb\mathbf{f}_{l+1}. \tag{12.73}$$

Les méthodes de Gear sont aussi appelées *méthodes BDF*, où BDF est l'acronyme de *backward-differentiation formulas*, car ces formules sont employées pour calculer leurs coefficients. G($k$) = BDF($k$) est telle que

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m \mathbf{x}_{l+1} - h\mathbf{f}_{l+1} = \mathbf{0}, \tag{12.74}$$

avec

$$\nabla \mathbf{x}_{l+1} = \mathbf{x}_{l+1} - \mathbf{x}_l, \tag{12.75}$$

$$\nabla^2 \mathbf{x}_{l+1} = \nabla(\nabla \mathbf{x}_{l+1}) = \mathbf{x}_{l+1} - 2\mathbf{x}_l + \mathbf{x}_{l-1}, \tag{12.76}$$

et ainsi de suite. G(1) est la méthode d'Euler implicite

$$\mathbf{x}_{l+1} = \mathbf{x}_l + h\mathbf{f}_{l+1}. \tag{12.77}$$

G(2) satisfait

$$\mathbf{x}_{l+1} = \frac{1}{3}(4\mathbf{x}_l - \mathbf{x}_{l-1} + 2h\mathbf{f}_{l+1}). \tag{12.78}$$

G(3) est telle que

$$\mathbf{x}_{l+1} = \frac{1}{11}(18\mathbf{x}_l - 9\mathbf{x}_{l-1} + 2\mathbf{x}_{l-2} + 6h\mathbf{f}_{l+1}), \tag{12.79}$$

et G(4) telle que

$$\mathbf{x}_{l+1} = \frac{1}{25}(48\mathbf{x}_l - 36\mathbf{x}_{l-1} + 16\mathbf{x}_{l-2} - 3\mathbf{x}_{l-3} + 12h\mathbf{f}_{l+1}). \tag{12.80}$$

Une variante de (12.74),

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m \mathbf{x}_{l+1} - h\mathbf{f}_{l+1} - \kappa \sum_{j=1}^{k} \frac{1}{j}(\mathbf{x}_{l+1} - \mathbf{x}_{l+1}^0) = \mathbf{0}, \tag{12.81}$$

a été étudiée dans [129] sous le nom de NDF, l'acronyme de *numerical differentiation formulas*, dans le but d'améliorer les propriétés de stabilité des méthodes BDF d'ordre élevé. Dans (12.81), $\kappa$ est un paramètre scalaire et $\mathbf{x}_{l+1}^0$ une prédiction (grossière) de $\mathbf{x}_{l+1}$ utilisée comme valeur initiale pour résoudre (12.81) en $\mathbf{x}_{l+1}$ par une méthode de Newton simplifiée (méthode des cordes). Sur la base des NDF, MATLAB a développé sa propre méthodologie dans `ode15s` [216, 7], avec un ordre qui varie de $k = 1$ à $k = 5$.

*Remarque 12.11.* Il est trivial de changer l'ordre $k$ d'une méthode linéaire à plusieurs pas quand on le juge utile, puisque cela se résume au calcul d'une autre combinaison linéaire de vecteurs déjà calculés $\mathbf{x}_{l-i}$ ou $\mathbf{f}_{l-i}$. On peut en tirer parti pour permettre à Adams-Bashforth de démarrer par elle-même, en utilisant AB(1) pour calculer $\mathbf{x}_1$ à partir de $\mathbf{x}_0$, puis AB(2) pour calculer $\mathbf{x}_2$ à partir de $\mathbf{x}_1$ et $\mathbf{x}_0$, et ainsi de suite jusqu'à ce que l'ordre désiré ait été atteint. □

#### 12.2.2.4 Problèmes pratiques avec les méthodes implicites

Avec les méthodes implicites, $\mathbf{x}_{l+1}$ est solution d'un système d'équations qu'on peut écrire sous la forme

$$\mathbf{g}(\mathbf{x}_{l+1}) = \mathbf{0}. \tag{12.82}$$

Ce système est en général non linéaire, mais devient linéaire quand (12.26) est satisfaite. Quand c'est possible, comme dans l'exemple 12.4, c'est une bonne habitude de mettre (12.82) sous forme explicite où $\mathbf{x}_{l+1}$ est exprimée comme une fonction de quantités calculées précédemment. Quand ceci ne peut pas être fait, on utilise souvent la méthode de Newton de la section 7.4.2 (ou une version simplifiée de celle-ci comme la méthode des cordes), qui requiert l'évaluation numérique ou formelle de la jacobienne de $\mathbf{g}(\cdot)$. Quand $\mathbf{g}(\mathbf{x})$ est linéaire en $\mathbf{x}$, sa jacobienne ne dépend pas de $\mathbf{x}$ et peut être calculée une fois pour toutes, ce qui représente une simplification considérable. Dans la fonction MATLAB `ode15s`, les jacobiennes sont évaluées le plus rarement possible.

Pour éviter une résolution numérique répétée et potentiellement coûteuse de (12.82) à chaque pas, on peut alterner

— une *prédiction*, où une méthode explicite (Adams-Bashforth, par exemple) est utilisée pour obtenir une première approximation $\mathbf{x}^1_{l+1}$ de $\mathbf{x}_{l+1}$, et
— une *correction*, où une méthode implicite (Adams-Moulton, par exemple), est utilisée pour obtenir une seconde approximation $\mathbf{x}^2_{l+1}$ de $\mathbf{x}_{l+1}$, avec $\mathbf{x}_{l+1}$ remplacé par $\mathbf{x}^1_{l+1}$ lors de l'évaluation de $\mathbf{f}_{l+1}$.

Comme la méthode de prédiction-correction résultante est explicite, certains avantages des méthodes implicites sont perdus, cependant.

*Exemple 12.5.* La prédition peut être conduite avec AB(2)

$$\mathbf{x}^1_{l+1} = \mathbf{x}_l + \frac{h}{2}(3\mathbf{f}_l - \mathbf{f}_{l-1}), \tag{12.83}$$

et la correction avec AM(2), où $\mathbf{x}_{l+1}$ sur le côté droit est remplacé par $\mathbf{x}^1_{l+1}$

$$\mathbf{x}^2_{l+1} = \mathbf{x}_l + \frac{h}{2}\left[\mathbf{f}\left(\mathbf{x}^1_{l+1}, t_{l+1}\right) + \mathbf{f}_l\right]. \tag{12.84}$$

□

*Remarque 12.12.* L'influence de la prédiction sur l'erreur de méthode locale finale est moindre que celle de la correction, de sorte qu'on peut utiliser un prédicteur d'ordre $k-1$ avec un correcteur d'ordre $k$. Quand la prédiction est effectuée par AB(1), c'est à dire par la méthode d'Euler explicite,

$$\mathbf{x}^1_{l+1} = \mathbf{x}_l + h\mathbf{f}_l, \tag{12.85}$$

et la correction par AM(2), c'est à dire par la méthode trapézoïdale implicite,

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \frac{h}{2}\left[\mathbf{f}\left(\mathbf{x}^1_{l+1}, t_{l+1}\right) + \mathbf{f}_l\right], \tag{12.86}$$

le résultat est la *méthode de Heun*, une méthode de Runge-Kutta explicite du second ordre tout comme RK(2) présentée en section 12.2.2.2. □

Adams-Bashforth-Moulton est utilisée dans la fonction MATLAB `ode113` (de $k = 1$ à $k = 13$). Cette fonction exploite la facilité avec laquelle on peut changer l'ordre d'une méthode à plusieurs pas.

### *12.2.3 Mise à l'échelle*

Pourvu que des bornes supérieures $\bar{x}_i$ puissent être obtenues sur les valeurs absolues des variables d'état $x_i$ $(i = 1, \cdots, n)$, on peut transformer l'équation d'état initiale (12.1) en

$$\dot{\mathbf{q}}(t) = \mathbf{g}(\mathbf{q}(t), t), \tag{12.87}$$

avec

$$q_i = \frac{x_i}{\bar{x}_i}, \quad i = 1, \cdots, n. \tag{12.88}$$

C'était plus ou moins obligatoire au temps des calculateurs analogiques, pour éviter de saturer les amplificateurs opérationnels. La gamme de magnitudes offerte par les nombres à virgule flottante a rendu cette pratique moins cruciale, mais elle peut encore se révéler très utile.

### *12.2.4 Choisir la taille du pas*

Quand la taille $h$ du pas augmente, la charge de calcul décroît mais l'erreur de méthode augmente. Il faut donc trouver un compromis [211]. Considérons l'influence de $h$ sur la stabilité avant d'aborder l'évaluation des erreurs et le réglage de la taille du pas.

#### 12.2.4.1 Influence de la taille du pas sur la stabilité

Considérons une équation d'état linéaire stationnaire

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}, \tag{12.89}$$

et supposons qu'il existe une matrice inversible $\mathbf{T}$ telle que

$$\mathbf{A} = \mathbf{T}\boldsymbol{\Lambda}\mathbf{T}^{-1}, \tag{12.90}$$

où $\boldsymbol{\Lambda}$ est une matrice diagonale avec les valeurs propres $\lambda_i$ $(i = 1, \cdots, n)$ de $\mathbf{A}$ sur sa diagonale. Supposons de plus (12.89) asymptotiquement stable, de sorte que ces valeurs propres (qui peuvent être complexes) ont des parties réelles stricte-

ment négatives. Le changement de coordonnées $\mathbf{q} = \mathbf{T}^{-1}\mathbf{x}$ conduit à la nouvelle représentation d'état

$$\dot{\mathbf{q}} = \mathbf{T}^{-1}\mathbf{A}\mathbf{T}\mathbf{q} = \boldsymbol{\Lambda}\mathbf{q}. \tag{12.91}$$

La $i$-ème composante du nouveau vecteur d'état $\mathbf{q}$ satisfait

$$\dot{q}_i = \lambda_i q_i. \tag{12.92}$$

Ceci motive l'étude de la stabilité des méthodes numériques pour la résolution des problèmes aux valeurs initiales sur *le problème test de Dahlquist* [47]

$$\dot{x} = \lambda x, \quad x(0) = 1, \tag{12.93}$$

où $\lambda$ est une constante *complexe* avec une partie réelle strictement négative, et non plus la constante réelle considérée dans l'exemple 12.4. La taille $h$ du pas doit être telle que le schéma d'intégration soit stable pour chacun des problèmes test obtenus en remplaçant $\lambda$ par une des valeurs propres de $\mathbf{A}$.

Voyons maintenant comment conduire cette étude de stabilité [142] ; cette partie peut être sautée par le lecteur qui ne s'intéresse qu'à ses résultats.

*Méthodes à un pas*

Quand on les applique au problème test (12.93), les méthodes à un pas calculent

$$x_{l+1} = R(z)x_l, \tag{12.94}$$

où $z = h\lambda$ est un argument complexe.

*Remarque 12.13.* La solution exacte de ce problème test satisfait

$$x_{l+1} = e^{h\lambda}x_l = e^z x_l, \tag{12.95}$$

de sorte que $R(z)$ est une approximation de $e^z$. Puisque $z$ est sans dimension, l'unité dans laquelle $t$ est exprimé n'a pas de conséquence, pourvu que ce soit la même pour $h$ et pour $\lambda^{-1}$. □

Pour la méthode d'Euler explicite,

$$\begin{aligned} x_{l+1} &= x_l + h\lambda x_l, \\ &= (1+z)x_l, \end{aligned} \tag{12.96}$$

de sorte que $R(z) = 1+z$. Si l'on utilise un développement de Taylor à l'ordre $k$,

$$x_{l+1} = x_l + h\lambda x_l + \cdots + \frac{1}{k!}(h\lambda)^k x_l, \tag{12.97}$$

alors $R(z)$ est le polynôme

$$R(z) = 1 + z + \cdots + \frac{1}{k!} z^k. \tag{12.98}$$

Il en est de même pour toute méthode de Runge-Kutta explicite d'ordre $k$, puisqu'elle a été conçue pour ça.

*Exemple 12.6.* Quand on applique la méthode de Heun au problème test, (12.85) devient

$$\begin{aligned} x_{l+1}^1 &= x_l + h\lambda x_l \\ &= (1+z)x_l, \end{aligned} \tag{12.99}$$

et (12.86) se traduit par

$$\begin{aligned} x_{l+1} &= x_l + \frac{h}{2}(\lambda x_{l+1}^1 + \lambda x_l) \\ &= x_l + \frac{z}{2}(1+z)x_l + \frac{z}{2}x_l \\ &= \left(1 + z + \frac{1}{2}z^2\right) x_l. \end{aligned} \tag{12.100}$$

Ceci n'est pas surprenant, car la méthode de Heun est une méthode de Runge-Kutta explicite du second ordre. □

Pour les méthodes implicites à un pas, $R(z)$ sera une fonction rationnelle. Pour AM(1), la méthode d'Euler implicite, on a

$$\begin{aligned} x_{l+1} &= x_l + h\lambda x_{l+1} \\ &= \frac{1}{1-z} x_l \end{aligned} \tag{12.101}$$

Pour AM(2), la méthode trapézoïdale, il vient

$$\begin{aligned} x_{l+1} &= x_l + \frac{h}{2}(\lambda x_{l+1} + \lambda x_l) \\ &= \frac{1 + \frac{z}{2}}{1 - \frac{z}{2}} x_l. \end{aligned} \tag{12.102}$$

Pour chacune de ces méthodes, la solution du problème test de Dahlquist sera (absolument) stable si et seulement si $z$ est tel que $|R(z)| \leqslant 1$ [142].

Pour la méthode d'Euler explicite, ceci signifie que $h\lambda$ doit être *à l'intérieur* du disque de rayon unité centré en $-1$, tandis que pour la méthode d'Euler implicite, $h\lambda$ doit être *à l'extérieur* du disque de rayon unité centré en $+1$. Comme $h$ est toujours réel et positif et comme $\lambda$ est supposé ici avoir une partie réelle négative, la méthode d'Euler implicite est donc toujours stable sur le problème test. L'intersection du disque de stabilité de la méthode d'Euler explicite avec l'axe réel est l'intervalle $[-2, 0]$, ce qui est cohérent avec les résultats de l'exemple 12.4.

AM(2) se révèle absolument stable pour tout $z$ à partie réelle négative (c'est à dire pour tout $\lambda$ tel que le problème test est stable) et instable pour tous les autres $z$.

La figure 12.3 présente des courbes de niveaux des régions où doit se trouver $z = h\lambda$ pour que les méthodes de Runge-Kutta explicites d'ordre $k = 1$ à 6 soient absolument stables. La surface de la région de stabilité absolue se révèle croître quand l'ordre de la méthode augmente. Le script MATLAB utilisé pour tracer les courbes de niveaux pour RK(4) est en section 12.4.1.

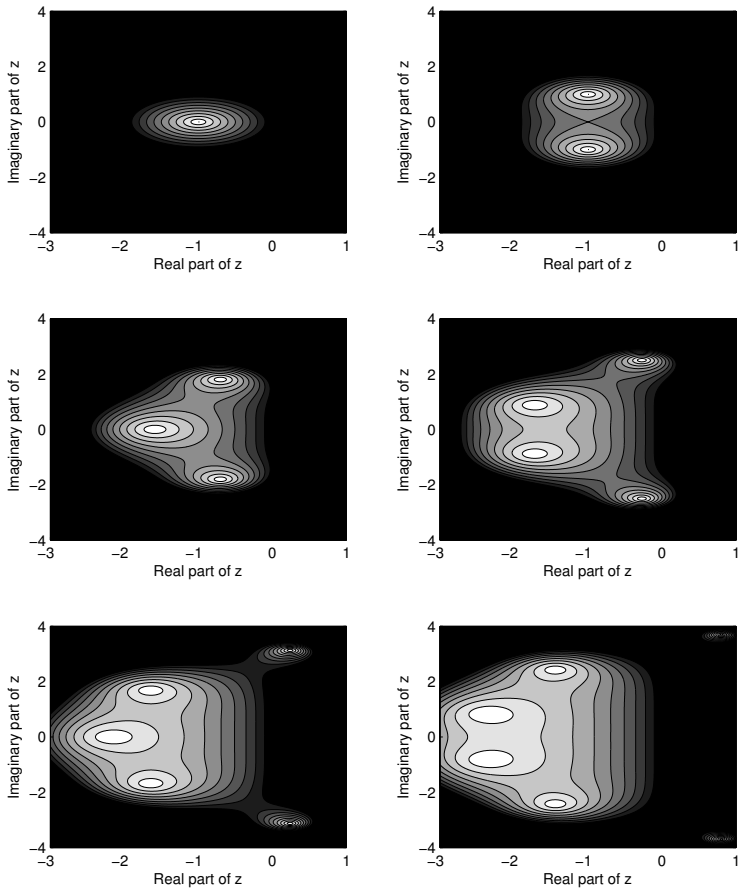


**Fig. 12.3** Courbes de niveaux des régions de stabilité absolue des méthodes de Runge-Kutta explicites sur le problème test de Dahlquist, de RK(1) (en haut à gauche) à RK(6) (en bas à droite) ; les régions en noir sont instables

*Méthodes linéaires à plusieurs pas*

Pour le problème test (12.93), la récurrence vectorielle non linéaire (12.62) qui contient toutes les méthodes linéaires à plusieurs pas devient scalaire et linéaire. On peut la réécrire

$$\sum_{j=0}^{r} \alpha_j x_{l+j} = h \sum_{j=0}^{r} \beta_j \lambda x_{l+j}, \tag{12.103}$$

ou encore

$$\sum_{j=0}^{r} (\alpha_j - z\beta_j) x_{l+j} = 0, \tag{12.104}$$

avec $r$ le nombre de pas de la méthode.

Cette récurrence linéaire est absolument stable si et seulement si les $r$ racines de son *polynôme caractéristique*

$$P_z(\zeta) = \sum_{j=0}^{r} (\alpha_j - z\beta_j) \zeta^j \tag{12.105}$$

appartiennent au disque de rayon unité centré sur l'origine. (Plus précisément, les racines simples doivent appartenir au disque fermé et les racines multiples au disque ouvert.)

*Exemple 12.7.* Bien que AB(1), AM(1) and AM(2) soient des méthodes à un pas, on peut les étudier via leur polynôme caractéristique, avec les même résultats que précédemment. Le polynôme caractéristique de AB(1) est

$$P_z(\zeta) = \zeta - (1+z). \tag{12.106}$$

Sa seule racine est $\zeta_{\text{ab1}} = 1+z$, de sorte que la région de stabilité absolue est

$$\mathbb{S} = \{z : |1+z| \leqslant 1\}. \tag{12.107}$$

Le polynôme caractéristique de AM(1) est

$$P_z(\zeta) = (1+z)\zeta - 1. \tag{12.108}$$

Sa seule racine est $\zeta_{\text{am1}} = 1/(1+z)$, de sorte que la région de stabilité absolue est

$$\mathbb{S} = \left\{ z : \left| \frac{1}{1+z} \right| \leqslant 1 \right\} = \{z : |1-z| \geqslant 1\}. \tag{12.109}$$

Le polynôme caractéristique de AM(2) est

$$P_z(\zeta) = \left(1 - \frac{1}{2}z\right)\zeta - \left(1 + \frac{1}{2}z\right). \tag{12.110}$$

Sa seule racine est

$$\zeta_{\text{am2}} = \frac{1+\frac{1}{2}z}{1-\frac{1}{2}z}, \tag{12.111}$$

et $|\zeta_{\text{am2}}| \leqslant 1 \Leftrightarrow \text{Re}(z) \leqslant 0$. □

Quand le degré $r$ du polynôme caractéristique est supérieur à un, la situation devient plus compliquée, car $P_z(\zeta)$ à maintenant plusieurs racines. Si $z$ est sur la frontière de la région de stabilité, alors une racine $\zeta_1$ au moins de $P_z(\zeta)$ doit avoir un module égal à un. Elle satisfait donc

$$\zeta_1 = e^{i\theta}, \tag{12.112}$$

pour un $\theta \in [0, 2\pi]$.

Puisque $z$ agit de façon affine dans (12.105), $P_z(\zeta)$ peut se réécrire

$$P_z(\zeta) = \rho(\zeta) - z\sigma(\zeta). \tag{12.113}$$

$P_z(\zeta_1) = 0$ se traduit alors par

$$\rho(e^{i\theta}) - z\sigma(e^{i\theta}) = 0, \tag{12.114}$$

de sorte que

$$z(\theta) = \frac{\rho(e^{i\theta})}{\sigma(e^{i\theta})}. \tag{12.115}$$

En traçant $z(\theta)$ pour $\theta \in [0, 2\pi]$, on obtient toutes les valeurs de $h\lambda$ qui *peuvent* être sur la frontière de la région de stabilité absolue, et ce tracé est appelé *lieu frontière*. Pour la méthode d'Euler explicite, par exemple, $\rho(\zeta) = \zeta - 1$ et $\sigma(\zeta) = 1$, de sorte que $z(\theta) = e^{i\theta} - 1$ et le lieu frontière est un cercle de rayon unité centrée en $-1$, comme il convient. Quand le lieu frontière ne se croise pas, il sépare la région de stabilité absolue du reste du plan complexe, et il est facile de décider qui est qui, en tirant un point $z$ arbitraire dans l'une des deux régions et en évaluant les racines de $P_z(\zeta)$ en ce point. Quand le lieu frontière se croise, il définit plus de deux régions dans le plan complexe, et chacune de ces régions doit être échantillonnée, en général pour constater que la stabilité absolue n'est atteinte que dans l'une d'entre elles au plus.

Dans une famille donnée de méthodes à plusieurs pas, le domaine de stabilité absolue tend à rétrécir quand on augmente l'ordre, contrairement à ce que nous avons observé pour les méthodes de Runge-Kutta explicites. Le domaine de stabilité absolue de G(6) est si petit que cette méthode est rarement utilisée, et il n'existe aucun $z$ tel que G($k$) soit stable pour $k > 6$ [92]. La détérioration du domaine de stabilité absolu est plus rapide avec les méthodes d'Adams-Bashforth qu'avec les méthodes d'Adams-Moulton. La section 12.4.1 détaille le script MATLAB utilisé pour visualiser les régions de stabilité absolue pour AB(1) et AB(2).

#### 12.2.4.2 Évaluer l'erreur de méthode locale en faisant varier la taille du pas

Quand $\mathbf{x}$ bouge lentement, on peut prendre un pas de taille $h$ plus grande que quand il varie vite, de sorte qu'un $h$ constant peut ne pas être approprié. Pour éviter des calculs inutiles (voire nuisibles), une couche est donc ajoutée au code du solveur d'EDO, en charge d'évaluer l'erreur de méthode locale pour adapter $h$ quand c'est nécessaire. Commençons par traiter le cas des méthodes à un pas, en utilisant l'approche la plus ancienne qui procède par variation de la taille du pas.

Considérons RK(4), par exemple. Soit $h_1$ la taille de pas courante, et $\mathbf{x}_l$ l'état initial du pas de simulation courant. Puisque l'erreur de méthode locale de RK(4) est génériquement en $\mathrm{O}(h^5)$, l'état après deux pas est tel que

$$\mathbf{x}(t_l + 2h_1) = \mathbf{r}_1 + h_1^5\mathbf{c}_1 + h_1^5\mathbf{c}_2 + \mathrm{O}(h^6), \tag{12.116}$$

où $\mathbf{r}_1$ est le résultat fourni par RK(4), et où

$$\mathbf{c}_1 = \frac{\mathbf{x}^{(5)}(t_l)}{5!} \tag{12.117}$$

et

$$\mathbf{c}_2 = \frac{\mathbf{x}^{(5)}(t_l + h_1)}{5!}. \tag{12.118}$$

Calculons maintenant $\mathbf{x}(t_l + 2h_1)$ à partir du même état initial $\mathbf{x}_l$ mais en un seul pas de taille $h_2 = 2h_1$, pour obtenir

$$\mathbf{x}(t_l + h_2) = \mathbf{r}_2 + h_2^5\mathbf{c}_1 + \mathrm{O}(h^6), \tag{12.119}$$

où $\mathbf{r}_2$ est le résultat fourni par RK(4).

Avec l'approximation $\mathbf{c}_1 = \mathbf{c}_2 = \mathbf{c}$ (qui deviendrait exacte si la solution était un polynôme d'ordre cinq au plus), et en négligeant tous les termes d'ordre supérieur à cinq, nous obtenons

$$\mathbf{r}_2 - \mathbf{r}_1 \approx (2h_1^5 - h_2^5)\mathbf{c} = -30h_1^5\mathbf{c}. \tag{12.120}$$

Une estimée de l'erreur de méthode locale pour la taille de pas $h_1$ est ainsi

$$2h_1^5\mathbf{c} \approx \frac{\mathbf{r}_1 - \mathbf{r}_2}{15}, \tag{12.121}$$

tandis qu'une estimée de l'erreur de méthode locale pour la taille de pas $h_2$ est

$$h_2^5\mathbf{c} = (2h_1)^5\mathbf{c} = 32h_1^5\mathbf{c} \approx \frac{32}{30}(\mathbf{r}_1 - \mathbf{r}_2). \tag{12.122}$$

Comme on s'y attendait, l'erreur de méthode locale croît donc considérablement quand on double la taille du pas. Puisqu'une estimée de cette erreur est maintenant disponible, on pourrait la soustraire de $\mathbf{r}_1$ pour améliorer la qualité du résultat, mais l'estimée de l'erreur de méthode locale serait alors perdue.

#### 12.2.4.3 Évaluer l'erreur de méthode locale en faisant varier l'ordre

Au lieu de faire varier la taille de leur pas pour évaluer leur erreur de méthode locale, les méthodes modernes tendent à faire varier leur ordre, d'une façon qui diminue le volume de calcul requis. C'est l'idée derrière les *méthodes de Runge-Kutta imbriquées*, comme les méthodes de *Runge-Kutta-Fehlberg* [227]. RKF45, par exemple [152], utilise une méthode RK(5), telle que

$$\mathbf{x}_{l+1}^5 = \mathbf{x}_l + \sum_{i=1}^{6} c_{5,i}\mathbf{k}_i + \mathrm{O}(h^6). \tag{12.123}$$

Les coefficients de cette méthode sont choisis pour assurer qu'une méthode RK(4) soit imbriquée, telle que

$$\mathbf{x}_{l+1}^4 = \mathbf{x}_l + \sum_{i=1}^{6} c_{4,i}\mathbf{k}_i + \mathrm{O}(h^5). \tag{12.124}$$

On prend alors comme estimée de l'erreur de méthode locale $\|\mathbf{x}_{l+1}^5 - \mathbf{x}_{l+1}^4\|$.

MATLAB fournit deux méthodes de Runge-Kutta explicites imbriquées, à savoir `ode23`, qui utilise une paire de formules d'ordres deux et trois due à Bogacki et Shampine [19] et `ode45`, qui utilise une paire de formules d'ordres quatre et cinq due à Dormand et Prince [58]. Dormand et Prince ont proposé plusieurs autres méthodes de Runge-Kutta imbriquées [58, 187, 59], jusqu'à une paire de formules d'ordres sept et huit. Shampine a développé un solveur MATLAB utilisant une autre paire de formules d'ordres sept et huit avec un contrôle d'erreur fort (disponible sur son site web), et a comparé ses performances à celles d'`ode45` dans [212].

L'erreur de méthode locale des méthodes à plusieurs pas peut de même être évaluée par comparaison des résultats à différents ordres. C'est facile, puisqu'il n'est pas nécessaire de procéder à de nouvelles évaluations de $\mathbf{f}$.

#### 12.2.4.4 Adapter la taille du pas

Le solveur d'EDO essaye de sélectionner un pas aussi grand que possible, tout en tenant compte de la précision requise. Il doit aussi tenir compte des contraintes de stabilité de la méthode utilisée (pour les EDO non linéaires, une règle empirique est d'assurer que $z = h\lambda$ soit dans la région de stabilité absolue pour chaque valeur propre $\lambda$ de la jacobienne de $\mathbf{f}$ au point de linéarisation).

Si l'estimée de l'erreur de méthode locale sur $\mathbf{x}_{l+1}$ se révèle supérieure à une tolérance spécifiée par l'utilisateur, alors $\mathbf{x}_{l+1}$ est refusé et la connaissance de l'ordre de la méthode est utilisée pour choisir une réduction de la taille du pas qui devrait rendre acceptable l'erreur de méthode locale. Il faut par ailleurs rester réaliste dans ses exigences de précision, pour deux raisons :

— augmenter la précision implique de réduire les tailles de pas et donc d'augmenter le volume des calculs,

— quand les tailles de pas deviennent trop petites, les erreurs d'arrondi dominent les erreurs de méthode et la qualité des résultats se dégrade.

*Remarque 12.14.* Le contrôle de la taille du pas sur la base d'estimées grossières comme celles décrites en sections 12.2.4.2 et 12.2.4.3 peut être pris en défaut. [209] donne un exemple pour lequel un code de qualité production fit *croître* l'erreur quand on fit *décroître* la tolérance sur celle-ci. [221] présente une classe de problèmes très simples pour lesquels le solveur `ode45` avec ses options par défaut donne des résultats fondamentalement incorrects parce que sa taille de pas est souvent située en dehors de la région de stabilité. □

S'il est facile de changer la taille du pas pour une méthode à un pas, cela devient beaucoup plus compliqué avec une méthode à plusieurs pas car plusieurs valeurs passées de $\mathbf{x}$ doivent être mises à jour quand on modifie $h$. Soit $\mathbf{Z}(h)$ la matrice obtenue en plaçant côte à côte toutes les valeurs passées du vecteur d'état utilisées pour le calcul de $\mathbf{x}_{l+1}$ :

$$\mathbf{Z}(h) = [\mathbf{x}_l, \mathbf{x}_{l-1}, \ldots, \mathbf{x}_{l-k}]. \tag{12.125}$$

Pour remplacer la taille de pas $h_{\text{old}}$ par $h_{\text{new}}$, il faut en principe remplacer $\mathbf{Z}(h_{\text{old}})$ par $\mathbf{Z}(h_{\text{new}})$, ce qui semble requérir la connaissance de valeurs passées de l'état.

Des approximations par différences finies telles que

$$\dot{\mathbf{x}}(t_l) \approx \frac{\mathbf{x}_l - \mathbf{x}_{l-1}}{h} \tag{12.126}$$

et

$$\ddot{\mathbf{x}}(t_l) \approx \frac{\mathbf{x}_l - 2\mathbf{x}_{l-1} + \mathbf{x}_{l-2}}{h^2} \tag{12.127}$$

permettent d'évaluer numériquement

$$\mathbf{X} = [\mathbf{x}(t_l), \dot{\mathbf{x}}(t_l), \cdots, \mathbf{x}^{(k)}(t_l)], \tag{12.128}$$

et de définir une transformation linéaire bijective $\mathbf{T}(h)$ telle que

$$\mathbf{X} \approx \mathbf{Z}(h)\mathbf{T}(h). \tag{12.129}$$

Pour $k = 2$, et (12.126) et (12.127), on obtient par exemple

$$\mathbf{T}(h) = \begin{bmatrix} 1 & \frac{1}{h} & \frac{1}{h^2} \\ 0 & -\frac{1}{h} & -\frac{2}{h^2} \\ 0 & 0 & \frac{1}{h^2} \end{bmatrix}. \tag{12.130}$$

Comme la valeur mathématique de $\mathbf{X}$ ne dépend pas de $h$, nous avons

$$\mathbf{Z}(h_{\text{new}}) \approx \mathbf{Z}(h_{\text{old}})\mathbf{T}(h_{\text{old}})\mathbf{T}^{-1}(h_{\text{new}}), \tag{12.131}$$

ce qui permet d'adapter la taille du pas sans redémarrage via une méthode à un pas.

Puisque

$$\mathbf{T}(h) = \mathbf{N}\mathbf{D}(h), \tag{12.132}$$

où $\mathbf{N}$ est une matrice inversible constante et

$$\mathbf{D}(h) = \mathrm{diag}(1, \frac{1}{h}, \frac{1}{h^2}, \cdots), \tag{12.133}$$

le calcul de $\mathbf{Z}(h_{\mathrm{new}})$ par (12.131) peut être simplifié en celui de

$$\mathbf{Z}(h_{\mathrm{new}}) \approx \mathbf{Z}(h_{\mathrm{old}}) \cdot \mathbf{N} \cdot \mathrm{diag}(1, \alpha, \alpha^2, \cdots, \alpha^k) \cdot \mathbf{N}^{-1}, \tag{12.134}$$

où $\alpha = h_{\mathrm{new}}/h_{\mathrm{old}}$. Une simplification supplémentaire est rendue possible par l'utilisation du *vecteur de Nordsieck*, qui contient les coefficients du développement de Taylor de $x$ au voisinage de $t_l$ jusqu'à l'ordre $k$

$$\mathbf{n}(t_l, h) = \left[ x(t_l), h\dot{x}(t_l), \cdots, \frac{h^k}{k!} x^{(k)}(t_l) \right]^{\mathrm{T}}, \tag{12.135}$$

avec $x$ une composante quelconque de $\mathbf{x}$. On peut montrer que

$$\mathbf{n}(t_l, h) \approx \mathbf{M}\mathbf{v}(t_l, h), \tag{12.136}$$

où $\mathbf{M}$ est une matrice inversible, connue et *constante*, et où

$$\mathbf{v}(t_l, h) = [x(t_l), x(t_l - h), \cdots, x(t_l - kh)]^{\mathrm{T}}. \tag{12.137}$$

Puisque

$$\mathbf{n}(t_l, h_{\mathrm{new}}) = \mathrm{diag}(1, \alpha, \alpha^2, \cdots, \alpha^k) \cdot \mathbf{n}(t_l, h_{\mathrm{old}}), \tag{12.138}$$

il est facile d'obtenir une valeur approximative de $\mathbf{v}(t_l, h_{\mathrm{new}})$ comme $\mathbf{M}^{-1}\mathbf{n}(t_l, h_{\mathrm{new}})$, sans changer l'ordre de l'approximation.

#### 12.2.4.5 Évaluer l'erreur de méthode globale

Ce qui est évalué en sections 12.2.4.2 et 12.2.4.3, c'est l'erreur de méthode locale sur un pas, et non l'erreur de méthode globale à la fin d'une simulation qui peut impliquer de nombreux pas. Le nombre total de pas est approximativement

$$N = \frac{t_{\mathrm{f}} - t_0}{h}, \tag{12.139}$$

avec $h$ la taille de pas moyenne. Si l'erreur globale d'une méthode d'ordre $k$ était égale à $N$ fois son erreur locale, elle serait en $N\mathrm{O}(h^{k+1}) = \mathrm{O}(h^k)$, et c'est pourquoi on dit que la méthode est d'ordre $k$. La situation est en fait plus compliquée car l'erreur de méthode globale dépend crucialement du degré de stabilité de l'EDO. Soit $\mathbf{s}(t_N, \mathbf{x}_0, t_0)$ la vraie valeur d'une solution $\mathbf{x}(t_N)$ à la fin d'une simulation qui est partie de $\mathbf{x}_0$ à $t_0$ et soit $\widehat{\mathbf{x}}_N$ l'estimée de cette solution fournie par la méthode

d'intégration. Pour tout $\mathbf{v} \in \mathbb{R}^n$, la norme de l'erreur globale satisfait

$$\begin{aligned} \|\mathbf{s}(t_N, \mathbf{x}_0, t_0) - \widehat{\mathbf{x}}_N\| &= \|\mathbf{s}(t_N, \mathbf{x}_0, t_0) - \widehat{\mathbf{x}}_N + \mathbf{v} - \mathbf{v}\| \\ &\leqslant \|\mathbf{v} - \widehat{\mathbf{x}}_N\| + \|\mathbf{s}(t_N, \mathbf{x}_0, t_0) - \mathbf{v}\|. \end{aligned} \tag{12.140}$$

Prenons $\mathbf{v} = \mathbf{s}(t_N, \widehat{\mathbf{x}}_{N-1}, t_{N-1})$. Le premier terme du membre de droite de (12.140) est alors la norme de la dernière erreur locale, tandis que le second est la norme de la différence entre des solutions exactes évaluées au même instant mais pour des conditions initiales différentes. Quand l'EDO est instable, les erreurs inévitables sur les conditions initiales sont amplifiées jusqu'à ce que la solution numérique devienne inutilisable. Quand au contraire l'EDO est si stable que l'effet des erreurs sur ses conditions initiales disparaît rapidement, l'erreur globale peut être bien moindre que ce qu'on aurait pu craindre.

Une façon simple et grossière d'évaluer l'erreur de méthode globale pour un problème aux valeurs initiales donné est de le résoudre une deuxième fois avec une tolérance réduite et d'estimer l'erreur sur la première série de résultats en les comparant avec ceux de la seconde série [211]. On devrait au moins vérifier que les résultats d'une simulation complète ne varient pas drastiquement quand l'utilisateur réduit la tolérance. Si cette stratégie peut permettre de détecter des erreurs inacceptables, elle est cependant incapable de prouver que les résultats sont corrects.

On peut préférer caractériser l'erreur globale en fournissant des vecteurs d'intervalles numériques $[\mathbf{x}_{\min}(t), \mathbf{x}_{\max}(t)]$ auxquels la solution mathématique $\mathbf{x}(t)$ appartient pour tout $t$ d'intérêt, avec toutes les sources d'erreur prises en compte (y compris les erreurs dues aux arrondis). Ceci est accompli dans le contexte de l'*intégration garantie*, voir [16, 148, 149]. Le défit est de contenir la croissance des intervalles d'incertitude, qui peuvent devenir excessivement pessimistes quand $t$ croît.

#### 12.2.4.6 Méthode de Bulirsch-Stoer

La méthode de Bulirsch-Stoer [227] est encore une application de l'extrapolation de Richardson. Une *méthode d'intégration à point milieu modifiée* est utilisée pour calculer $\mathbf{x}(t_l + H)$ à partir de $\mathbf{x}(t_l)$ en $N$ sous-pas de taille $h$, comme suit :

$$\begin{aligned} \mathbf{z}_0 &= \mathbf{x}(t_l), \\ \mathbf{z}_1 &= \mathbf{z}_0 + h\mathbf{f}(\mathbf{z}_0, t_l), \\ \mathbf{z}_{i+1} &= \mathbf{z}_{i-1} + 2h\mathbf{f}(\mathbf{z}_i, t_l + ih), \quad i = 1, \cdots, N-1, \\ \mathbf{x}(t_l + H) = \mathbf{x}(t_l + Nh) &\approx \frac{1}{2}[\mathbf{z}_N + \mathbf{z}_{N-1} + h\mathbf{f}(\mathbf{z}_N, t_l + Nh)]. \end{aligned}$$

Un avantage crucial de cette stratégie est que le terme d'erreur de méthode dans le calcul de $\mathbf{x}(t_l + H)$ est strictement pair (c'est une fonction de $h^2$). L'ordre de l'erreur de méthode croît donc de deux avec chaque pas d'extrapolation de Richardson, tout comme pour l'intégration de Romberg (voir la section 6.2.2). On obtient ainsi

très rapidement des résultats très précis, pourvu que la solution de l'EDO soit suffisamment lisse. Ceci rend la méthode de Bulirsch-Stoer particulièrement appropriée quand on veut une grande précision ou quand l'évaluation de $\mathbf{f}(\mathbf{x},t)$ est coûteuse.

### *12.2.5 EDO raides*

Considérons le modèle d'état linéaire stationnaire

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}, \tag{12.141}$$

et supposons le asymptotiquement stable, c'est à dire tel que toutes les valeurs propres de $\mathbf{A}$ aient des parties réelles strictement négatives. Ce modèle est *raide* si les valeurs absolues de ces parties réelles sont telles que le rapport de la plus grande à la plus petite soit très grand. De façon similaire, le modèle non linéaire

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \tag{12.142}$$

est raide si sa dynamique comporte des composantes très rapides et très lentes. C'est souvent le cas pour les réactions chimiques, par exemple, où les constantes de vitesse peuvent différer de plusieurs ordres de grandeur.

Les EDO raides sont particulièrement difficiles à simuler précisément, car les composantes rapides requièrent un pas de petite taille tandis que les composantes lentes requièrent un horizon d'intégration long. Même quand les composantes rapides deviennent négligeables dans la solution, de sorte qu'on pourrait rêver d'augmenter la taille du pas, les méthodes d'intégration explicites continueront d'exiger un pas de petite taille pour leur stabilité. En conséquence, la résolution d'une EDO raide avec une méthode pour les problèmes non raides, telle que les fonctions `ode23` ou `ode45` de MATLAB, peut se révéler beaucoup trop lente pour être acceptable en pratique. Les méthodes implicites, dont les méthodes de Runge-Kutta implicites telles que `ode23s` et les méthodes de Gear et leurs variantes telles que `ode15s`, peuvent alors nous sauver la mise [213]. Les méthodes de prédiction-correction telle que `ode113` ne méritent pas d'être appelées implicites et sont à éviter pour les EDO raides.

### *12.2.6 Équations algébro-différentielles*

Les équations algébro-différentielles (ou EAD) peuvent d'écrire

$$\mathbf{r}(\dot{\mathbf{q}}(t),\mathbf{q}(t),t) = \mathbf{0}. \tag{12.143}$$

Un cas particulier important est quand on peut les écrire comme une EDO sous forme d'état couplée avec des contraintes algébriques :

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{z},t), \tag{12.144}$$
$$\mathbf{0} = \mathbf{g}(\mathbf{x},\mathbf{z},t). \tag{12.145}$$

Les perturbations singulières sont un grand fournisseur de tels systèmes d'équations.

#### 12.2.6.1 Perturbations singulières

Supposons que l'état d'un système puisse être partitionné en une partie lente $\mathbf{x}$ et une partie rapide $\mathbf{z}$, telles que

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{z},t,\varepsilon), \tag{12.146}$$
$$\mathbf{x}(t_0) = \mathbf{x}_0(\varepsilon), \tag{12.147}$$
$$\varepsilon\dot{\mathbf{z}} = \mathbf{g}(\mathbf{x},\mathbf{z},t,\varepsilon), \tag{12.148}$$
$$\mathbf{z}(t_0) = \mathbf{z}_0(\varepsilon), \tag{12.149}$$

avec $\varepsilon$ un paramètre positif traité comme un petit terme de perturbation. Plus $\varepsilon$ est petit et plus le système d'EDO devient raide. A la limite, quand $\varepsilon$ devient nul, (12.148) devient une équation algébrique

$$\mathbf{g}(\mathbf{x},\mathbf{z},t,0) = \mathbf{0}, \tag{12.150}$$

et on obtient une EAD. La perturbation est dite singulière parce que la dimension de l'espace d'état change quand $\varepsilon$ s'annule.

Il est parfois possible, comme dans l'exemple qui suit, de résoudre explicitement (12.150) en exprimant $\mathbf{z}$ comme une fonction de $\mathbf{x}$ et $t$, et de reporter l'expression formelle résultante dans (12.146) pour obtenir une EDO sous forme d'état d'ordre réduit, avec la condition initiale $\mathbf{x}(t_0) = \mathbf{x}_0(0)$.

*Exemple 12.8. Réaction enzyme-substrat*

Considérons la réaction biochimique

$$E + S \rightleftarrows C \rightarrow E + P, \tag{12.151}$$

où $E$, $S$, $C$ et $P$ sont respectivement l'enzyme, le substrat, le complexe enzyme-substrat et le produit. On suppose en général que cette réaction suit les équations

$$[\dot{E}] = -k_1[E][S] + k_{-1}[C] + k_2[C], \tag{12.152}$$
$$[\dot{S}] = -k_1[E][S] + k_{-1}[C], \tag{12.153}$$
$$[\dot{C}] = k_1[E][S] - k_{-1}[C] - k_2[C], \tag{12.154}$$
$$[\dot{P}] = k_2[C], \tag{12.155}$$

avec les conditions initiales

$$
\begin{aligned}
[E](t_0) &= E_0, && (12.156)\\
[S](t_0) &= S_0, && (12.157)\\
[C](t_0) &= 0, && (12.158)\\
[P](t_0) &= 0. && (12.159)
\end{aligned}
$$

Sommons (12.152) et (12.154) pour prouver que $[\dot{E}]+[\dot{C}] \equiv 0$, et éliminons (12.152) en remplaçant $[E]$ par $E_0 - [C]$ dans (12.153) et (12.154) pour obtenir le modèle réduit

$$
\begin{aligned}
[\dot{S}] &= -k_1(E_0-[C])[S]+k_{-1}[C], && (12.160)\\
[\dot{C}] &= k_1(E_0-[C])[S]-(k_{-1}+k_2)[C], && (12.161)\\
[S](t_0) &= S_0, && (12.162)\\
[C](t_0) &= 0. && (12.163)
\end{aligned}
$$

L'hypothèse d'*état quasi-stationnaire* [208] revient à supposer que, après un court transitoire et avant que $[S]$ ne soit épuisé, le taux de production de $P$ reste approximativement constant. L'équation (12.155) implique alors que $[C]$ reste aussi approximativement constant, ce qui transforme l'EDO en EAD

$$
\begin{aligned}
[\dot{S}] &= -k_1(E_0-[C])[S]+k_{-1}[C], && (12.164)\\
0 &= k_1(E_0-[C])[S]-(k_{-1}+k_2)[C]. && (12.165)
\end{aligned}
$$

La situation est suffisamment simple pour permettre d'exprimer $[C]$ en fonction de $[S]$ et des constantes cinétiques

$$\mathbf{p} = (k_1, k_{-1}, k_2)^{\mathrm{T}}, \tag{12.166}$$

sous la forme

$$[C] = \frac{E_0[S]}{K_m + [S]}, \tag{12.167}$$

avec

$$K_m = \frac{k_{-1}+k_2}{k_1}. \tag{12.168}$$

$[C]$ peut alors être remplacé dans (12.164) par son expression (12.167) pour obtenir une EDO où $[\dot{S}]$ est exprimée en fonction de $[S]$, $E_0$ et $\mathbf{p}$. □

Des extensions de l'hypothèse d'état quasi-stationnaire à des modèles plus généraux sont présentées dans [61] et [27]. Quand on ne dispose pas d'une solution explicite de l'équation algébrique, on peut utiliser la différentiation répétée pour transformer une EAD en EDO, voir la section 12.2.6.2. Une autre option est d'essayer une approche par différences finies, voir la section 12.3.3.

#### 12.2.6.2 Différentiation répétée

En dérivant formellement (12.145) par rapport à $t$ autant de fois que nécessaire et en remplaçant tout $\dot{x}_i$ ainsi créé par son expression tirée de (12.144), on peut obtenir une EDO, comme illustré par l'exemple qui suit.

*Exemple 12.9.* Considérons à nouveau l'exemple 12.8 et l'EAD (12.164, 12.165), mais supposons maintenant ne pas disposer de la solution explicite de (12.165) en $[C]$. Dérivons (12.165) par rapport à $t$, pour obtenir

$$k_1(E_0-[C])[\dot{S}]-k_1[S][\dot{C}]-(k_{-1}+k_2)[\dot{C}]=0, \tag{12.169}$$

et donc

$$[\dot{C}]=\frac{k_1(E_0-[C])}{k_{-1}+k_2+k_1[S]}[\dot{S}], \tag{12.170}$$

où $[\dot{S}]$ est donné par (12.164) et le dénominateur ne peut pas s'annuler. L'EAD a ainsi été transformée en l'EDO

$$[\dot{S}] = -k_1(E_0-[C])[S]+k_{-1}[C], \tag{12.171}$$

$$[\dot{C}] = \frac{k_1(E_0-[C])}{k_{-1}+k_2+k_1[S]}\{-k_1(E_0-[C])[S]+k_{-1}[C]\}, \tag{12.172}$$

et les conditions initiales doivent satisfaire (12.165). □

L'*index différentiel* d'une EAD est le nombre de dérivations qu'il faut effectuer pour la transformer en EDO. Pour l'exemple 12.9, cet index est égal à un.

[181] contient un rappel utile des difficultés qu'on peut rencontrer quand on résout une EAD avec des outils dédiés aux EDO.

## 12.3 Problèmes aux limites

Ce qu'on connaît des conditions initiales ne les spécifie pas toujours de façon unique. Il faut alors des conditions aux limites supplémentaires. Quand certaines de ces conditions aux limites ne sont pas relatives à l'état initial, on obtient un *problème aux limites*. Dans le présent contexte d'EDO, un cas particulier important est le *problème aux deux bouts*, où les états initiaux et finaux sont partiellement spécifiés. Les problèmes aux limites se révèlent plus compliqués à résoudre que les problèmes aux valeurs initiales.

*Remarque 12.15.* De nombreuses méthodes de résolution de problèmes aux limites pour des EDO s'appliquent aussi *mutatis mutandis* aux équations aux dérivées partielles, de sorte que cette partie peut aussi servir d'introduction au chapitre 13. □

### *12.3.1 Un minuscule champ de bataille*

Considérons le champ de bataille à deux dimensions illustré par la figure 12.4.


**Fig. 12.4** Un champ de bataille 2D

Le canon à l'origine O $(x = y = 0)$ doit tirer sur une cible immobile située en $(x = x_{\text{cible}}, y = 0)$. Le module $v_0$ de la vélocité initiale de l'obus est fixé, et le canonnier ne peut choisir que l'angle de visée $\theta$ dans l'intervalle ouvert $\left(0, \frac{\pi}{2}\right)$. Si l'on néglige la traînée, l'altitude de l'obus avant l'impact est donnée par

$$y_{\text{obus}}(t) = (v_0 \sin\theta)(t - t_0) - \frac{g}{2}(t - t_0)^2, \tag{12.173}$$

où $g$ est l'accélération de la pesanteur et $t_0$ l'instant du tir. La distance horizontale couverte par l'obus avant l'impact est telle que

$$x_{\text{obus}}(t) = (v_0 \cos\theta)(t - t_0). \tag{12.174}$$

Le canonnier doit donc trouver $\theta$ tel qu'il existe $t > t_0$ pour lequel $x_{\text{obus}}(t) = x_{\text{cible}}$ et $y_{\text{obus}}(t) = 0$, ou de façon équivalente tel que

$$(v_0 \cos\theta)(t - t_0) = x_{\text{cible}}, \tag{12.175}$$

et

$$(v_0 \sin\theta)(t - t_0) = \frac{g}{2}(t - t_0)^2. \tag{12.176}$$

C'est un problème aux deux bouts, puisque nous avons des informations partielles sur les états initial et final de l'obus. Pour toute valeur numérique admissible de $\theta$, le calcul de la trajectoire de l'obus est un problème aux valeurs initiales dont la solution est unique, mais ceci n'implique pas que la solution du problème aux deux bouts soit unique, ni même qu'elle existe.

Cet exemple est si simple que le nombre des solutions est facile à calculer analytiquement. Résolvons (12.176) par rapport à $t - t_0$, et reportons le résultat dans (12.175) pour obtenir

$$x_{\text{cible}} = 2\sin(\theta)\cos(\theta)\frac{v_0^2}{g} = \sin(2\theta)\frac{v_0^2}{g}. \tag{12.177}$$

Pour que $\theta$ existe, il faut que $x_{\text{cible}}$ n'excède pas la portée maximale $v_0^2/g$ du canon. Pour tout $x_{\text{cible}}$ atteignable, il y a génériquement deux valeurs $\theta_1$ et $\theta_2$ de $\theta$ pour lequelles (12.177) est satisfaite, comme le sait tout joueur de pétanque. Ces valeurs sont symétriques par rapport à $\theta = \pi/4$, et la portée maximale est atteinte quand $\theta_1 = \theta_2 = \pi/4$. Suivant les conditions imposées sur l'état final, le nombre des solutions de ce problème aux limites peut donc être zéro, un ou deux.

Ne pas savoir a priori s'il existe une solution est une difficulté typique des problèmes aux limites. Nous supposons dans ce qui suit que le problème a au moins une solution.

### *12.3.2 Méthodes de tir*

Les méthodes de tir, appelées ainsi par analogie avec l'artillerie et l'exemple de la section 12.3.1, utilisent un vecteur $\mathbf{x}_0(\mathbf{p})$ satisfaisant ce qu'on connait des conditions initiales. Le vecteur de paramètres $\mathbf{p}$ représente les degrés de liberté restants dans les conditions initiales. Pour toute valeur numérique donnée de $\mathbf{p}$, on connaît la valeur numérique de $\mathbf{x}_0(\mathbf{p})$ de sorte que le calcul de la trajectoire de l'état devient un problème aux valeurs initiales, pour lequel on peut utiliser les méthodes de la section 12.2. Le vecteur $\mathbf{p}$ doit alors être réglé de façon à satisfaire les autres conditions aux limites. On peut, par exemple, minimiser

$$J(\mathbf{p}) = \|\widehat{\boldsymbol{\sigma}} - \boldsymbol{\sigma}(\mathbf{x}_0(\mathbf{p}))\|_2^2, \tag{12.178}$$

où $\widehat{\boldsymbol{\sigma}}$ est un vecteur de conditions aux limites désirées (par exemple des conditions terminales), et $\boldsymbol{\sigma}(\mathbf{x}_0(\mathbf{p}))$ est un vecteur de conditions aux limites réalisées. Voir le chapitre 9 pour des méthodes utilisables dans ce contexte.

On peut aussi calculer $\mathbf{p}$ en résolvant

$$\boldsymbol{\sigma}(\mathbf{x}_0(\mathbf{p})) = \widehat{\boldsymbol{\sigma}}, \tag{12.179}$$

voir les chapitres 3 et 7 pour des méthodes pour ce faire.

*Remarque 12.16.* Minimiser (12.178) ou résoudre (12.179) peut impliquer la résolution d'un grand nombre de problèmes aux valeurs initiales si l'équation d'état est non linéaire. □

*Remarque 12.17.* Les méthodes de tir ne sont utilisables que sur des EDO assez stables pour que leur solution numérique n'explose pas avant la fin de la résolution des problèmes aux valeurs initiales associés. □

### *12.3.3 Méthode des différences finies*

Nous supposons ici que l'EDO s'écrit

$$g(t, y, \dot{y}, \cdots, y^{(n)}) = 0, \tag{12.180}$$

et qu'il n'est pas possible (ou pas souhaitable) de la mettre sous forme d'état. Le principe de la méthode des différences finies (MDF) est alors le suivant.

- Discrétiser l'intervalle d'intérêt pour la variable indépendante $t$, en utilisant une grille de points $t_l$ régulièrement espacés. Si l'on veut calculer la solution approchée en $t_l$, $l = 1, \cdots, N$, s'assurer que la grille contient aussi tous les points additionnels éventuellement nécessaires pour la prise en compte des informations fournies par les conditions aux limites.
- Remplacer les dérivées $y^{(j)}$ dans (12.180) par des différences finies, en utilisant pas exemple les approximations par différences centrées

$$\dot{y}_l \approx \frac{Y_{l+1} - Y_{l-1}}{2h} \tag{12.181}$$

et

$$\ddot{y}_l \approx \frac{Y_{l+1} - 2Y_l + Y_{l-1}}{h^2}, \tag{12.182}$$

où $Y_l$ est la solution approchée de (12.180) au point de discrétisation indexé par $l$ et où

$$h = t_l - t_{l-1}. \tag{12.183}$$

- Écrire les équations résultantes en $l = 1, \cdots, N$, en tenant compte des informations fournies par les conditions aux limites là où c'est nécessaire, pour obtenir un système de $N$ équations scalaires en $N$ inconnues $Y_l$.
- Résoudre ce système, qui sera linéaire si l'EDO est linéaire (voir le chapitre 3). Quand l'EDO est non linéaire, la résolution sera le plus souvent itérative (voir le chapitre 7) et à base de linéarisation, de sorte que la résolution des systèmes d'équations linéaires jouent un rôle clé dans les deux cas. Comme les approximations par différences finies sont locales (elles n'impliquent qu'un petit nombre de points de la grille proches de ceux où l'on approxime la dérivée), les systèmes linéaires à résoudre sont creux, et souvent à diagonale dominante.

*Exemple 12.10.* Supposons que l'EDO non stationnaire

$$\ddot{y}(t) + a_1(t)\dot{y}(t) + a_2(t)y(t) = u(t) \tag{12.184}$$

doive satisfaire les conditions aux limites $y(t_0) = y_0$ et $y(t_\mathrm{f}) = y_\mathrm{f}$, avec $t_0$, $t_\mathrm{f}$, $y_0$ et $y_\mathrm{f}$ connus (de telles conditions sur les valeurs de la solution à la frontière du domaine sont appelées *conditions de Dirichlet*). Supposons aussi les coefficients $a_1(t)$, $a_2(t)$ et l'entrée $u(t)$ connus pour tout $t$ dans $[t_0, t_\mathrm{f}]$.

Plutôt que d'utiliser une méthode de tir pour trouver la bonne valeur de $\dot{y}(t_0)$, prenons la grille

$$t_l = t_0 + lh, \quad l = 0, \cdots, N+1, \quad \text{avec} \quad h = \frac{t_\mathrm{f} - t_0}{N+1}, \tag{12.185}$$

qui a $N$ points intérieurs (sans compter les points à la frontière $t_0$ et $t_\mathrm{f}$). Notons $Y_l$ la valeur approchée de $y(t_l)$ à calculer ($l = 1, \cdots, N$), avec $Y_0 = y_0$ et $Y_{N+1} = y_\mathrm{f}$. Reportons (12.181) et (12.182) dans (12.184) pour obtenir

$$\frac{Y_{l+1} - 2Y_l + Y_{l-1}}{h^2} + a_1(t_l)\frac{Y_{l+1} - Y_{l-1}}{2h} + a_2(t_l)Y(t_l) = u(t_l). \tag{12.186}$$

Réarrangeons (12.186) comme

$$a_l Y_{l-1} + b_l Y_l + c_l Y_{l+1} = h^2 u_l, \tag{12.187}$$

avec

$$\begin{aligned} a_l &= 1 - \frac{h}{2}a_1(t_l), \\ b_l &= h^2 a_2(t_l) - 2, \\ c_l &= 1 + \frac{h}{2}a_1(t_l), \\ u_l &= u(t_l). \end{aligned} \tag{12.188}$$

Écrivons (12.187) en $l = 1, 2, \cdots, N$, pour obtenir

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{12.189}$$

avec

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & \cdots & \cdots & 0 & a_N & b_N \end{bmatrix}, \tag{12.190}$$

$$\mathbf{x} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_{N-1} \\ Y_N \end{bmatrix} \quad \text{et} \quad \mathbf{b} = \begin{bmatrix} h^2u_1 - a_1y_0 \\ h^2u_2 \\ \vdots \\ h^2u_{N-1} \\ h^2u_N - c_Ny_\mathrm{f} \end{bmatrix}. \tag{12.191}$$

Comme **A** est tridiagonale, le calcul de **x** par résolution de (12.189) a une complexité très faible et peut être menée à bien rapidement, même quand $N$ est grand. De plus, cette approche peut être utilisée pour des EDO instables, contrairement aux méthodes de tir. □

*Remarque 12.18.* L'approche par différences finies peut aussi être utilisée pour résoudre des problèmes aux valeurs initiales ou des EAD. □

### *12.3.4 Méthodes par projection*

Parmi les méthodes par projection pour les problèmes aux limites, il y a les approches de collocation, de Ritz-Galerkin et des moindres carrés [192]. Les splines y jouent un rôle prééminent [23]. Soit $\Delta$ une partition de l'intervalle $\mathbb{I} = [t_0, t_\mathrm{f}]$ en $n$ sous-intervalles $[t_{i-1}, t_i]$, $i = 1, \cdots, n$, tels que

$$t_0 < t_1 < \cdots < t_n = t_\mathrm{f}. \tag{12.192}$$

Les splines sont des éléments de l'ensemble $S(r,k,\Delta)$ de toutes les fonctions polynomiales par morceaux qui sont $k$ fois continûment différentiables sur $[t_0, t_\mathrm{f}]$ et prennent la même valeur qu'un polynôme de degré au plus $r$ sur $[t_{i-1}, t_i]$, $i = 1, \cdots, n$. La dimension $N$ de $S(r,k,\Delta)$ est égale au nombre de paramètres scalaires (et donc au nombre d'équations) requis pour spécifier une fonction spline donnée dans $S(r,k,\Delta)$. Les splines cubiques de la section 5.3.2 appartiennent à $S(3,2,\Delta)$, mais de nombreux autres choix sont possibles. Les polynômes de Bernstein, au cœur de la conception de formes assistée par ordinateur [62], sont une alternative attractive, considérée dans [17].

L'exemple 12.10 sera utilisé pour illustrer le plus simplement possible les approches de collocation, de Ritz-Galerkin et des moindres carrés.

#### 12.3.4.1 Collocation

Pour l'exemple 12.10, les méthodes de collocation déterminent une solution approchée $y_N \in S(r,k,\Delta)$ telle que les $N$ équations suivantes soient satisfaites

$$\ddot{y}_N(x_i) + a_1(x_i)\dot{y}_N(x_i) + a_2(x_i)y_N(x_i) = u(x_i), \quad i = 1, \cdots, N-2, \tag{12.193}$$

$$y_N(t_0) = y_0 \quad \text{et} \quad y_N(t_\mathrm{f}) = y_\mathrm{f}. \tag{12.194}$$

Les $x_i$ où $y_N$ doit satisfaire l'EDO sont appelés *points de collocation*. Il est facile d'évaluer les dérivées de $y_N$ qui apparaissent dans (12.193), puisque $y_N(\cdot)$ est polynomiale sur chaque sous-intervalle. Pour $S(3,2,\Delta)$, il n'est pas nécessaire d'introduire des équations supplémentaires à cause des contraintes de différentiabilité, de sorte que $x_i = t_i$ et $N = n+1$.

[197] traite de la résolution de problèmes aux limites par collocation, y compris pour des problèmes non linéaires. [215] et [127] contiennent des informations sur le solveur MATLAB `bvp4c`.

#### 12.3.4.2 Méthodes de Ritz-Galerkin

L'histoire fascinante de la famille de méthodes de Ritz-Galerkin est comptée dans [66]. L'approche fut développée par Ritz dans un cadre théorique, et appliquée par Galerkin (qui l'attribua bien à Ritz) à nombre de problèmes d'ingénierie. Des figures illustrant le travail d'Euler suggèrent qu'il utilisa l'idée sans même prendre la peine de l'expliquer.

Considérons l'EDO

$$L_t(y) = u(t), \tag{12.195}$$

où $L(\cdot)$ est un opérateur différentiel linéaire, $L_t(y)$ est la valeur prise par $L(y)$ en $t$, et $u$ est une fonction d'entrée connue. Supposons que les conditions aux limites soient

$$y(t_j) = y_j, \quad j = 1, \cdots, m, \tag{12.196}$$

avec les $y_j$ connus. Pour tenir compte de (12.196), approximons $y(t)$ par une combinaison linéaire $\widehat{y}_N(t)$ de *fonctions de base* connues (des splines, par exemple)

$$\widehat{y}_N(t) = \sum_{j=1}^{N} x_j \phi_j(t) + \phi_0(t), \tag{12.197}$$

avec $\phi_0(\cdot)$ telle que

$$\phi_0(t_i) = y_i, \quad i = 1, \cdots, m, \tag{12.198}$$

et les autres fonctions de base $\phi_j(\cdot)$, $j = 1, \cdots, N$, telles que

$$\phi_j(t_i) = 0, \quad i = 1, \cdots, m. \tag{12.199}$$

La solution approchée peut alors s'écrire

$$\widehat{y}_N(t) = \boldsymbol{\Phi}^{\mathrm{T}}(t)\mathbf{x} + \phi_0(t), \tag{12.200}$$

avec

$$\boldsymbol{\Phi}^{\mathrm{T}}(t) = [\phi_1(t), \cdots, \phi_N(t)] \tag{12.201}$$

un vecteur ligne de fonctions de base connues et

$$\mathbf{x} = (x_1, \cdots, x_N)^{\mathrm{T}} \tag{12.202}$$

un vecteur colonne de coefficients constants à déterminer. La solution approchée $\widehat{y}_N$ appartient donc à un espace de *dimension finie*.

Les méthodes de Ritz-Galerkin recherchent alors $\mathbf{x}$ tel que

$$< L(\widehat{y}_N - \phi_0), \varphi_i >=< u - L(\phi_0), \varphi_i >, \quad i = 1, \cdots, N, \tag{12.203}$$

où $< \cdot, \cdot >$ est le produit scalaire dans l'espace fonctionnel et où les $\varphi_i$ sont des *fonctions de test* connues. Nous choisissons des fonctions de base et de test qui sont de carré intégrable sur $\mathbb{I}$, et prenons

$$< f_1, f_2 >= \int_{\mathbb{I}} f_1(\tau) f_2(\tau) \mathrm{d}\tau. \tag{12.204}$$

Comme

$$< L(\widehat{y}_n - \phi_0), \varphi_i > = < L\left(\boldsymbol{\Phi}^{\mathrm{T}} \mathbf{x}\right), \varphi_i > \tag{12.205}$$

est linéaire en $\mathbf{x}$, (12.203) se traduit par un système d'équations linéaires

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{12.206}$$

Les méthodes de Ritz-Galerkin utilisent en général des fonctions de test identiques aux fonctions de base, telles que

$$\phi_i \in S(r, k, \Delta), \quad \varphi_i = \phi_i, \quad i = 1, \cdots, N, \tag{12.207}$$

mais rien n'y oblige. La collocation correspond à $\varphi_i(t) = \delta(t - t_i)$, où $\delta(t - t_i)$ est la distribution de Dirac de masse unité en $t = t_i$, car on a alors

$$< f, \varphi_i >= \int_{\mathbb{I}} f(\tau) \delta(\tau - t_i) \mathrm{d}\tau = f(t_i) \tag{12.208}$$

pour tout $t_i$ dans $\mathbb{I}$.

*Exemple 12.11.* Considérons à nouveau l'exemple 12.10, où

$$L_t(y) = \ddot{y}(t) + a_1(t)\dot{y}(t) + a_2(t)y(t). \tag{12.209}$$

Prenons $\phi_0(\cdot)$ tel que

$$\phi_0(t_0) = y_0 \quad \text{et} \quad \phi_0(t_{\mathrm{f}}) = y_{\mathrm{f}}. \tag{12.210}$$

Par exemple

$$\phi_0(t) = \frac{y_{\mathrm{f}} - y_0}{t_{\mathrm{f}} - t_0}(t - t_0) + y_0. \tag{12.211}$$

L'équation (12.206) est satisfaite, avec

$$a_{i,j} = \int_{\mathbb{I}} [\ddot{\phi}_j(\tau) + a_1(\tau)\dot{\phi}_j(\tau) + a_2(\tau)\phi_j(\tau)]\phi_i(\tau)\mathrm{d}\tau \tag{12.212}$$

et

$$b_i = \int_{\mathbb{I}} [u(\tau) - \ddot{\phi}_0(\tau) - a_1(\tau)\dot{\phi}_0(\tau) - a_2(\tau)\phi_0(\tau)]\phi_i(\tau)\mathrm{d}\tau, \tag{12.213}$$

pour $i = 1, \cdots, N$ et $j = 1, \cdots, N$.

L'intégration par partie peut être utilisée pour faire décroître le nombre des dérivations requises dans (12.212) et (12.213). Puisque (12.199) se traduit par

$$\phi_i(t_0) = \phi_i(t_\mathrm{f}) = 0, \quad i = 1, \cdots, N, \tag{12.214}$$

Nous avons

$$\int_{\mathbb{I}} \ddot{\phi}_j(\tau)\phi_i(\tau)\mathrm{d}\tau = -\int_{\mathbb{I}} \dot{\phi}_j(\tau)\dot{\phi}_i(\tau)\mathrm{d}\tau, \tag{12.215}$$

$$-\int_{\mathbb{I}} \ddot{\phi}_0(\tau)\phi_i(\tau)\mathrm{d}\tau = \int_{\mathbb{I}} \dot{\phi}_0(\tau)\dot{\phi}_i(\tau)\mathrm{d}\tau. \tag{12.216}$$

□

Les intégrales définies impliquées sont souvent évaluées par quadrature de Gauss sur chacun des sous-intervalles générés par $\Delta$. Si le nombre total de points de quadrature était égal à la dimension de $\mathbf{x}$, Ritz-Galerkin équivaudrait à de la collocation en ces points, mais on utilise en général plus de points de quadrature [192].

La méthodologie de Ritz-Galerkin peut être étendue à des problèmes non linéaires.

#### 12.3.4.3 Moindres carrés

Si la solution approchée obtenue avec l'approche de Ritz-Galerkin satisfait les conditions aux limites par construction, elle ne satisfait pas, en général, l'équation différentielle (12.195), de sorte que le résidu

$$\varepsilon_{\mathbf{x}}(t) = L_t(\widehat{y}_N) - u(t) = L_t\left(\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{x}\right) + L_t(\phi_0) - u(t) \tag{12.217}$$

n'est pas identiquement nul sur $\mathbb{I}$. On peut donc essayer de minimiser

$$J(\mathbf{x}) = \int_{\mathbb{I}} \varepsilon_{\mathbf{x}}^2(\tau)\mathrm{d}\tau. \tag{12.218}$$

Puisque $\varepsilon_{\mathbf{x}}(\tau)$ est affine en $\mathbf{x}$, $J(\mathbf{x})$ est quadratique en $\mathbf{x}$ et la version à temps continu des moindres carrés linéaires peut être employée. La valeur optimale $\widehat{\mathbf{x}}$ de $\mathbf{x}$ satisfait donc l'équation normale

$$\mathbf{A}\widehat{\mathbf{x}} = \mathbf{b}, \tag{12.219}$$

avec

$$\mathbf{A} = \int_{\mathbb{I}} [L_\tau(\boldsymbol{\Phi})][L_\tau(\boldsymbol{\Phi})]^{\mathrm{T}}\mathrm{d}\tau \tag{12.220}$$

et

$$\mathbf{b} = \int_{\mathbb{I}} [L_\tau(\boldsymbol{\Phi})][u(\tau) - L_\tau(\phi_0)]\mathrm{d}\tau. \tag{12.221}$$

Voir [145] pour plus de détails (y compris un type plus général de condition aux limites et le traitement de systèmes d'EDO) et une comparaison sur des exemples

numériques avec les résultats obtenus par l'approche de Ritz-Galerkin. Une comparaison des trois approches par projection de la section 12.3.4 peut être trouvée dans [198] et [25].

## 12.4 Exemples MATLAB

### *12.4.1 Régions de stabilité absolue pour le test de Dahlquist*

La force brute et une grille sont utilisées pour caractériser la région de stabilité absolue de RK(4) avant d'exploiter la notion d'équation caractéristique pour tracer les frontières des régions de stabilité absolue de AB(1) et AB(2).

#### 12.4.1.1 RK(4)

Nous exploitons (12.98), qui implique pour RK(4) que

$$R(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4. \tag{12.222}$$

La région de stabilité absolue est l'ensemble des $z$ tels que $|R(z)| \leqslant 1$. Le script

```
clear all
[X,Y] = meshgrid(-3:0.05:1,-3:0.05:3);
Z = X + i*Y;
modR=abs(1+Z+Z.^2/2+Z.^3/6+Z.^4/24);
GoodR = ((1-modR)+abs(1-modR))/2;

% Tracé de surface en 3D
figure;
surf(X,Y,GoodR);
colormap(gray)
xlabel('Real part of z')
ylabel('Imaginary part of z')
zlabel('Margin of stability')

% Tracé de courbes de niveau remplies en 2D
figure;
contourf(X,Y,GoodR,15);
colormap(gray)
xlabel('Real part of z')
ylabel('Imaginary part of z')
```

produit les figures 12.5 et 12.6.

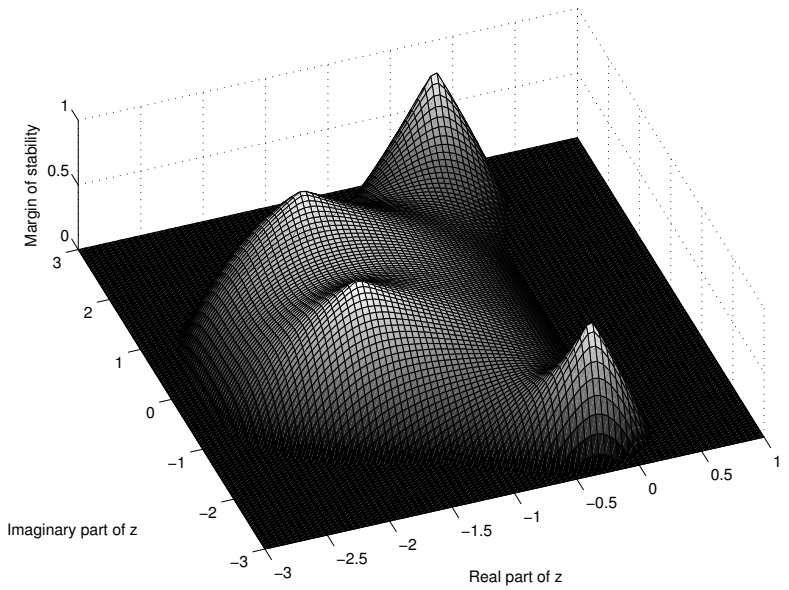


**Fig. 12.5** Visualisation 3D de la marge de stabilité de RK(4) pour le test de Dahlquist ; la région en noir est instable

#### 12.4.1.2 AB(1) et AB(2)

Tout point sur la frontière de la région de stabilité absolue de AB(1) doit être tel que le module de la racine de (12.106) soit égal à un. Ceci implique qu'il existe un $\theta$ tel que $\exp(\mathrm{i}\theta) = 1+z$, de sorte que

$$z = \exp(\mathrm{i}\theta) - 1. \tag{12.223}$$

AB(2) satisfait l'équation de récurrence (12.65), dont le polynôme caractéristique est

$$P_z(\zeta) = \zeta^2 - (1+\frac{3}{2}z)\zeta + \frac{z}{2}. \tag{12.224}$$

Pour que $\zeta = \exp(\mathrm{i}\theta)$ soit une racine de cette équation caractéristique, il faut que $z$ soit tel que

$$\exp(2\mathrm{i}\theta) - (1+\frac{3}{2}z)\exp(\mathrm{i}\theta) + \frac{z}{2} = 0, \tag{12.225}$$

ce qui implique que

$$z = \frac{\exp(2\mathrm{i}\theta) - \exp(\mathrm{i}\theta)}{1.5\exp(\mathrm{i}\theta) - 0.5}. \tag{12.226}$$

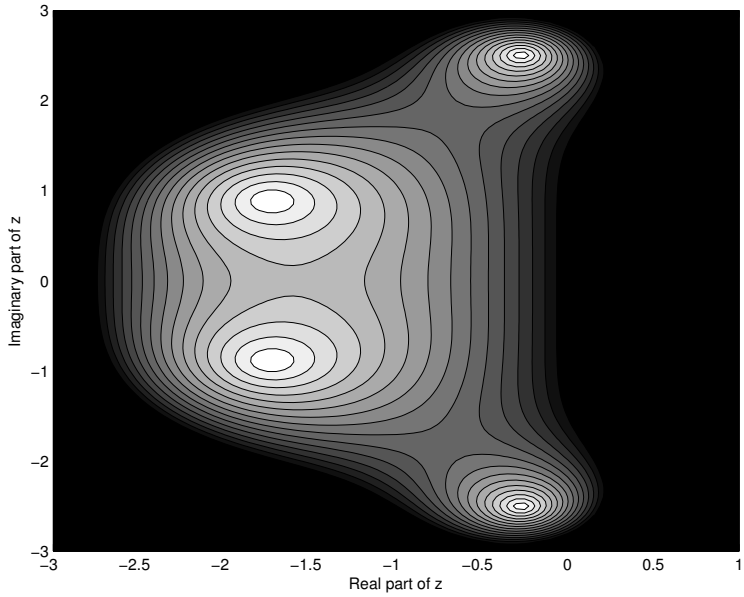


**Fig. 12.6** Courbes de niveau de la marge de stabilité de RK(4) pour le test de Dahlquist ; la région en noir est instable

Les équations (12.223) et (12.226) suggèrent le script suivant, utilisé pour produire la figure 12.7.

```
clear all
theta = 0:0.001:2*pi;
zeta = exp(i*theta);
hold on

% Tracé de courbes de niveau remplies pour AB(1)
boundaryAB1 = zeta - 1;
area(real(boundaryAB1), imag(boundaryAB1),...
'FaceColor',[0.5 0.5 0.5]);  % Gris
xlabel('Real part of z')
ylabel('Imaginary part of z')
grid on
axis equal

% Tracé de courbes de niveau remplies pour AB(2)
boundaryAB2 = (zeta.^2-zeta)./(1.5*zeta-0.5);
area(real(boundaryAB2), imag(boundaryAB2),...
    'FaceColor',[0 0 0]);  % Noir
```

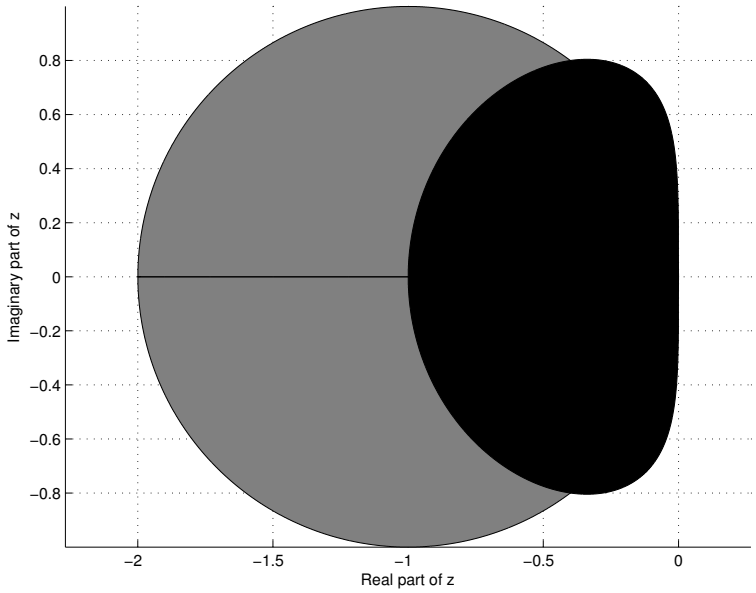


**Fig. 12.7** La région de stabilité absolue est en gris pour AB(1), en noir pour AB(2)

### *12.4.2 Influence de la raideur*

Un modèle simple de la propagation d'une boule de feu est

$$\dot{y} = y^2 - y^3, \quad y(0) = y_0, \tag{12.227}$$

où $y(t)$ est le diamètre de la boule à l'instant $t$. Ce diamètre croît de façon monotone de sa valeur initiale $y_0 < 1$ vers sa valeur asymptotique $y = 1$. Pour cette valeur asymptotique, la vitesse de consommation de l'oxygène à l'intérieur de la boule (proportionnel à $y^3$) est égale à celle de fourniture de l'oxygène à travers la surface de la boule (proportionnelle à $y^2$) et $\dot{y} = 0$. Plus $y_0$ est petit et plus la solution devient raide, ce qui rend cet exemple particulièrement adapté à l'illustration de l'influence de la raideur sur les performances des solveurs d'EDO [158]. Toutes les solutions seront calculées pour des temps allant de 0 à $2/y_0$.

Le script qui suit appelle `ode45`, un solveur pour EDO non raides, avec $y_0 = 0.1$ et une tolérance relative fixée à $10^{-4}$.

```
clear all
y0 = 0.1;
```

```
f = @(t,y) y^2 - y^3';
option = odeset('RelTol',1.e-4);
ode45(f,[0 2/y0],y0,option);
xlabel('Time')
ylabel('Diameter')
```

Il produit la figure 12.8 en environ 1.2 s. La solution est tracée au fur et à mesure de son calcul.

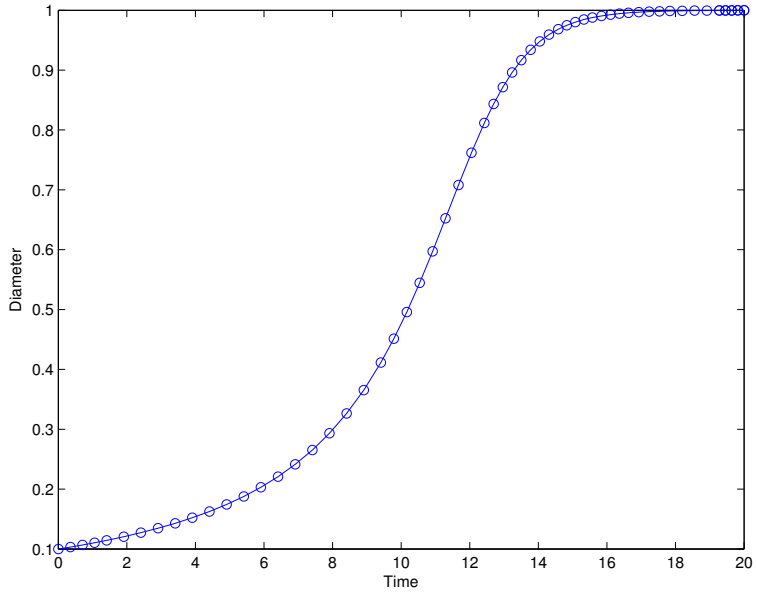


**Fig. 12.8** `ode45` sur l'évolution d'une boule de feu quand $y_0 = 0.1$

En remplaçant la seconde ligne de ce script par `y0 = 0.0001;` pour rendre le système plus raide, nous obtenons la figure 12.9 en environ 84.8 s. La progression après le saut devient *très* lente.

A la place de `ode45`, le script qui suit appelle `ode23s`, un solveur pour EDO raides, là aussi avec $y_0 = 0.0001$ et la même tolérance relative.

```
clear all
y0 = 0.0001;
f = @(t,y) y^2 - y^3';
option = odeset('RelTol',1.e-4);
ode23s(f,[0 2/y0],y0,option);
xlabel('Time')
ylabel('Diameter')
```

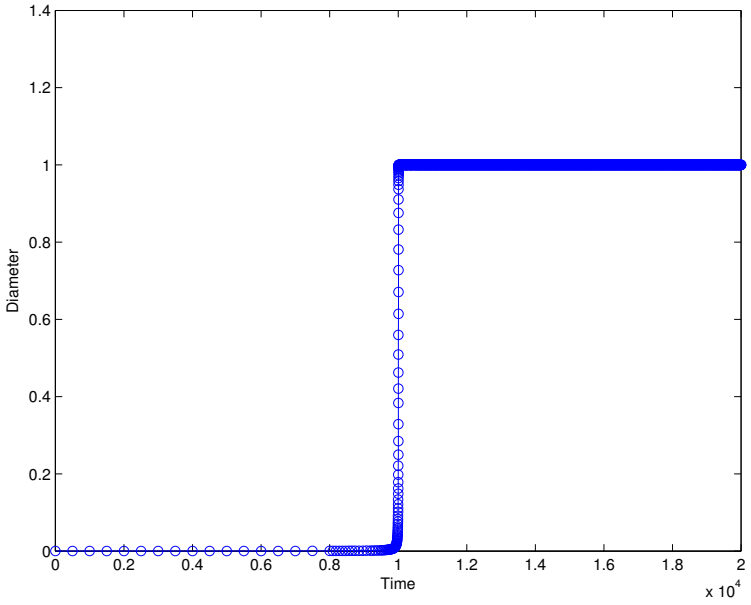


**Fig. 12.9** `ode45` sur l'évolution d'une boule de feu quand $y_0 = 0.0001$

Ce script produit la figure 12.10 en environ 2.8 s. Alors que `ode45` crawlait péniblement après le saut pour maintenir l'erreur de méthode locale sous contrôle, `ode23s` obtenait le même résultat en beaucoup moins d'évaluations de $\dot{y}$.

Si nous avions utilisé `ode15s`, un autre solveur pour EDO raides, la solution approchée aurait été obtenue en environ 4.4 s (pour la même tolérance relative). C'est plus lent qu'avec `ode23s`, mais encore beaucoup plus rapide qu'avec `ode45`. Ces résultats sont cohérents avec la documentation MATLAB, qui indique que `ode23s` *peut être plus efficace que* `ode15s` *pour des tolérances grossières* et *peut résoudre certains types de problèmes raides pour lesquels* `ode15s` *n'est pas efficace*. Il est si simple de basculer d'un solveur d'EDO à un autre qu'il ne faut pas hésiter à faire des essais sur le problème considéré pour un choix informé.

### *12.4.3 Simulation pour l'estimation de paramètres*

Considérons maintenant le modèle à compartiments de la figure 12.1, décrit par l'équation d'état (12.6), avec **A** et **b** donnés par (12.7) et (12.8). Pour simplifier les notations, posons

$$\mathbf{p} = (\theta_{2,1} \quad \theta_{1,2} \quad \theta_{0,1})^{\mathrm{T}}. \tag{12.228}$$

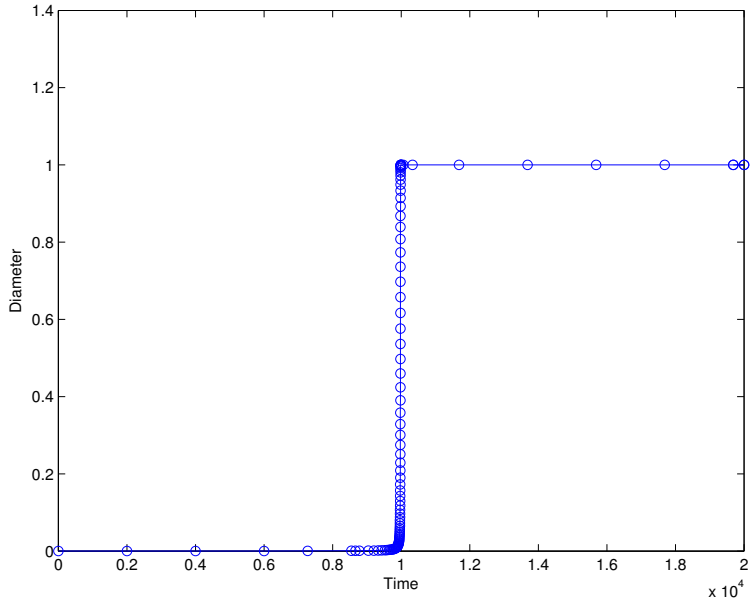


**Fig. 12.10** `ode23s` sur l'évolution d'une boule de feu quand $y_0 = 0.0001$

Supposons $\mathbf{p}$ à estimer sur la base de mesures des contenus $x_1$ et $x_2$ des deux compartiments à des instants donnés, quand il n'y a pas d'entrée et qu'on sait que les conditions initiales sont

$$\mathbf{x} = (1 \quad 0)^{\mathrm{T}}. \tag{12.229}$$

Des données artificielles peuvent être générées en résolvant le problème de Cauchy correspondant pour une vraie valeur $\mathbf{p}^\star$ du vecteur des paramètres. On peut alors calculer une estimée $\widehat{\mathbf{p}}$ de $\mathbf{p}^\star$ en minimisant une norme $J(\mathbf{p})$ de la différence entre les sorties du modèle calculées en $\mathbf{p}^\star$ et en $\mathbf{p}$. Pour minimiser $J(\mathbf{p})$, la routine d'optimisation non linéaire doit passer la valeur de $\mathbf{p}$ à un solveur d'EDO. Aucun des solveurs d'EDO de MATLAB n'est près à accepter cela directement, et c'est pourquoi des fonctions imbriquées vont être utilisées, comme expliqué dans la documentation MATLAB.

Supposons pour commencer que la vraie valeur du vecteur des paramètres soit

$$\mathbf{p}^\star = (0.6 \quad 0.15 \quad 0.35)^{\mathrm{T}}, \tag{12.230}$$

et que les instants de mesure soient

$$\mathbf{t} = (0 \quad 1 \quad 2 \quad 4 \quad 7 \quad 10 \quad 20 \quad 30)^{\mathrm{T}}. \tag{12.231}$$

Notons que ces instants ne sont pas régulièrement espacés. Le solveur d'EDO devra produire les valeurs prises par des solutions approchées à ces instants spécifiques ainsi que sur une grille permettant de tracer correctement les solutions en temps continu sous-jacentes. Ceci est mené à bien par la fonction suivante, qui génère les données de la figure 12.11.

```
function Compartments
% Paramètres
p = [0.6;0.15;0.35];
% Conditions initiales
x0 = [1;0];
% Instants et plage des mesures
Times = [0,1,2,4,7,10,20,30];
Range = [0:0.01:30];
% Options du solveur
options = odeset('RelTol',1e-6);

% Résolution du problème de Cauchy
% Le solveur est appelé deux fois, pour
% les instants et la plage des mesures
[t,X] = SimulComp(Times,x0,p);
[r,Xr] = SimulComp(Range,x0,p);
function [t,X] = SimulComp(RangeOrTimes,x0,p)
[t,X] = ode45(@Compart,RangeOrTimes,x0,options);
    function [xDot]= Compart(t,x)
    % définit les équations d'état
    M = [-(p(1)+p(3)), p(2);p(1),-p(2)];
    xDot = M*x;
    end
end

% Tracé des résultats
figure;
hold on
plot(t,X(:,1),'ks');plot(t,X(:,2),'ko');
plot(r,Xr(:,1));plot(r,Xr(:,2));
legend('x_1','x_2');ylabel('State');xlabel('Time')
end
```

Supposons maintenant que la vraie valeur du vecteur des paramètres soit

$$\mathbf{p}^\star = (0.6 \quad 0.35 \quad 0.15)^{\mathrm{T}}, \tag{12.232}$$

ce qui correspond à échanger les valeurs de $p_2^\star$ et $p_3^\star$. `Compartments` produit maintenant les données décrites par la figure 12.12.

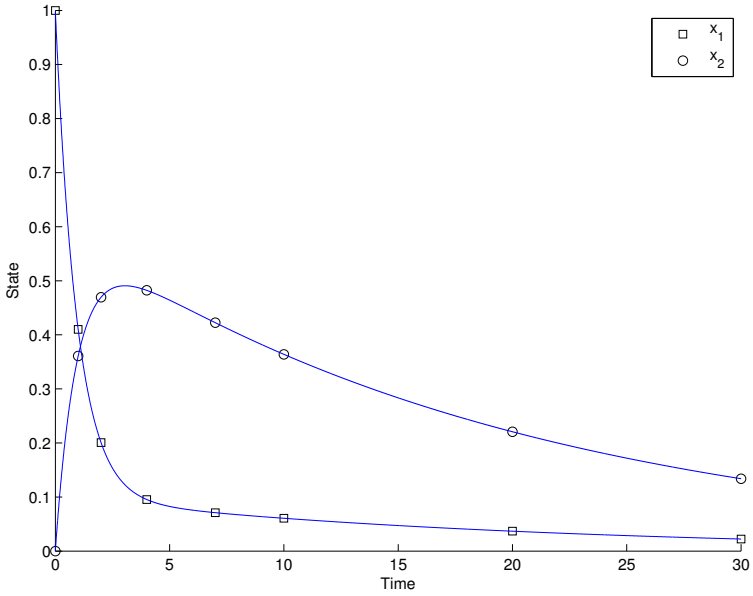


**Fig. 12.11** Données générées pour le modèle à compartiments de la figure 12.1 par `ode45` pour $\mathbf{x}(0) = (1,0)^{\mathrm{T}}$ et $\mathbf{p}^{\star} = (0.6, 0.15, 0.35)^{\mathrm{T}}$

Si les solutions pour $x_1$ sont très différentes dans les figures 12.11 et 12.12, les solutions pour $x_2$ sont *extrêmement* similaires, ce que confirme la figure 12.13 qui correspond à leur différence.

Ceci n'est en fait pas surprenant, car une analyse d'identifiabilité [243] montrerait que les paramètres de ce modèle ne peuvent pas être estimés de façon unique à partir de mesures effectuées sur le seul $x_2$, parce que la solution pour $x_2$ est invariante par échange des rôles de $p_2$ et $p_3$. Si nous avions essayé d'estimer $\widehat{\mathbf{p}}$ avec l'une quelconque des méthodes locales d'optimisation non linéaire présentées au chapitre 9 à partir de données artificielles sans bruit sur le seul $x_2$, nous aurions convergé vers une approximation de $\mathbf{p}^{\star}$ comme donné par (12.230) ou par (12.232) suivant notre initialisation. Une stratégie *multistart* nous aurait sans doute permis de détecter l'existence de deux minimiseurs globaux, tous deux associés à un minimum global très petit.

### *12.4.4 Problème aux limites*

Un fluide sous pression à haute température circule dans un tuyau droit long et épais. Nous considérons une coupe de ce tuyau, située loin de ses extrémités. La

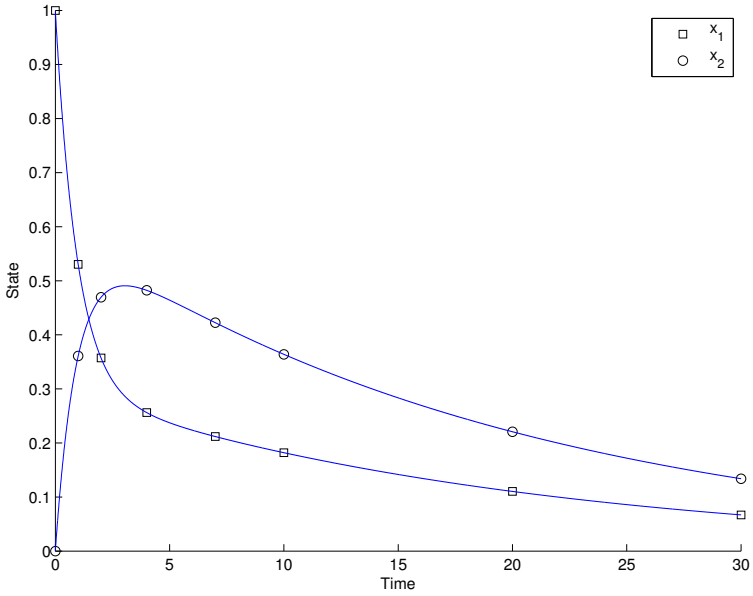


**Fig. 12.12** Données générées pour le modèle à compartiments de la figure 12.1 par `ode45` pour $\mathbf{x}(0) = (1,0)^{\mathrm{T}}$ et $\mathbf{p}^{\star} = (0.6, 0.35, 0.15)^{\mathrm{T}}$

symétrie de rotation du problème permet d'étudier la distribution des températures à l'équilibre le long d'un rayon de cette coupe. Le rayon interne du tuyau est $r_{\mathrm{in}} = 1$ cm, et son rayon externe est $r_{\mathrm{out}} = 2$ cm. La température (en C) au rayon $r$ (en cm), notée $T(r)$, est supposée satisfaire

$$\frac{\mathrm{d}^2 T}{\mathrm{d}r^2} = -\frac{1}{r}\frac{\mathrm{d}T}{\mathrm{d}r}, \tag{12.233}$$

et les conditions aux deux bouts sont

$$T(1) = 100 \quad \text{et} \quad T(2) = 20. \tag{12.234}$$

L'équation (12.233) peut être mise sous la forme d'état

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}r} = \mathbf{f}(\mathbf{x}, r), \tag{12.235}$$

$$T(r) = g(\mathbf{x}(r)), \tag{12.236}$$

avec

$$\mathbf{x}(r) = (T(r), \dot{T}(r))^{\mathrm{T}}, \tag{12.237}$$

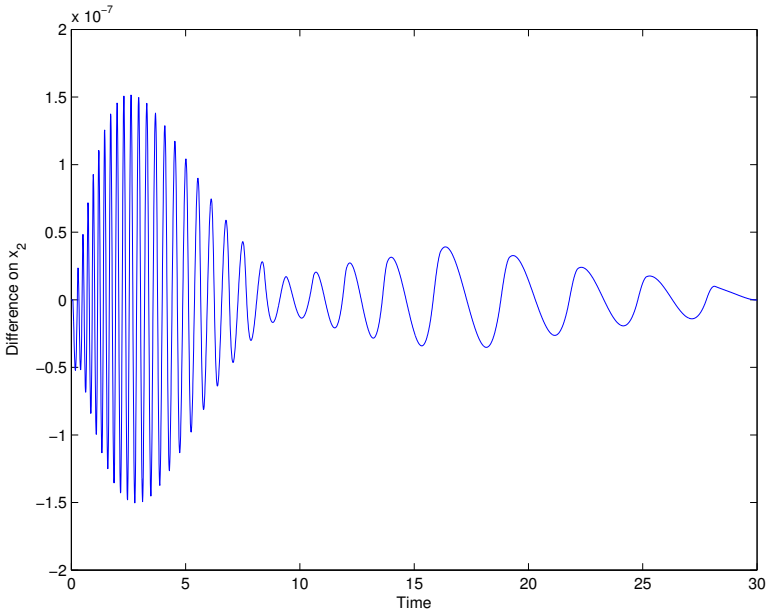


**Fig. 12.13** Différence entre les solutions pour $x_2$ quand $\mathbf{p}^\star = (0.6, 0.15, 0.35)^{\mathrm{T}}$ et quand $\mathbf{p}^\star = (0.6, 0.35, 0.15)^{\mathrm{T}}$, telles que calculées par `ode45`

$$\mathbf{f}(\mathbf{x}, r) = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{r} \end{bmatrix} \mathbf{x}(r) \tag{12.238}$$

et

$$g(\mathbf{x}(r)) = (1 \quad 0)\, \mathbf{x}(r), \tag{12.239}$$

et les conditions aux deux bouts deviennent

$$x_1(1) = 100 \quad \text{et} \quad x_1(2) = 20. \tag{12.240}$$

Ce problème aux deux bouts peut être résolu analytiquement, ce qui fournit la solution de référence à laquelle nous pourrons comparer les solutions obtenues par des méthodes numériques .

#### 12.4.4.1 Calcul de la solution analytique

Il est facile de montrer que

$$T(r) = p_1 \ln(r) + p_2, \tag{12.241}$$

avec $p_1$ et $p_2$ spécifiés par les conditions aux deux bouts et obtenus en résolvant le système linéaire

$$\begin{bmatrix} \ln(r_{\text{in}}) & 1 \\ \ln(r_{\text{out}}) & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} T(r_{\text{in}}) \\ T(r_{\text{out}}) \end{bmatrix}. \tag{12.242}$$

Le script qui suit évalue et trace la solution analytique sur une grille régulière entre $r = 1$ et $r = 2$

```
Radius = (1:0.01:2);
A = [log(1),1;log(2),1];
b = [100;20];
p = A\b;
MathSol = p(1)*log(Radius)+p(2);
figure;
plot(Radius,MathSol)
xlabel('Radius')
ylabel('Temperature')
```

Il produit la figure 12.14. Les méthodes numériques utilisées dans les sections 12.4.4.2 à 12.4.4.4 pour résoudre ce problème aux deux bouts produisent des tracés qui sont visuellement indiscernables de la figure 12.14, de sorte que ce sont les erreurs entre les solutions numérique et analytique que nous tracerons.

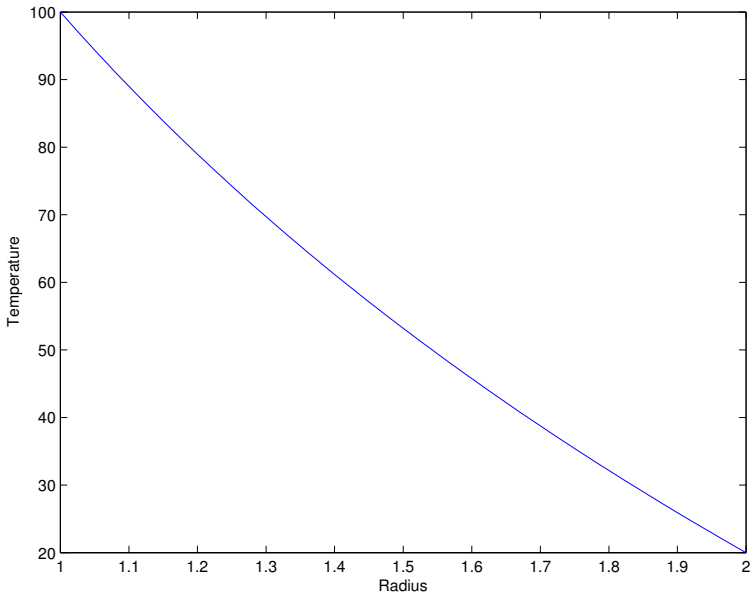


**Fig. 12.14** Distribution des températures dans le tuyau, telle que calculée analytiquement

#### 12.4.4.2 Utilisation d'une méthode de tir

Pour calculer la distribution des températures entre $r_{\text{in}}$ et $r_{\text{out}}$ avec une méthode de tir, paramétrons le second élément de l'état en $r_{\text{in}}$ par $p$. Pour n'importe quelle valeur donnée de $p$, le calcul de la distribution des températures dans le tuyau est un problème de Cauchy. Le script qui suit recherche la valeur `pHat` de $p$ qui minimise le carré de l'écart entre la température connue en $r_{\text{out}}$ et celle calculée par `ode45`, et compare le profile de températures résultant au profil analytique obtenu en section 12.4.4.1. Il produit la figure 12.15.

```
% Résolution du problème de tuyau
% par la méthode de tir
clear all
p0 = -50; % Initialisation de  x_2(1)
pHat = fminsearch(@PipeCost,p0)

% Comparaison avec la solution mathématique
X1 = [100;pHat];
[Radius, SolByShoot] = ...
    ode45(@PipeODE,[1,2],X1);
A = [log(1),1;log(2),1];
b = [100;20];
p = A\b;
MathSol = p(1)*log(Radius)+p(2);
Error = MathSol-SolByShoot(:,1);

% Tracé de l'erreur
figure;
plot(Radius,Error)
xlabel('Radius')
ylabel('Error on temperature of the shooting method')
```

L'EDO (12.235) est implémentée dans la fonction

```
function [xDot] = PipeODE(r,x)
xDot = [x(2); -x(2)/r];
end
```

La fonction

```
function [r,X] = SimulPipe(p)
X1 = [100;p];
[r,X] = ode45(@PipeODE,[1,2],X1);
end
```

est utilisée pour résoudre le problème de Cauchy pour la valeur de $x_2(1) = \dot{T}(r_{\text{in}})$ définie par `p`, et la fonction

```
function [Cost] = PipeCost(p)
[Radius,X] = SimulPipe(p);
Cost = (20 - X(length(X),1))^2;
end
```

évalue le coût à minimiser par `fminsearch`.

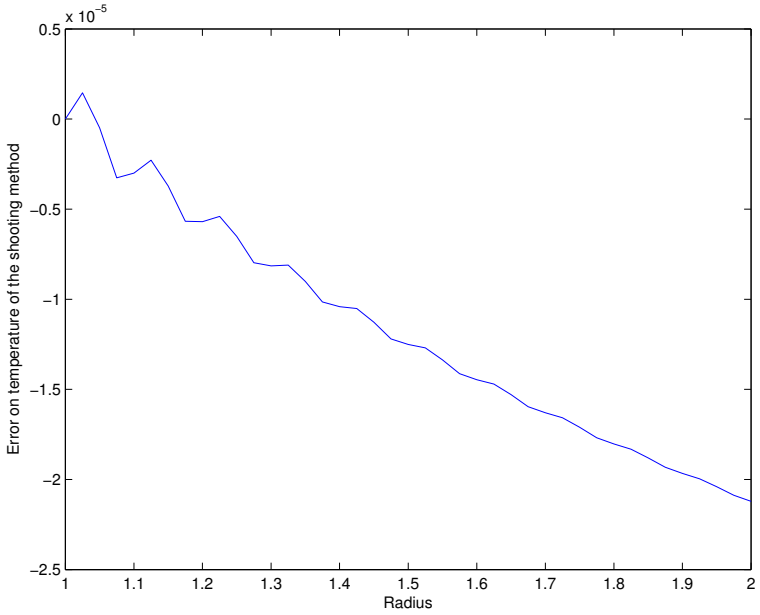


**Fig. 12.15** Erreur sur la distribution des températures dans le tuyau, telle que calculée avec la méthode de tir

#### 12.4.4.3 Utilisation des différences finies

Pour calculer la distribution des températures entre $r_{\text{in}}$ et $r_{\text{out}}$ avec une méthode par différences finies, il suffit de spécialiser (12.184) en (12.233), ce qui revient à poser

$$a_1(r) = 1/r, \tag{12.243}$$
$$a_2(r) = 0, \tag{12.244}$$
$$u(r) = 0. \tag{12.245}$$

Ceci est implémenté dans le script qui suit, dans lequel `sAgrid` et `sbgrid` sont des représentations creuses de $\mathbf{A}$ et $\mathbf{b}$ tels que définis par (12.190) et (12.191).

```
% Résolution du problème de tuyau par MDF
clear all

% Valeurs aux limites
InitialSol = 100;
FinalSol = 20;

% Spécification de la grille
Step = 0.001; % step-size
Grid = (1:Step:2)';
NGrid = length(Grid);
% Np = nombre des points de la grille
% où la solution est inconnue
Np = NGrid-2;
Radius = zeros(Np,1);
for i = 1:Np;
    Radius(i) = Grid(i+1);
end

% Construction du système linéaire creux
% à résoudre
a = zeros(Np,1);
c = zeros(Np,1);
HalfStep = Step/2;
for i=1:Np,
    a(i) = 1-HalfStep/Radius(i);
    c(i) = 1+HalfStep/Radius(i);
end
sAgrid = -2*sparse(1:Np,1:Np,1);
sAgrid(1,2) = c(1);
sAgrid(Np,Np-1) = a(Np);
for i=2:Np-1,
    sAgrid(i,i+1) = c(i);
    sAgrid(i,i-1) = a(i);
end
sbgrid = sparse(1:Np,1,0);
sbgrid(1) = -a(1)*InitialSol;
sbgrid(Np) = -c(Np)*FinalSol;

% Résolution du système linéaire creux
pgrid = sAgrid\sbgrid;
SolByFD = zeros(NGrid,1);
SolByFD(1) = InitialSol;
SolByFD(NGrid) = FinalSol;
for i = 1:Np,
```

```
    SolByFD(i+1) = pgrid(i);
end

% Comparaison avec la solution mathématique
A = [log(1),1;log(2),1];
b = [100;20];
p = A\b;
MathSol = p(1)*log(Grid)+p(2);
Error = MathSol-SolByFD;

% Tracé de l'erreur
figure;
plot(Grid,Error)
xlabel('Radius')
ylabel('Error on temperature of the FDM')
```

Ce script produit la figure 12.16.

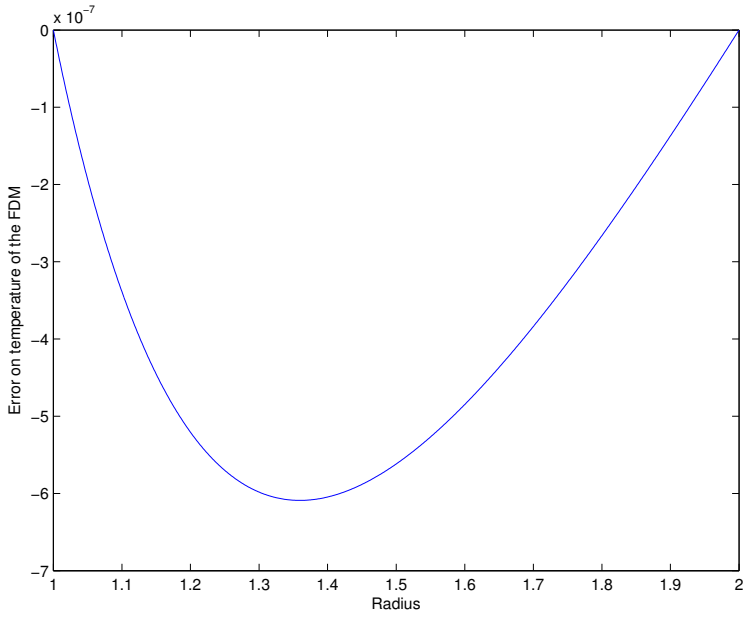


**Fig. 12.16** Erreur sur la distribution des températures dans le tuyau, telle que calculée avec la méthode des différences finies

*Remarque 12.19.* Nous avons exploité le caractère creux de $\mathbf{A}$, mais pas le fait qu'elle soit tridiagonale. Avec un pas de taille $10^{-3}$ comme dans ce script, une représentation dense de $\mathbf{A}$ aurait $10^6$ éléments. □

#### 12.4.4.4 Utilisation de la collocation

Des détails sur les principes et des exemples d'utilisation du solveur par collocation `bvp4c` peuvent être trouvés dans [215, 127]. L'EDO (12.235) reste décrite par la fonction `PipeODE`, et les erreurs sur la satisfaction des conditions aux deux bouts sont évaluées par la fonction

```
function [ResidualsOnBounds] = ...
    PipeBounds(xa,xb)
ResidualsOnBounds = [xa(1) - 100
                     xb(1) - 20];
end
```

Il faut fournir au solveur une approximation initiale de la solution. Le script qui suit la prend identiquement nulle sur $[1,2]$. La fonction d'assistance `bvpinit` se charge alors de construire une structure correspondant à cette hypothèse audacieuse avant l'appel à `bvp4c`. Finalement, la fonction `deval` est en charge de l'évaluation de la solution approchée fournie par `bvp4c` sur une grille identique à celle utilisée pour la solution mathématique.

```
% Résolution du problème de tuyau par collocation
clear all

% Choix d'un point de départ
Radius = (1:0.1:2); % Maillage initial
xInit = [0; 0]; % Approximation initiale de la solution

% Construction d'une structure pour cette approximation
PipeInit = bvpinit(Radius,xInit);

% Appel du solveur par collocation
SolByColloc = bvp4c(@PipeODE,...
    @PipeBounds,PipeInit);
VisuCollocSol = deval(SolByColloc,Radius);

% Comparaison avec la solution mathématique
A = [log(1),1;log(2),1];
b = [100;20];
p = A\b;
MathSol = p(1)*log(Radius)+p(2);
Error = MathSol-VisuCollocSol(1,:);
```

```
% Tracé de l'erreur
figure;
plot(Radius,Error)
xlabel('Radius')
ylabel('Error on temperature of the collocation method')
```

Les résultats sont en figure 12.17. On peut obtenir une solution plus précise en faisant décroître la tolérance relative par rapport à sa valeur de défaut $10^{-3}$ (on pourrait aussi choisir une approximation initiale plus raffinée à passer à `bvp4c` par `bvpinit`). En remplaçant l'appel à `bvp4c` dans le script qui précède par

```
optionbvp = bvpset('RelTol',1e-6)
SolByColloc = bvp4c(@PipeODE,...
    @PipeBounds,PipeInit,optionbvp);
```

pour améliorer la précision de la solution, nous obtenons les résultats de la figure 12.18.

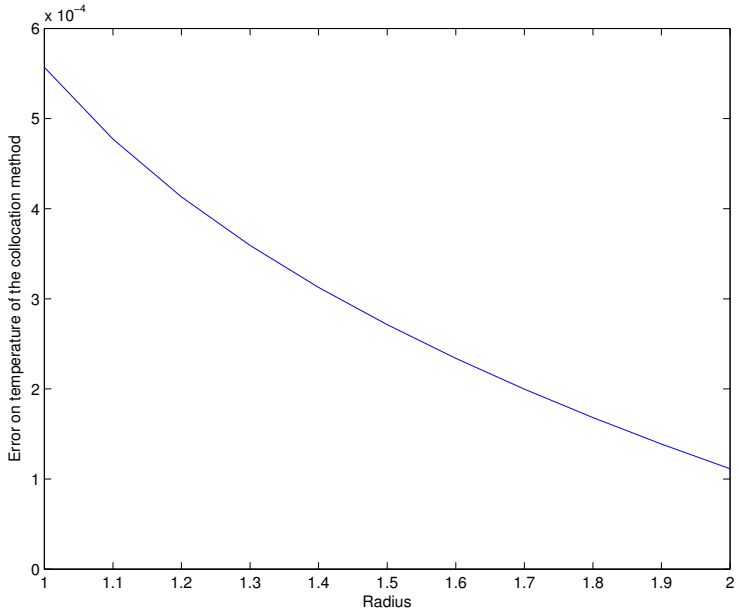


**Fig. 12.17** Erreur sur la distribution des températures dans le tuyau, telle que calculée par la méthode de collocation implémentée dans `bvp4c` avec `RelTol` = $10^{-3}$

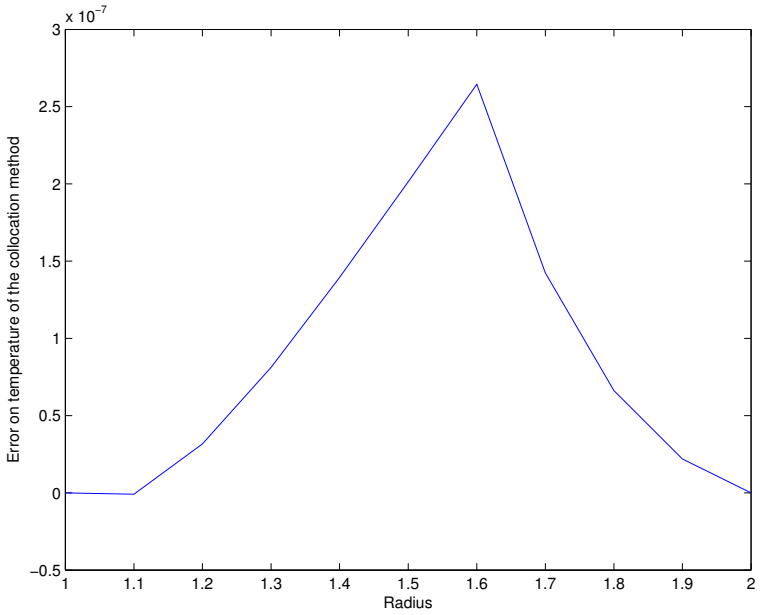


**Fig. 12.18** Erreur sur la distribution des températures dans le tuyau, telle que calculée par la méthode de collocation implémentée dans `bvp4c` avec `RelTol` = $10^{-6}$

## 12.5 En résumé

— Les EDO n'ont qu'une variable indépendante, qui n'est pas forcément le temps.
— La plupart des méthodes pour résoudre des EDO requièrent qu'elles soient mises sous forme d'état, ce qui n'est pas toujours possible ou désirable.
— Les problèmes aux valeurs initiales sont plus faciles à résoudre que les problèmes aux limites, dont les problèmes aux deux bouts sont un cas particulier.
— La résolution d'EDO raides avec des solveurs pour EDO non raides est possible, mais *très* lente.
— Les méthodes disponibles pour résoudre les problèmes aux valeurs initiales peuvent être explicites ou implicites, à un pas ou plusieurs pas.
— Les méthodes implicites ont de meilleures propriétés de stabilité que les méthodes explicites. Elles sont par contre plus difficiles à mettre en œuvre, à moins que leurs équations puissent être mises sous forme explicite.
— Les méthodes explicites à un pas peuvent démarrer toutes seules. On peut les utiliser pour initialiser des méthodes à plusieurs pas.

— La plupart des méthodes à un pas requièrent des évaluations intermédiaires de la dérivée de l'état qui ne peuvent pas être réutilisées. Ceci tend à les rendre moins efficaces que les méthodes à plusieurs pas.
— Les méthodes linéaires à plusieurs pas ont besoin des méthodes à un pas pour démarrer. Elles font un usage plus efficace des évaluations de la dérivée de l'état mais sont moins robustes aux mers démontées.
— Il est souvent utile d'adapter la taille du pas le long de la trajectoire d'état, ce qu est facile avec les méthodes à un pas.
— Il est souvent utile d'adapter l'ordre de la méthode le long de la trajectoire d'état, ce qui est facile avec les méthodes linéaires à plusieurs pas.
— La résolution de problèmes aux limites peut se faire via des méthodes de tir et la minimisation d'une norme des déviations de la solution par rapport aux conditions aux limites, pourvu que l'EDO soit stable.
— Les méthodes par différences finies n'exigent pas que les EDO soient mises sous forme d'état. On peut les utiliser pour résoudre des problèmes aux valeurs initiales et aux limites. Un important ingrédient en est la résolution de (grands) systèmes (creux) d'équations linéaires.
— Les approches par projection utilisent des approximations de dimension finie des solutions des EDO. Les paramètres libres de ces approximations sont évalués en résolvant un système d'équations (approches par collocation et de Ritz-Galerkin) ou en minimisant un coût quadratique (approche des moindres carrés).
— La compréhension des approches par différences finies et par projection pour les EDO devrait faciliter celle des mêmes techniques pour les équations aux dérivées partielles.

# Chapitre 13
# Simuler des équations aux dérivées partielles

## 13.1 Introduction

Contrairement aux équations différentielles ordinaires (ou EDO) considérées au chapitre 12, les équations aux dérivées partielles (ou EDP) impliquent plus d'une variable indépendante. Les modèles physiques à base de connaissances impliquent typiquement des EDP (équations de Maxwell en électromagnétisme, équation de Schrödinger en mécanique quantique, équation de Navier-Stokes en mécanique des fluides, équation de Fokker-Planck en mécanique statistique, etc.). Ce n'est que dans des cas très particuliers que les EDP se simplifient en EDO. En génie chimique, par exemple, les concentrations des espèces réagissantes obéissent en général à des EDP. C'est seulement dans les réacteurs continus parfaitement agités qu'on peut les considérer comme indépendantes de la position et que le temps devient la seule variable indépendante.

L'étude des propriétés mathématiques des EDP est considérablement plus complexe que celle des EDO. Prouver, par exemple, qu'il existe des solutions lisses des équations de Navier-Stokes sur $\mathbb{R}^3$ (ou donner un contre-exemple) serait l'un des accomplissements pour lesquels le *Clay Mathematics Institute* est prêt, depuis mai 2000, à attribuer l'un de ses sept *Millennium Prizes* d'un million de dollars.

Ce chapitre ne fera qu'effleurer la surface de la simulation des EDP. De bons points de départ pour aller plus loin sont [154], qui aborde la modélisation de problèmes réels, l'analyse des modèles à EDP résultants et leur simulation numérique via une approche par différences finies, [110], qui développe de nombreux schémas de différences finies avec des applications en mécanique des fluides et [138], qui considère à la fois des méthodes par différences finies et par éléments finis. Chacun de ces livres traite de nombreux exemples de façon détaillée.

## 13.2 Classification

Les méthodes à choisir pour résoudre des EDP dépendent, entre autres, de leur caractère linéaire ou pas, de leur ordre, et du type des conditions aux limites considérées.

### *13.2.1 EDP linéaires et non linéaires*

Comme avec les EDO, un cas particulier important est quand les variables dépendantes et leurs dérivées partielles par rapport aux variables indépendantes entrent linéairement dans l'EDP. L'*équation des ondes* scalaire à deux dimensions d'espace

$$\frac{\partial^2 y}{\partial t^2} = c^2 \left( \frac{\partial^2 y}{\partial x_1^2} + \frac{\partial^2 y}{\partial x_2^2} \right), \tag{13.1}$$

où $y(t,\mathbf{x})$ spécifie un déplacement au temps $t$ et au point $\mathbf{x} = (x_1, x_2)^{\mathrm{T}}$ dans un espace 2D et où $c$ est la vitesse de propagation, est ainsi une EDP linéaire. Ses variables indépendantes sont $t, x_1$ et $x_2$, et sa variable dépendante est $y$. Le principe de superposition s'applique aux EDP linéaires, de sorte que la somme de deux solutions est une solution. Les coefficients des EDP linéaires peuvent être des fonctions des variables indépendantes, mais pas des variables dépendantes.

L'*équation de Burgers avec viscosité* de la mécanique des fluides

$$\frac{\partial y}{\partial t} + y \frac{\partial y}{\partial x} = \nu \frac{\partial^2 y}{\partial x^2}, \tag{13.2}$$

où $y(t,x)$ est la vitesse d'écoulement du fluide et $\nu$ sa viscosité, est non linéaire, car le second terme de son membre de gauche implique le produit de $y$ par sa dérivée partielle par rapport à $x$.

### *13.2.2 Ordre d'une EDP*

L'ordre d'une EDP scalaire est celui de la dérivée d'ordre le plus élevé de la variable dépendante par rapport aux variables indépendantes. L'équation (13.2) est ainsi une EDP du second ordre sauf quand $\nu = 0$, ce qui correspond à l'*équation de Burgers pour les fluides non visqueux*

$$\frac{\partial y}{\partial t} + y \frac{\partial y}{\partial x} = 0, \tag{13.3}$$

qui est du premier ordre. Comme pour les EDO, une EDP scalaire peut être décomposée en un système d'EDP du premier ordre. L'ordre de ce système est alors celui de l'EDP scalaire de départ.

*Exemple 13.1.* Le système de trois EDP du premier ordre scalaires

$$\begin{aligned} \frac{\partial u}{\partial x_1} + \frac{\partial v}{\partial x_2} &= \frac{\partial u}{\partial t}, \\ u &= \frac{\partial y}{\partial x_1}, \\ v &= \frac{\partial y}{\partial x_2} \end{aligned} \tag{13.4}$$

équivaut à

$$\frac{\partial^2 y}{\partial x_1^2} + \frac{\partial^2 y}{\partial x_2^2} = \frac{\partial^2 y}{\partial t \partial x_1}. \tag{13.5}$$

Son ordre est donc deux. □

### *13.2.3 Types de conditions aux limites*

Comme pour les EDO, il faut imposer des conditions aux limites pour spécifier les solutions des EDP, et nous supposons ces conditions telles qu'il y a au moins une solution.

- Les *conditions de Dirichlet* spécifient des valeurs de la solution $y$ sur la frontière $\partial\mathbb{D}$ du domaine d'étude $\mathbb{D}$. Ceci peut correspondre, par exemple, au potentiel à la surface d'une électrode, à la température à une extrémité d'une barre ou à la position d'une extrémité fixe d'une corde vibrante.
- Les *conditions de Neumann* spécifient des valeurs du flux $\frac{\partial y}{\partial \mathbf{n}}$ de la solution à travers $\partial\mathbb{D}$, avec $\mathbf{n}$ un vecteur normal à $\partial\mathbb{D}$. Ceci peut correspondre, par exemple, à l'injection d'un courant électrique dans un système.
- Les *conditions de Robin* sont des combinaisons linéaires de conditions de Dirichlet et de Neumann.
- Les *conditions aux limites mixtes* sont telles qu'une condition de Dirichlet s'applique à une partie de $\partial\mathbb{D}$ et une condition de Neumann à une autre partie.

### *13.2.4 Classification des EDP linéaires du second ordre*

Les EDP linéaires du second ordre sont suffisamment importantes pour recevoir leur propre classification. Nous supposons ici, pour simplifier, qu'il n'y a que deux variables indépendantes $t$ et $x$ et que la solution $y(t,x)$ est scalaire. La première

variable indépendante peut être associée au temps et la seconde à l'espace, mais d'autres interprétations sont bien sûr possibles.

*Remarque 13.1.* Souvent, $x$ devient un vecteur $\mathbf{x}$, qui peut spécifier la position dans un espace 2D ou 3D, et la solution $y$ devient aussi un vecteur $\mathbf{y}(t,\mathbf{x})$, parce qu'on est intéressé, par exemple, par la température *et* la composition chimique à l'instant $t$ et au point $\mathbf{x}$ dans un réacteur piston. De tels problèmes, qui impliquent plusieurs domaines de la physique et de la chimie (ici, la mécanique des fluides, la thermodynamique et la cinétique chimique), relèvent de ce qu'on appelle parfois la *multiphysique*. □

Pour alléger les notations, posons

$$y_x \equiv \frac{\partial y}{\partial x}, \quad y_{xx} \equiv \frac{\partial^2 y}{\partial x^2}, \quad y_{xt} \equiv \frac{\partial^2 y}{\partial x \partial t}, \tag{13.6}$$

et ainsi de suite. L'opérateur laplacien, par exemple, est alors tel que

$$\Delta y = y_{tt} + y_{xx}. \tag{13.7}$$

Toutes les EDP considérées ici peuvent s'écrire

$$a y_{tt} + 2b y_{tx} + c y_{xx} = g(t,x,y,y_t,y_x). \tag{13.8}$$

Comme les solutions de (13.8) doivent être telles que

$$\mathrm{d}y_t = y_{tt}\mathrm{d}t + y_{tx}\mathrm{d}x, \tag{13.9}$$
$$\mathrm{d}y_x = y_{xt}\mathrm{d}t + y_{xx}\mathrm{d}x, \tag{13.10}$$

où $y_{xt} = y_{tx}$, le système d'équations linéaires suivant doit être satisfait

$$\mathbf{M}\begin{bmatrix} y_{tt} \\ y_{tx} \\ y_{xx} \end{bmatrix} = \begin{bmatrix} g(t,x,y,y_t,y_x) \\ \mathrm{d}y_t \\ \mathrm{d}y_x \end{bmatrix}, \tag{13.11}$$

où

$$\mathbf{M} = \begin{bmatrix} a & 2b & c \\ \mathrm{d}t & \mathrm{d}x & 0 \\ 0 & \mathrm{d}t & \mathrm{d}x \end{bmatrix}. \tag{13.12}$$

La solution $y(t,x)$ est supposée une fois continûment différentiable par rapport à $t$ et $x$. Des discontinuités peuvent apparaître dans les dérivées secondes si $\det\mathbf{M} = 0$, c'est à dire quand

$$a(\mathrm{d}x)^2 - 2b(\mathrm{d}x)(\mathrm{d}t) + c(\mathrm{d}t)^2 = 0. \tag{13.13}$$

Divisons (13.13) par $(\mathrm{d}t)^2$ pour obtenir

$$a\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 - 2b\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right) + c = 0. \tag{13.14}$$

Les solutions de cette équation sont telles que

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{b \pm \sqrt{b^2 - \mathrm{ac}}}{a}. \tag{13.15}$$

Elles définissent les *courbes caractéristiques* de l'EDP. Le nombre des solutions réelles dépend du signe du discriminant $b^2 - ac$.

— Quand $b^2 - ac < 0$, il n'y a pas de courbe caractéristique réelle et l'EDP est *elliptique*.
— Quand $b^2 - ac = 0$, il n'y a qu'une courbe caractéristique réelle et l'EDP est *parabolique*.
— Quand $b^2 - ac > 0$, il y a deux courbes caractéristiques réelles et l'EDP est *hyperbolique*.

Cette classification ne dépend que des coefficients des dérivées d'ordre le plus élevé de l'EDP. Les qualificatifs de ces trois types d'EDP ont été choisis parce que, dans l'espace $(\mathrm{d}x, \mathrm{d}t)$, l'équation quadratique

$$a(\mathrm{d}x)^2 - 2b(\mathrm{d}x)(\mathrm{d}t) + c(\mathrm{d}t)^2 = \text{constante} \tag{13.16}$$

définit une ellipse si $b^2 - ac < 0$, une parabole si $b^2 - ac = 0$ et une hyperbole si $b^2 - ac > 0$.

*Exemple 13.2.* L'*équation de Laplace* en électrostatique

$$y_{tt} + y_{xx} = 0, \tag{13.17}$$

avec $y$ un potentiel, est elliptique. L'*équation de la chaleur*

$$cy_{xx} = y_t, \tag{13.18}$$

avec $y$ une température, est parabolique. L'*équation des cordes vibrantes*

$$ay_{tt} = y_{xx}, \tag{13.19}$$

avec $y$ un déplacement, est hyperbolique. L'équation

$$y_{tt} + (t^2 + x^2 - 1)y_{xx} = 0 \tag{13.20}$$

est elliptique hors du cercle unité centré en $(0,0)$, et hyperbolique à l'intérieur. □

*Exemple 13.3.* Un avion volant à Mach 0.7 sera entendu par des observateurs au sol dans toutes les directions, et l'EDP qui décrit la propagation du son pendant un tel vol subsonique est elliptique. Quand la vitesse atteint Mach 1, un front se développe en avant duquel le bruit n'est plus entendu ; ce front correspond à une seule courbe caractéristique réelle, et l'EDP qui décrit la propagation du son durant un tel vol sonique est parabolique. Quand la vitesse croît encore, le bruit n'est plus entendu qu'entre les *lignes de Mach*, qui forment une paire de courbes caractéristiques réelles, et l'EDP qui décrit la propagation du son pendant un tel vol supersonique est

hyperbolique. Les courbes caractéristiques réelles, quand elles existent, raccordent ainsi des solutions radicalement différentes. □

## 13.3 Méthode des différences finies

Comme pour les EDO, l'idée de base de la méthode des différences finies (MDF) est de remplacer l'EDP initiale par une équation approchée liant les valeurs prises par la solution approchée aux nœuds d'une grille. Les aspects analytiques et numériques de l'approche par différences finies des problèmes elliptiques, paraboliques et hyperboliques sont traités dans [154], qui consacre une attention considérable aux problèmes de modélisation et présente nombre d'applications pratiques. Voir aussi [110] et [138].

Nous supposons ici que la grille sur laquelle la solution sera approchée est régulière, et telle que

$$t_l = t_1 + (l-1)h_t, \tag{13.21}$$
$$x_m = x_1 + (m-1)h_x, \tag{13.22}$$

comme illustré par la figure 13.1. (Cette hypothèse pourrait être relaxée.)

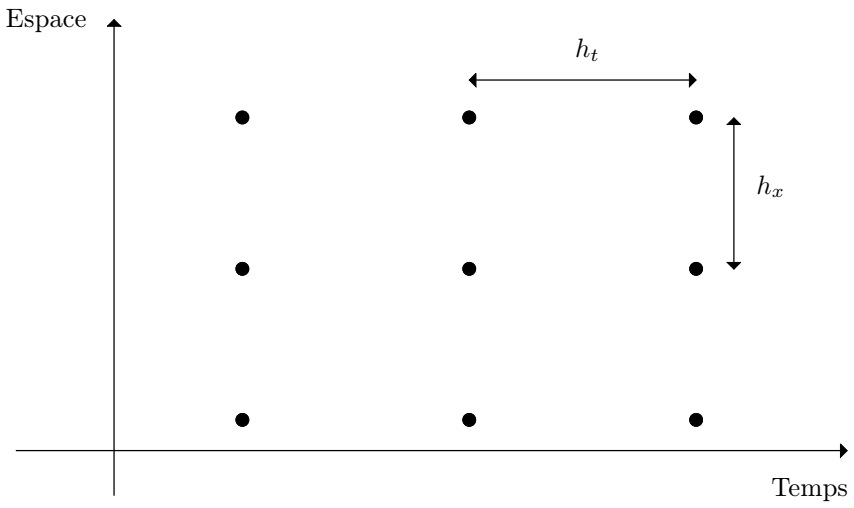


**Fig. 13.1** Grille régulière

### 13.3.1 Discrétisation de l'EDP

La procédure est très proche de celle utilisée pour les EDO dans la section 12.3.3 :

1. Remplacer dans l'EDP les dérivées partielles par des approximations par différences finies, par exemple

$$y_t(t_l, x_m) \approx \frac{Y_{l,m} - Y_{l-1,m}}{h_t}, \tag{13.23}$$

$$y_{xx}(t_l, x_m) \approx \frac{Y_{l,m+1} - 2Y_{l,m} + Y_{l,m-1}}{h_x^2}, \tag{13.24}$$

   avec $Y_{l,m}$ la valeur approchée à calculer de $y(t_l, x_m)$.

2. Écrire les équations discrètes résultantes en tous les points de la grille où c'est possible, en tenant compte des informations fournies par les conditions aux limites chaque fois que nécessaire.
3. Résoudre le système d'équations résultant pour trouver les $Y_{l,m}$.

Il y a, bien sûr, des degrés de liberté dans le choix des approximations des dérivées partielles. On peut par exemple choisir

$$y_t(t_l, x_m) \approx \frac{Y_{l,m} - Y_{l-1,m}}{h_t}, \tag{13.25}$$

$$y_t(t_l, x_m) \approx \frac{Y_{l+1,m} - Y_{l,m}}{h_t} \tag{13.26}$$

ou

$$y_t(t_l, x_m) \approx \frac{Y_{l+1,m} - Y_{l-1,m}}{2h_t}. \tag{13.27}$$

On peut exploiter ces degrés de liberté pour faciliter la propagation des informations fournies par les conditions aux limites et pour atténuer l'effet des erreurs de méthodes.

### 13.3.2 Méthodes explicites et implicites

On peut parfois arranger les calculs de telle façon que la solution approchée aux points de la grille où elle est encore inconnue s'exprime comme une fonction des conditions aux limites connues et de valeurs $Y_{i,j}$ déjà calculées. On peut alors calculer la solution approchée en tous les points de la grille par une *méthode explicite*, grâce à une équation de récurrence. Dans les *méthodes implicites*, par contre, toutes les équations reliant tous les $Y_{l,m}$ sont considérées simultanément.

Les méthodes explicites ont deux sérieux inconvénients. D'abord, elles imposent des contraintes sur les tailles de pas pour assurer la stabilité de l'équation de récurrence. Par ailleurs, les erreurs commises durant les pas passés de la récurrence

ont des conséquences sur les pas futurs. C'est pourquoi on peut leur préférer des méthodes implicites même quand on aurait pu les appliquer.

Pour les EDP linéaires, les méthodes implicites requièrent la résolution de grands systèmes d'équations linéaires

$$\mathbf{A}\mathbf{y} = \mathbf{b}, \tag{13.28}$$

avec $\mathbf{y} = \text{vect}(Y_{l,m})$. La difficulté est atténuée par le fait que la matrice $\mathbf{A}$ est creuse et souvent diagonalement dominante, de sorte que les méthodes itératives sont particulièrement bien adaptées (voir la section 3.7). Comme $\mathbf{A}$ peut être gigantesque, il faut prêter attention à la façon dont on range cette matrice en mémoire et à l'indexation des points de la grille, pour éviter de ralentir les calculs par des accès à la mémoire de masse qui auraient pu être évités.

### *13.3.3 Illustration : schéma de Crank-Nicolson*

Considérons l'équation de la chaleur à une seule variable d'espace $x$ :

$$\frac{\partial y(t,x)}{\partial t} = \gamma^2 \frac{\partial^2 y(t,x)}{\partial x^2}. \tag{13.29}$$

Avec les notations simplifiées, cette équation parabolique devient

$$c y_{xx} = y_t, \tag{13.30}$$

où $c = \gamma^2$.

Prenons une approximation *avant du premier ordre* de $y_t(t_l, x_m)$

$$y_t(t_l, x_m) \approx \frac{Y_{l+1,m} - Y_{l,m}}{h_t}. \tag{13.31}$$

Au milieu de l'arrête liant les points de la grille indexés par $(l,m)$ et $(l+1,m)$, elle devient une approximation *centrée du second ordre*

$$y_t(t_l + \frac{h_t}{2}, x_m) \approx \frac{Y_{l+1,m} - Y_{l,m}}{h_t}. \tag{13.32}$$

Pour exploiter cette augmentation de l'ordre de l'erreur de méthode, le schéma de *Crank-Nicolson* approxime (13.29) en de tels points hors grille (figure 13.2). La valeur de $y_{xx}$ au point hors grille indexé par $(l+1/2,m)$ est alors approximée par la moyenne arithmétique de ses valeurs aux deux points adjacents de la grille :

$$y_{xx}(t_l + \frac{h_t}{2}, x_m) \approx \frac{1}{2}[y_{xx}(t_{l+1}, x_m) + y_{xx}(t_l, x_m)], \tag{13.33}$$

avec $y_{xx}(t_l, t_m)$ approximé comme dans (13.24), qui est aussi une approximation au second ordre.

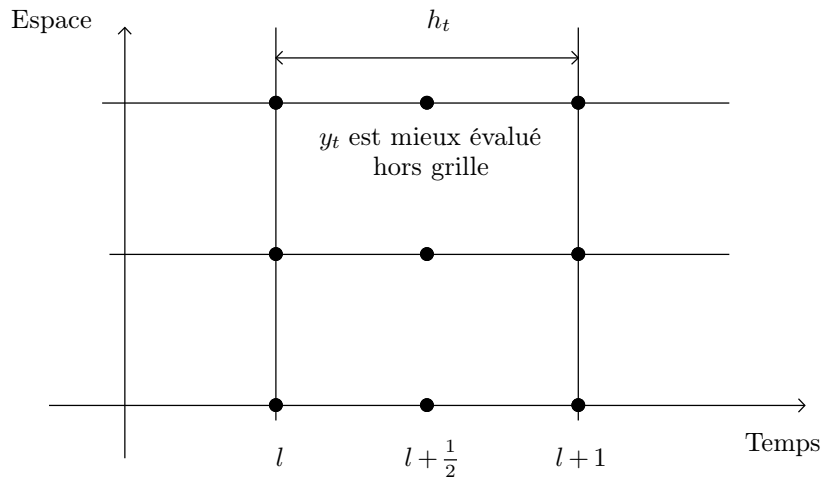


**Fig. 13.2** Schéma de Crank-Nicolson

Si on choisit les pas de temps et d'espace tels que

$$h_t = \frac{h_x^2}{c^2}, \tag{13.34}$$

alors l'EDP (13.30) se traduit par

$$-Y_{l+1,m+1} + 4Y_{l+1,m} - Y_{l+1,m-1} = Y_{l,m+1} + Y_{l,m-1}, \tag{13.35}$$

où les tailles de pas ont disparu.

Supposons que les conditions initiales connues soient

$$Y_{l,1} = y(t_l, x_1), \quad l = 1, \cdots, N, \tag{13.36}$$

$$Y_{l,N} = y(t_l, x_N), \quad l = 1, \cdots, N, \tag{13.37}$$

et que le profil spatial initial connu soit

$$Y(1,m) = y(t_1, x_m), \quad m = 1, \cdots, M, \tag{13.38}$$

et écrivons (13.35) partout où c'est possible. Le profil spatial à l'instant $t_l$ peut alors être calculé en fonction du profil spatial à l'instant $t_{l-1}$, $l = 2, \cdots, N$. On obtient ainsi une solution explicite, puisque le profil spatial initial est connu. On peut préférer une approche implicite, où toutes les équations liant les $Y_{l,m}$ sont considérées simultanément. Le système d'équations résultant peut être mis sous la forme (13.28), avec **A** tridiagonale, ce qui simplifie considérablement sa résolution.

### *13.3.4 Principal inconvénient de la méthode des différences finies*

Le principal inconvénient de la MDF, qui est aussi un argument fort en faveur de la méthode des éléments finis présentée ensuite, est qu'une grille régulière est souvent insuffisamment flexible pour s'adapter à la complexité des conditions aux limites rencontrées dans des applications industrielles, ainsi qu'au besoin de faire varier les tailles de pas quand et où c'est nécessaire pour obtenir des approximations suffisamment précises. Les recherches sur la génération de grilles ont cependant abouti à une situation moins tranchée [110, 88].

## 13.4 Quelques mots sur la méthode des éléments finis

La méthode des éléments finis (MEF) [37] est l'outil principal pour la résolution des EDP avec des conditions aux limites compliquées comme on en rencontre dans les vraies applications en ingénierie, par exemple dans l'industrie aérospatiale. Une présentation détaillée de cette méthode sort du cadre de cet ouvrage, mais nous indiquerons les principales ressemblances et différences par rapport à la MDF.

Parce que le développement d'un logiciel d'éléments finis multiphysique de classe professionnelle est particulièrement complexe, il est encore plus important que pour des cas plus simples de savoir quels sont les logiciels disponibles, avec leurs forces et leurs limitations. De nombreux constituants des solveurs par éléments finis devraient sembler familiers au lecteur des chapitres qui précèdent.

### *13.4.1 Composants de base de la MEF*

#### 13.4.1.1 Mailles

Le domaine d'intérêt dans l'espace des variables indépendantes est partitionné en objets géométriques simples appelés *mailles*, par exemple des triangles dans un espace 2D ou des tétraèdres dans un espace 3D. Le calcul de cette partition et son résultat sont appelés *maillage*. Dans ce qui suit nous utiliserons des mailles triangulaires pour nos illustrations.

Les mailles peuvent être très irrégulières, pour au moins deux raisons :

1. il peut être nécessaire de diminuer la taille des mailles près de la frontière du domaine d'intérêt, afin de décrire ce domaine avec plus de précision,
2. diminuer la taille des mailles partout où l'on s'attend à ce que la norme du gradient de la solution soit grande facilite l'obtention de solutions précises, tout comme adapter la taille du pas fait sens quand on résout des EDO.

Des logiciels, les *mailleurs*, sont disponibles pour automatiser le maillage d'objets géométriques complexes comme ceux que génère la conception assistée par ordi-

nateur, et un maillage manuel est à exclure, même si l'on peut avoir à ajuster un maillage généré automatiquement.

La figure 13.3 présente un maillage créé en un clic sur un domaine ellipsoïdal en utilisant l'interface utilisateur graphique `pdetool` de la *MATLAB PDE Toolbox*. Un deuxième clic produit le maillage plus fin de la figure 13.4. Il est souvent plus économique de laisser le solveur d'EDP raffiner le maillage aux seuls endroits où c'est nécessaire pour obtenir une solution précise.

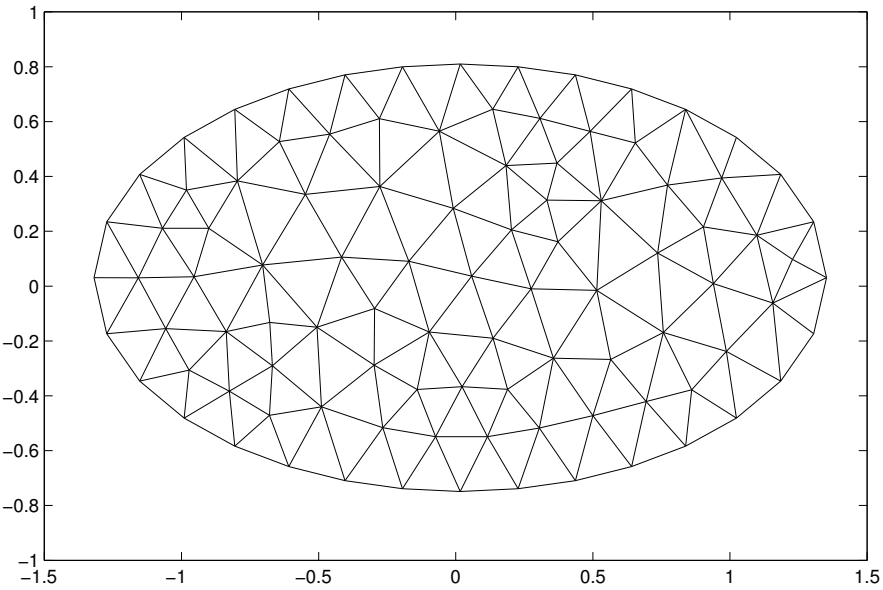


**Fig. 13.3** Maillage créé par `pdetool`

*Remarque 13.2.* En optimisation de forme, un maillage automatique peut devoir être conduit à chaque itération de l'algorithme d'optimisation, puisque la frontière du domaine d'intérêt change. □

*Remarque 13.3.* Les applications peuvent impliquer des milliards de sommets de mailles, et une indexation appropriée de ces sommets est cruciale pour éviter de ralentir les calculs. □

#### 13.4.1.2 Éléments finis

À chaque maille est associé un *élément fini*, qui approxime la solution sur cette maille et est identiquement nul à l'extérieur. (Les splines, décrits en section 5.3.2, peuvent être vus comme des éléments finis sur un maillage 1D constitué d'inter-

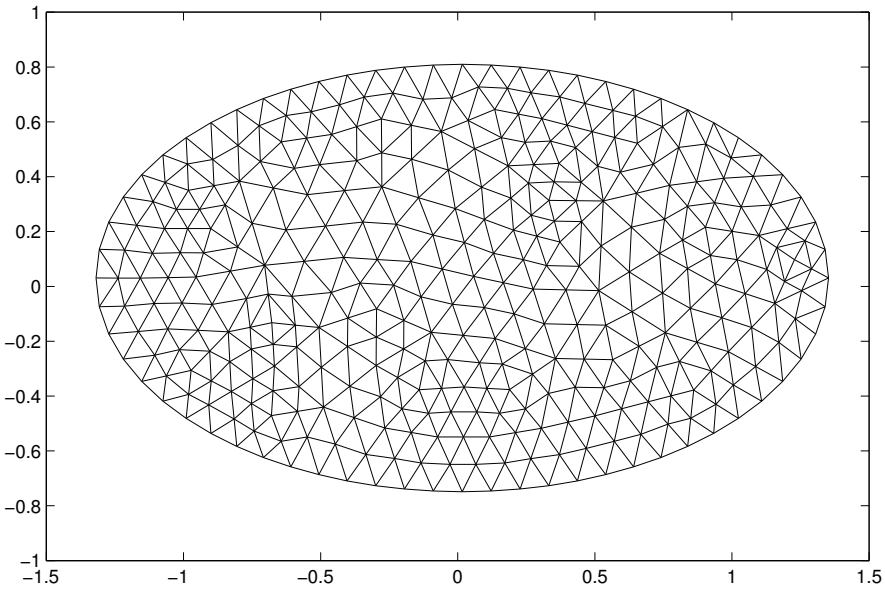


**Fig. 13.4** Maillage plus fin créé par `pdetool`

valles. Chacun de ces éléments est polynomial sur un intervalle et identiquement nul sur les autres.)

La figure 13.5 illustre un cas 2D où les éléments sont des triangles sur un maillage triangulaire. Dans cette configuration simple, l'approximation de la solution sur un triangle donné du maillage est spécifiée par les trois valeurs $Y(t_i,x_i)$ de la solution approchée aux sommets $(t_i,x_i)$ de ce triangle, et la solution approchée à l'intérieur de ce triangle est obtenue par interpolation linéaire. (Des schémas d'interpolation plus complexes peuvent être utilisés pour assurer des transitions plus douces entre éléments finis.) La solution approchée en tout sommet donné $(t_i,x_i)$ doit bien sûr être la même pour tous les triangles du maillage qui ont ce sommet en commun.

*Remarque 13.4.* En multiphysique, les couplages aux interfaces sont pris en compte en imposant des relations entre les quantités physiques concernées aux sommets du maillage situés à ces interfaces. □

*Remarque 13.5.* Comme pour la MDF, la solution approchée obtenue par la MEF est caractérisée par $Y(t,x)$ en des points spécifiques de la région d'intérêt dans l'espace des variables indépendantes $t$ et $x$. Il y a cependant deux différences importantes :

1. ces points sont distribués de façon beaucoup plus flexible,
2. les valeurs prises par la solution approchée sur l'ensemble du domaine d'intérêt peuvent être prises en considération, et pas seulement les valeurs aux points de la grille.

□

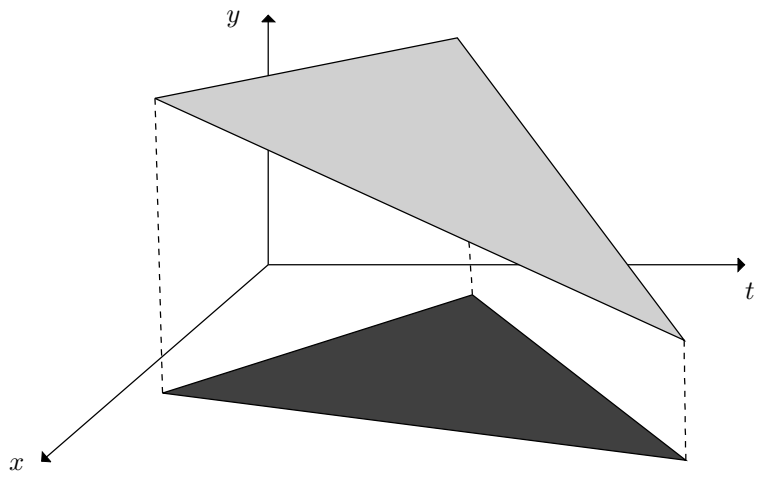


**Fig. 13.5** Un élément fini (en gris clair) et le triangle du maillage qui lui correspond (en gris sombre)

### *13.4.2 Approximation de la solution par des éléments finis*

Soit $y(\mathbf{r})$ la solution de l'EDP, avec $\mathbf{r}$ le vecteur des coordonnées dans l'espace des variables indépendantes, ici $t$ et $x$. Cette solution est approximée par une combinaison linéaire d'éléments finis

$$\widehat{y}_{\mathbf{p}}(\mathbf{r}) = \sum_{k=1}^{K} f_k(\mathbf{r}, Y_{1,k}, Y_{2,k}, Y_{3,k}), \tag{13.39}$$

où $f_k(\mathbf{r},\cdot,\cdot,\cdot)$ vaut zéro hors de la maille associée au $k$-ème élément (supposé triangulaire ici) et où $Y_{i,k}$ est la valeur prise par la solution approchée au $i$-ème sommet de cette maille ($i = 1,2,3$). Les quantités à déterminer sont alors les éléments de $\mathbf{p}$, qui sont certains des $Y_{i,k}$. (Les $Y_{i,k}$ qui correspondent au même point de l'espace des $\mathbf{r}$ doivent être égaux.)

### *13.4.3 Tenir compte de l'EDP*

On peut voir (13.39) comme la définition d'une fonction de splines multivariables qu'on pourrait utiliser pour approximer à peu près n'importe quelle fonction dans l'espace des $\mathbf{r}$. Les mêmes méthodes que celles présentées en section 12.3.4 pour les EDO peuvent maintenant être utilisées pour prendre l'EDP en compte.

Supposons, pour simplifier, que l'EDP à résoudre soit

$$L_{\mathbf{r}}(y) = u(\mathbf{r}), \tag{13.40}$$

où $L(\cdot)$ est un opérateur différentiel linéaire, où $L_{\mathbf{r}}(y)$ est la valeur prise par $L(y)$ en $\mathbf{r}$ et où $u(\mathbf{r})$ est une fonction d'entrée connue. Supposons aussi que la solution $y(\mathbf{r})$ soit à calculer pour des conditions aux limites de Dirichlet sur $\partial\mathbb{D}$, avec $\mathbb{D}$ un domaine dans l'espace des $\mathbf{r}$.

Pour tenir compte de ces conditions aux limites, réécrivons (13.39) comme

$$\widehat{y}_{\mathbf{p}}(\mathbf{r}) = \boldsymbol{\Phi}^{\mathrm{T}}(\mathbf{r})\mathbf{p} + \phi_0(\mathbf{r}), \tag{13.41}$$

où $\phi_0(\cdot)$ satisfait les conditions aux limites, où

$$\boldsymbol{\Phi}(\mathbf{r}) = \mathbf{0}, \quad \forall \mathbf{r} \in \partial\mathbb{D}, \tag{13.42}$$

et où $\mathbf{p}$ correspond maintenant aux paramètres nécessaires pour spécifier la solution une fois que les conditions aux limites ont été prises en compte par $\phi_0(\cdot)$.

Injectons la solution approchée (13.41) dans (13.40) pour définir le résidu

$$\varepsilon_{\mathbf{p}}(\mathbf{r}) = L_{\mathbf{r}}\left(\widehat{y}_{\mathbf{p}}(\mathbf{r})\right) - u(\mathbf{r}), \tag{13.43}$$

qui est affine en $\mathbf{p}$. On peut utiliser les mêmes approches de projection qu'en section 12.3.4 pour rendre ces résidus petits.

#### 13.4.3.1 Collocation

La collocation est la plus simple de ces approches. Comme en section 12.3.4.1, elle impose

$$\varepsilon_{\mathbf{p}}(\mathbf{r}_i) = 0, \quad i = 1, \cdots, \dim \mathbf{p}, \tag{13.44}$$

où les $\mathbf{r}_i$ sont les points de collocation. Ceci se traduit par un système d'équations linéaires à résoudre pour trouver $\mathbf{p}$.

#### 13.4.3.2 Méthodes de Ritz-Galerkin

Avec les méthodes de Ritz-Galerkin, comme en section 12.3.4.2, $\mathbf{p}$ est obtenu en résolvant le système linéaire

$$\int_{\mathbb{D}} \varepsilon_{\mathbf{p}}(\mathbf{r})\varphi_i(\mathbf{r})\mathrm{d}\mathbf{r} = 0, \quad i = 1, \cdots, \dim \mathbf{p}, \tag{13.45}$$

où $\varphi_i(\mathbf{r})$ est une fonction de test, qui peut être le $i$-ème élément de $\boldsymbol{\Phi}(\mathbf{r})$. On retrouve la collocation en remplaçant $\varphi_i(\mathbf{r})$ dans (13.45) par $\delta(\mathbf{r} - \mathbf{r}_i)$, où $\delta(\cdot)$ est la mesure de Dirac.

#### 13.4.3.3 Moindres carrés

Comme en section 12.3.4.3, on peut aussi minimiser une fonction de coût quadratique et choisir

$$\widehat{\mathbf{p}} = \arg\min_{\mathbf{p}} \int_{\mathbb{D}} \varepsilon_{\mathbf{p}}^2(\mathbf{r}) \mathrm{d}\mathbf{r}. \tag{13.46}$$

Comme $\varepsilon_{\mathbf{p}}(\mathbf{r})$ est affine en $\mathbf{p}$, on peut à nouveau utiliser les moindres carrés linéaires. Les conditions nécessaires d'optimalité du premier ordre se traduisent alors en un système d'équations linéaires que $\widehat{\mathbf{p}}$ doit satisfaire.

*Remarque 13.6.* Pour les EDP linéaires, chacune des trois approches de la section 13.4.3 produit un système d'équations linéaires à résoudre pour trouver $\mathbf{p}$. La matrice de ce système sera creuse car chaque composante de $\mathbf{p}$ est associée à un très petit nombre d'éléments, mais elle pourra avoir des composantes non nulles loin de sa diagonale principale. Là encore, une réindexation pourra s'avérer nécessaire pour éviter un ralentissement potentiellement important des calculs.

Quand l'EDP est non linéaire, les méthodes de collocation et de Ritz-Galerkin requièrent la résolution d'un système d'équations non linéaires, tandis que l'optimisation des moindres carrés est menée à bien par programmation non linéaire. □

## 13.5 Exemple MATLAB

Une corde vibrante sans raideur de longueur $L$ satisfait

$$\rho y_{tt} = T y_{xx}, \tag{13.47}$$

où

— $y(x,t)$ est son élongation au point $x$ et à l'instant $t$,
— $\rho$ est sa densité linéique,
— $T$ est sa tension.

La corde est attachée en ses deux extrémités, de sorte que

$$y(0,t) = y(L,t) = 0. \tag{13.48}$$

Sa forme à $t = 0$ est donnée par

$$y(x,0) = \sin(\pi x) \quad \forall x \in [0,L], \tag{13.49}$$

et elle ne bouge pas, de sorte que

$$y_t(x,0) = 0. \tag{13.50}$$

Définissons une grille régulière sur $[0,t_{\max}] \times [0,L]$, telle que (13.21) et (13.22) soient satisfaites et notons $Y_{m,l}$ l'approximation de $y(x_m,t_l)$. L'utilisation de différences centrées du second ordre (6.75),

$$y_{tt}(x_i,t_n) \approx \frac{Y(i,n+1)-2Y(i,n)+Y(i,n-1)}{h_t^2} \tag{13.51}$$

et

$$y_{xx}(x_i,t_n) \approx \frac{Y(i+1,n)-2Y(i,n)+Y(i-1,n)}{h_x^2}, \tag{13.52}$$

conduit à remplacer (13.47) par la récurrence

$$\frac{Y(i,n+1)-2Y(i,n)+Y(i,n-1)}{h_t^2} = \frac{T}{\rho}\frac{Y(i+1,n)-2Y(i,n)+Y(i-1,n)}{h_x^2}. \tag{13.53}$$

Avec

$$R = \frac{Th_t^2}{\rho h_x^2}, \tag{13.54}$$

cette récurrence devient

$$Y(i,n+1)+Y(i,n-1)-RY(i+1,n)-2(1-R)Y(i,n)-RY(i-1,n)=0. \tag{13.55}$$

L'équation (13.49) se traduit par

$$Y(i,1) = \sin(\pi(i-1)h_x), \tag{13.56}$$

et (13.50) par

$$Y(i,2) = Y(i,1). \tag{13.57}$$

Les valeurs de la solution approchée pour $y$ en tous les points de la grille sont empilés dans un vecteur $\mathbf{z}$ qui satisfait un système linéaire $\mathbf{Az} = \mathbf{b}$, où les contenus de $\mathbf{A}$ et $\mathbf{b}$ sont spécifiés par (13.55) et les conditions aux limites. Après avoir évalué $\mathbf{z}$, il faut le dépiler pour visualiser la solution. Tout ceci est réalisé par le script suivant, qui produit les figures 13.6 and 13.7. La fonction `condest` fournit une estimée grossière du conditionnement de $\mathbf{A}$ pour la norme 1, approximativement égale à 5000 ; le problème est donc bien conditionné.

```
clear all

% Paramètres de la corde
L = 1;            % Longueur
T = 4;            % Tension
Rho = 1;          % Densité linéique

% Paramètres de discrétisation
TimeMax = 1;            % Horizon temporel
Nx = 50;                % Nombre de pas d'espace
Nt = 100;               % Nombre de pas de temps
hx = L/Nx;              % Taille du pas d'espace
ht = TimeMax/Nt;        % Taille du pas de temps
```

```
% Création de la matrice creuse A et du vecteur creux b
% pleins de zéros
SizeA = (Nx+1)*(Nt+1);
A = sparse(1:SizeA,1:SizeA,0);
b = sparse(1:SizeA,1,0);

% Remplissage de A et b (les indices MATLAB
% ne peuvent pas être nuls)
R = (T/Rho)*(ht/hx)^2;
Row = 0;
for i=0:Nx,
    Column=i+1;
    Row=Row+1;
    A(Row,Column)=1;
    b(Row)=sin(pi*i*hx/L);
end
for i=0:Nx,
    DeltaCol=i+1;
    Row=Row+1;
    A(Row,(Nx+1)+DeltaCol)=1;
    b(Row)=sin(pi*i*hx/L);
end
for n=1:Nt-1,
    DeltaCol=1;
    Row = Row+1;
    A(Row,(n+1)*(Nx+1)+DeltaCol)=1;
    for i=1:Nx-1
        DeltaCol=i+1;
        Row = Row+1;
        A(Row,n*(Nx+1)+DeltaCol)=-2*(1-R);
        A(Row,n*(Nx+1)+DeltaCol-1)=-R;
        A(Row,n*(Nx+1)+DeltaCol+1)=-R;
        A(Row,(n+1)*(Nx+1)+DeltaCol)=1;
        A(Row,(n-1)*(Nx+1)+DeltaCol)=1;
    end
    i=Nx; DeltaCol=i+1;
    Row=Row+1;
    A(Row,(n+1)*(Nx+1)+DeltaCol)=1;
end

% Calcul d'une borne inférieure
% de Cond(A) pour la norme 1
ConditionNumber=condest(A)

% Calcul de z par résolution des équations linéaires
```

```
Z=A\b;

% Dépilage de z dans Y
for i=0:Nx,
    Delta=i+1;
    for n=0:Nt,
        ind_n=n+1;
        Y(Delta,ind_n)=Z(Delta+n*(Nx+1));
    end
end

% Tracé des résultats en 2D
figure;
for n=0:Nt
    ind_n = n+1;
    plot([0:Nx]*hx,Y(1:Nx+1,ind_n)); hold on
end
xlabel('Location')
ylabel('Elongation')

% Tracé des résultats en 3D
figure;
surf([0:Nt]*ht,[0:Nx]*hx,Y);
colormap(gray)
xlabel('Time')
ylabel('Location')
zlabel('Elongation')
```

## 13.6 En résumé

— Contrairement aux EDO, les EDP ont plusieurs variables indépendantes.
— L'étude des EDP est beaucoup plus complexe que celle des EDO.
— Comme pour les EDO, il faut des conditions aux limites pour spécifier les solutions des EDP.
— La MDF pour les EDP est fondée sur les mêmes principes que pour les EDO.
— La MDF explicite calcule les solutions des EDP par récurrence à partir de profils spécifiés par les conditions aux limites. Elle ne s'applique pas toujours et les erreurs commises sur les pas passés ont des conséquences sur les pas futurs.
— La MDF implicite implique la résolution de systèmes d'équations linéaires (grands et creux). Elle évite les erreurs cumulatives de la MDF explicite.

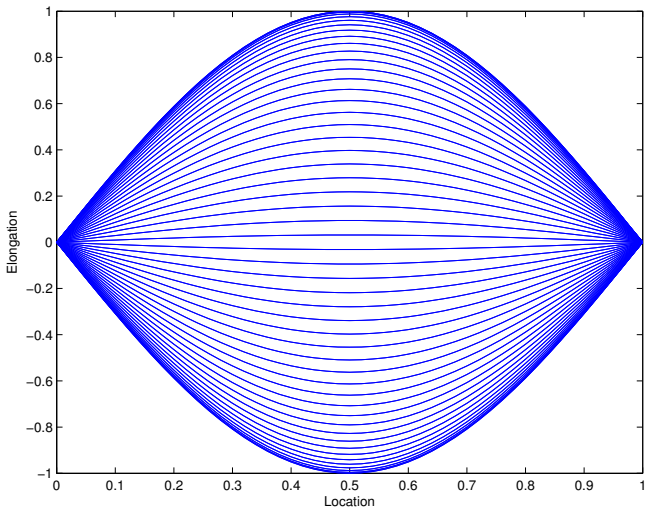


**Fig. 13.6** Visualisation 2D de la solution de l'exemple de la corde vibrante par la MDF

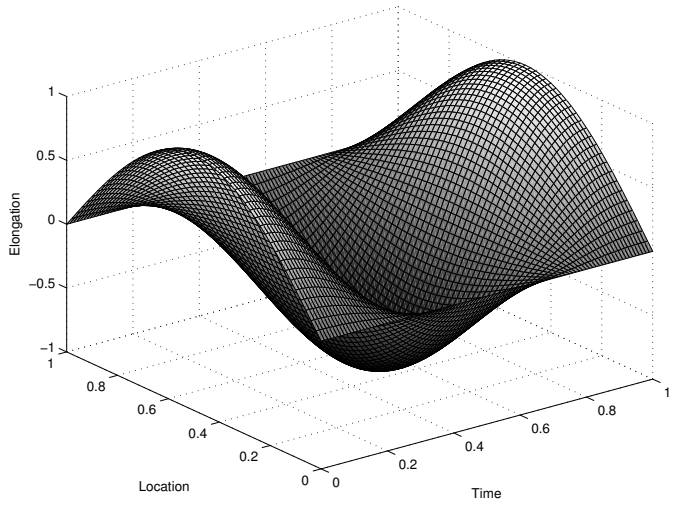


**Fig. 13.7** Visualisation 3D de la solution de l'exemple de la corde vibrante par la MDF

— La MEF est plus flexible que la MDF en ce qui concerne les conditions aux limites. Elle implique un maillage (automatique) et une approximation de dimension finie de la solution.

— Les principes de base des approches de collocation, de Ritz-Galerkin et des moindres carrés pour la résolution des EDP et des EDO sont similaires.

# Chapitre 14
# Évaluer la précision des résultats

## 14.1 Introduction

Ce chapitre traite principalement de méthodes utilisant l'ordinateur lui-même pour évaluer l'effet des *erreurs dues aux arrondis* sur la précision des résultats numériques obtenus par des calculs en virgule flottante. Il aborde aussi brièvement l'évaluation de l'effet des *erreurs de méthode*. (Voir aussi à ce sujet les sections 6.2.1.5, 12.2.4.2 et 12.2.4.3.) La section 14.2 distingue les types d'algorithmes qui seront considérés. La section 14.3 décrit la représentation à virgule flottante des nombres réels et les modes d'arrondi disponibles dans la norme IEEE 754, suivie par la plupart des ordinateurs actuels. L'effet cumulatif des erreurs dues aux arrondis est étudié en section 14.4. Les principales classes de méthodes disponibles pour quantifier les erreurs numériques sont décrites en section 14.5. La section 14.5.2.2 mérite une mention spéciale, car elle décrit une approche particulièrement simple mais potentiellement très utile. La section 14.6 décrit de façon plus détaillée une méthode pour évaluer le nombre de chiffres décimaux significatifs dans un résultat en virgule flottante. Bien qu'elle ait été proposée plus tôt, on peut voir cette méthode comme un raffinement de celle de la section 14.5.2.2.

## 14.2 Types d'algorithmes numériques

On peut distinguer trois types d'algorithmes numériques [182], à savoir les algorithmes finis exacts, itératifs exacts, et approximatifs. Chacun requiert une analyse d'erreur spécifique, voir la section 14.6. Quand l'algorithme est vérifiable, ceci joue aussi un rôle important.

### 14.2.1 Algorithmes vérifiables

Un algorithme est *vérifiable* s'il existe des tests de la validité des solutions qu'il fournit. Si, par exemple, on cherche la solution $\mathbf{x}$ d'un système d'équations linéaires $\mathbf{Ax} = \mathbf{b}$ et si la solution proposée par l'algorithme est $\widehat{\mathbf{x}}$, alors on peut tester à quel point $\mathbf{A}\widehat{\mathbf{x}} - \mathbf{b}$ est proche de $\mathbf{0}$.

La vérification est parfois partielle. Supposons, par exemple, que $\mathbf{x}^k$ soit l'estimée à l'itération $k$ d'un minimiseur sans contrainte d'une fonction de coût différentiable $J(\cdot)$. Il est alors possible d'exploiter la condition nécessaire $\mathbf{g}(\widehat{\mathbf{x}}) = \mathbf{0}$ pour que $\widehat{\mathbf{x}}$ soit un minimiseur, où $\mathbf{g}(\widehat{\mathbf{x}})$ est le gradient de $J(\cdot)$ évalué en $\widehat{\mathbf{x}}$ (voir la section 9.1). On peut ainsi évaluer à quel point $\mathbf{g}(\mathbf{x}^k)$ est proche de $\mathbf{0}$. Rappelons que le fait que $\mathbf{g}(\widehat{\mathbf{x}})$ soit égal à $\mathbf{0}$ ne garantit pas que $\widehat{\mathbf{x}}$ soit un minimiseur et encore moins un minimiseur global, à moins que la fonction de coût n'ait d'autres propriétés (comme d'être convexe).

### 14.2.2 Algorithmes finis exacts

La version mathématique d'un algorithme fini exact produit un résultat exact en un nombre fini d'opérations. L'algèbre linéaire est un fournisseur important de ce type d'algorithme. La seule source d'erreur numérique est alors le passage des nombres réels aux nombres à virgule flottante. Quand il existe plusieurs algorithmes finis exacts pour résoudre le même problème, ils donnent (par définition) la même solution *mathématique*. Ceci n'est plus vrai quand ces algorithmes sont mis en œuvre en utilisant des nombres à virgule flottante, et l'effet cumulatif des erreurs dues aux arrondis sur la solution *numérique* peut beaucoup dépendre de l'algorithme utilisé. C'est particulièrement évident pour les algorithmes qui contiennent des branchements conditionnels, puisque des erreurs sur les conditions de ces branchements peuvent avoir des conséquences catastrophiques.

### 14.2.3 Algorithmes itératifs exacts

La version mathématique d'un algorithme itératif exact produit un résultat exact $\mathbf{x}$ comme la limite d'une séquence infinie calculant $\mathbf{x}^{k+1} = \mathbf{f}(\mathbf{x}^k)$. Certains algorithmes itératifs exacts ne sont pas vérifiables. Une mise en œuvre à virgule flottante d'un algorithme itératif évaluant une série est incapable, par exemple, de vérifier que cette série converge.

Puisqu'on ne peut pas en pratique effectuer une séquence infinie de calculs, on s'arrête après un nombre fini d'itérations, ce qui induit une *erreur de méthode*. Cette erreur de méthode doit être contrôlée par des règles d'arrêt appropriées (voir les sections 7.6 et 9.3.4.8). On peut, par exemple, utiliser la condition absolue

$$||\mathbf{x}^k - \mathbf{x}^{k-1}|| < \delta \tag{14.1}$$

ou la condition relative

$$||\mathbf{x}^k - \mathbf{x}^{k-1}|| < \delta ||\mathbf{x}^{k-1}||. \tag{14.2}$$

Aucun de ces tests n'est sans défaut, comme illustré par l'exemple qui suit.

*Exemple 14.1.* Si une condition absolue telle que (14.1) est utilisée pour évaluer la limite quand $k$ tend vers l'infini de $x_k$ calculé par la récurrence

$$x_{k+1} = x_k + \frac{1}{k+1}, \quad x_1 = 1, \tag{14.3}$$

alors on obtiendra un résultat fini bien que la série *diverge*.

Si une condition relative telle que (14.2) est utilisée pour évaluer $x_N = N$, tel que calculé par la récurrence

$$x_{k+1} = x_k + 1, \quad k = 1, \cdots, N-1, \tag{14.4}$$

commencée en $x_1 = 1$, alors la sommation sera arrêtée trop tôt si $N$ est suffisamment grand. □

Pour les algorithmes vérifiables, des conditions d'arrêt supplémentaires sont disponibles. Si, par exemple, $\mathbf{g}(\cdot)$ est la fonction gradient associée à une fonction de coût $J(\cdot)$, alors on peut utiliser la condition d'arrêt

$$||\mathbf{g}(\mathbf{x}^k)|| < \delta \quad \text{ou} \quad ||\mathbf{g}(\mathbf{x}^k)|| < \delta ||\mathbf{g}(\mathbf{x}^0)|| \tag{14.5}$$

pour la minimisation de $J(\mathbf{x})$ en l'absence de contrainte.

Dans chacune de ces conditions d'arrêt, $\delta > 0$ est un seuil à choisir par l'utilisateur, et la valeur donnée à $\delta$ est critique. Trop petite, elle induit des itérations inutiles, qui peuvent même être nuisibles si les arrondis forcent l'approximation à dériver en s'écartant de la solution. Trop grande, elle conduit à une approximation moins bonne que ce qui aurait été possible. Les sections 14.5.2.2 et 14.6 fourniront des outils qui permettent d'arrêter quand une estimée de la précision avec laquelle $\mathbf{g}(\mathbf{x}^k)$ est évalué devient trop petite.

*Remarque 14.1.* Pour de nombreux algorithmes itératifs, le choix d'une approximation initiale $\mathbf{x}^0$ de la solution est aussi critique. □

### *14.2.4 Algorithmes approximatifs*

Un algorithme approximatif induit une *erreur de méthode*. L'existence d'une telle erreur ne signifie pas que l'algorithme ne devrait pas être utilisé, mais l'effet de cette erreur doit être pris en compte, au même titre que celui des erreurs dues aux arrondis. La discrétisation et la troncature des séries de Taylor sont d'important

fournisseurs d'erreurs de méthode, par exemple quand on approxime des dérivées par des différences finies. Il faut alors choisir une taille de pas. Typiquement, l'erreur de méthode décroît quand la taille du pas décroît tandis que les erreurs dues aux arrondis croissent, ce qui impose un compromis.

*Exemple 14.2.* Considérons l'évaluation de la dérivée première de $f(x) = x^3$ en $x = 2$ avec une différence avant du premier ordre. L'erreur globale résultant des effets combinés des erreurs de méthode et de celles dues aux arrondis est facile à évaluer puisque le résultat exact est $\dot{f}(x) = 3x^2$, et nous pouvons étudier son évolution en fonction de la valeur de la taille $h$ du pas. Le script

```
x = 2;
F = x^3;
TrueDotF = 3*x^2;
i = -20:0;
h = 10.^i;
% Différence avant du premier ordre
NumDotF = ((x+h).^3-F)./h;
AbsErr = abs(TrueDotF - NumDotF);
MethodErr = 3*x*h;
loglog(h,AbsErr,'k-s');
hold on
loglog(h,MethodErr,'k-.');
xlabel('Step-size h (in log scale)')
ylabel('Absolute errors (in log scale)')
```

produit la figure 14.1, qui illustre cette nécessité d'un compromis. La courbe en trait plein interpole les valeurs absolues prises par l'erreur globale pour diverses valeurs de $h$. La ligne en traits mixtes correspond au seul effet de l'erreur de méthode, tel qu'estimé à partir du premier terme négligé dans (6.55), qui est égal à $\ddot{f}(x)h/2 = 3xh$. Quand $h$ est trop petit, les erreurs dues aux arrondis dominent, tandis que quand $h$ est trop grand c'est l'erreur de méthode. □

Idéalement, on devrait choisir $h$ pour minimiser une mesure de l'erreur globale sur le résultat final, mais c'est en général impossible, car l'erreur de méthode ne peut pas être évaluée avec précision. (Autrement, on préférerait la soustraire du résultat numérique pour obtenir un algorithme exact.) Des estimées grossières d'erreurs de méthode peuvent toutefois être obtenues, par exemple en effectuant les mêmes calculs avec plusieurs tailles de pas ou plusieurs ordres de méthodes (voir les sections 6.2.1.5, 12.2.4 et 12.2.4.3). Des bornes strictes des erreurs de méthode peuvent être calculées grâce à l'analyse par intervalles (voir la remarque 14.6).

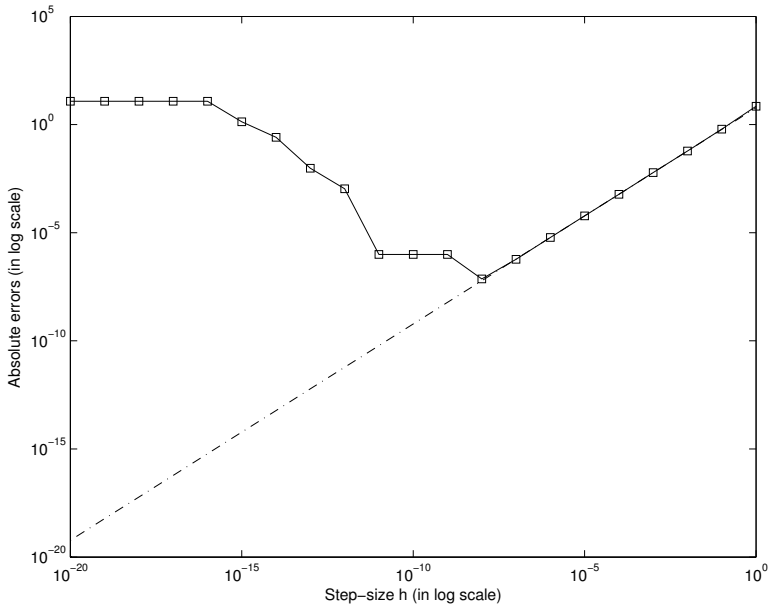


**Fig. 14.1** Nécessité d'un compromis ; courbe en trait plein : erreur globale, ligne en traits mixtes : erreur de méthode

## 14.3 Arrondi

### *14.3.1 Nombres réels et à virgule flottante*

Tout nombre réel $x$ peut s'écrire

$$x = s \cdot m \cdot b^e, \tag{14.6}$$

où $b$ est la *base* (qui appartient à l'ensemble $\mathbb{N}$ des entiers positifs), $e$ est l'*exposant* (qui appartient à l'ensemble $\mathbb{Z}$ des entiers relatifs), $s \in \{-1,+1\}$ est le *signe* et $m$ est la *mantisse*

$$m = \sum_{i=0}^{\infty} a_i b^{-i}, \quad \text{avec} \quad a_i \in \{0,1,\cdots,b-1\}. \tag{14.7}$$

Tout nombre réel non nul a une représentation *normalisée* où $m \in [1,b]$, telle que le triplet $\{s,m,e\}$ est unique.

Une telle représentation ne peut pas être utilisée sur un ordinateur à mémoire finie, et une représentation à virgule flottante avec un nombre de bits fini (et constant) est en général employée à sa place [76].

*Remarque 14.2.* Les nombres à virgule flottante ne sont pas forcément les meilleurs substituts pour les réels. Si la plage de valeurs des réels intervenant dans un calcul donné est suffisamment réduite (par exemple parce qu'une mise à l'échelle a été effectuée), alors on peut préférer calculer avec des entiers ou des rapports d'entiers. Les systèmes de calcul formel comme MAPLE utilisent aussi des rapports d'entiers pour des calculs numériques en précision infinie, avec les entiers représentés exactement par des mots binaires de longueur variable. □

Le fait de remplacer les réels par des nombres à virgule flottante induit des erreurs sur les résultats de calculs numériques, erreurs dont il convient de minimiser les conséquences. Dans ce qui suit, les minuscules en italique sont utilisées pour les réels, et les majuscules en italique pour leurs représentations à virgule flottante.

Soit $\mathbb{F}$ l'ensemble de tous les nombres à virgule flottante de la représentation considérée. On est conduit à remplacer $x \in \mathbb{R}$ par $X \in \mathbb{F}$, avec

$$X = \mathrm{fl}(x) = S \cdot M \cdot b^E. \tag{14.8}$$

Si une représentation normalisée est utilisée pour $x$ et $X$, et pourvu que la base $b$ soit la même, on devrait avoir $S = s$ et $E = e$, mais les calculs précédents peuvent être si faux que $E$ diffère de $e$, ou même $S$ de $s$.

Les résultats sont en général présentés en utilisant une représentation décimale ($b = 10$), mais la représentation des nombres flottants dans l'ordinateur est binaire ($b = 2$), de sorte que

$$M = \sum_{i=0}^{p} A_i \cdot 2^{-i}, \tag{14.9}$$

où $A_i \in \{0,1\},\ i = 0, \cdots, p$, et où $p$ est un entier positif fini. $E$ est en général codé en utilisant $(q+1)$ digits binaires $B_i$, avec un biais qui permet de coder tous les exposants (positifs comme négatifs) par des entiers positifs.

### *14.3.2 Norme IEEE 754*

La plupart des ordinateurs actuels utilisent une représentation binaire à virgule flottante normalisée des réels spécifiée par la norme IEEE 754, mise à jour en 2008 [112]. La normalisation implique que $A_0$, le bit le plus à gauche de $M$, est toujours égal à 1. Il n'est donc pas utile de le stocker (on l'appelle le *bit caché*), pourvu que zéro soit traité comme un cas particulier. Deux formats principaux sont disponibles :

— les nombres à virgule flottante en *simple précision* (ou *flottants*), codés sur 32 bits, sont composés d'un bit de signe, de huit bits pour l'exposant ($q = 7$) et de 23 bits pour la mantisse (plus le bit caché, de sorte que $p = 23$) ; ce format maintenant désuet correspond approximativement à sept digits décimaux significatifs et à des nombres dont la valeur absolue peut aller de $10^{-38}$ à $10^{38}$ ;

— les nombres à virgule flottante en *double précision* (ou *doubles*), codés sur 64 bits, sont composés d'un bit de signe, de onze bits pour l'exposant ($q = 10$) et de 52 bits pour la mantisse (plus le bit caché, de sorte que $p = 52$) ; ce format beaucoup plus couramment utilisé correspond approximativement à seize digits décimaux significatifs et à des nombres dont la valeur absolue peut aller de $10^{-308}$ à $10^{308}$. C'est l'option par défaut en MATLAB.

Le signe $S$ est codé sur un bit, qui prend la valeur zéro si $S = +1$ et un si $S = -1$.

Certains nombres reçoivent un traitement particulier. *Zéro* a deux représentations à virgule flottante : $+0$ et $-0$, avec tous les bits de l'exposant et de la mantisse égaux à zéro. Quand la magnitude de $x$ devient si petite qu'il serait arrondi à zéro si une représentation normalisée était utilisée, des nombres *sous-normaux* sont utilisés comme support d'un *underflow* graduel. Quand la magnitude de $x$ devient si grande qu'il y a *overflow*, $X$ est pris égal à $+\infty$ ou $-\infty$. Si une opération invalide est effectuée, son résultat est NaN, l'acronyme de *Not a Number*. Ceci permet de continuer les calculs tout en indiquant qu'un problème a été rencontré. Notons que l'affirmation NaN = NaN est *fausse*, tandis que l'affirmation $+0 = -0$ est *vraie*.

*Remarque 14.3.* Les nombres à virgule flottante ainsi créés ne sont pas régulièrement espacés sur la droite réelle, car c'est la distance *relative* entre deux doubles consécutifs de même signe qui est constante. La distance entre zéro et le plus petit double positif se révèle beaucoup plus grande que la distance entre ce double et celui immédiatement au dessus, ce qui est l'une des raisons pour l'introduction des nombres sous-normaux. □

### *14.3.3 Erreurs dues aux arrondis*

Le fait de remplacer $x$ par $X$ se traduit presque toujours par une erreur, puisque $\mathbb{F} \neq \mathbb{R}$. Parmi les conséquences de cette substitution, il y a la perte de la notion de continuité ($\mathbb{F}$ est un ensemble discret) et de l'associativité et de la commutativité de certaines opérations.

*Exemple 14.3.* Avec des doubles à la norme IEEE 754, si $x = 10^{25}$ alors

$$(-X + X) + 1 = 1 \neq -X + (X + 1) = 0. \tag{14.10}$$

De même, si $x = 10^{25}$ et $y = 10^{-25}$ alors

$$\frac{(X + Y) - X}{Y} = 0 \neq \frac{(X - X) + Y}{Y} = 1. \tag{14.11}$$

□

Les résultats peuvent donc dépendre de l'ordre dans lequel les calculs sont conduits. Pire, certains compilateurs éliminent les parenthèses qu'ils jugent superflues, de sorte qu'on peut ne même pas savoir ce que cet ordre sera...

### 14.3.4 Modes d'arrondi

IEEE 754 définit quatre modes d'arrondi dirigé :
— vers 0,
— vers le flottant ou le double le plus proche,
— vers $+\infty$,
— vers $-\infty$.
Ces modes spécifient la direction à suivre pour remplacer $x$ par le premier $X$ rencontré. On peut les utiliser pour évaluer l'effet des arrondis sur les résultats de calculs numériques, voir la section 14.5.2.2.

### 14.3.5 Bornes sur les erreurs d'arrondi

Quel que soit le mode d'arrondi, une borne supérieure de l'erreur relative due au fait d'arrondir un réel en un double à la norme IEEE 754 est

$$\text{eps} = 2^{-52} \approx 2.22 \cdot 10^{-16}, \tag{14.12}$$

souvent appelé l'*epsilon machine*.

Pourvu que l'arrondi soit vers le double le plus proche, comme d'habitude, une borne supérieure de l'erreur relative est

$$u = \text{eps}/2 \approx 1.11 \cdot 10^{-16}, \tag{14.13}$$

appelée *unit roundoff*. Pour les opérations arithmétiques de base $\text{op} \in \{+, -, \star, /\}$, le respect de la norme IEEE 754 implique alors que

$$\text{fl}(X \text{op } Y) = (X \text{op } Y)(1+\delta), \quad \text{avec} \quad |\delta| \leqslant u. \tag{14.14}$$

C'est le *modèle standard des opérations arithmétiques* [100], qui peut aussi prendre la forme

$$\text{fl}(X \text{op } Y) = \frac{X \text{op } Y}{1+\delta} \quad \text{avec} \quad |\delta| \leqslant u. \tag{14.15}$$

La situation est beaucoup plus compliquée pour les fonctions transcendantes, et la révision de la norme IEEE 754 se borne à recommander qu'elles soient arrondies correctement, sans l'exiger [166].

L'équation (14.15) implique que

$$|\text{fl}(X \text{op } Y) - (X \text{op } Y)| \leqslant u|\text{fl}(X \text{op } Y)|. \tag{14.16}$$

Il est ainsi facile de calculer une borne pour l'erreur d'arrondi sur $X \text{op } Y$, puisque le *unit roundoff* $u$ est connu et que $\text{fl}(X \text{op } Y)$ est le nombre à virgule flottante fourni par l'ordinateur comme résultat de l'évaluation de $X \text{op } Y$. Les équations (14.15)

et (14.16) sont au cœur de l'analyse d'erreur courante (*running error analysis*), voir la section 14.5.2.4.

## 14.4 Effet cumulatif des erreurs dues aux arrondis

Cette section s'inspire de l'approche probabiliste présentée dans [38, 39, 182]. Les résultats qui y sont résumés jouent un rôle clé dans l'analyse de la méthode CESTAC/CADNA décrite en section 14.6 et mettent en évidence des opérations dangereuses qu'il faut éviter autant que possible.

### *14.4.1 Représentations binaires normalisées*

Tout réel $x$ non nul peut s'écrire suivant la représentation binaire normalisée

$$x = s \cdot m \cdot 2^e. \tag{14.17}$$

Rappelons que quand $x$ n'est pas exactement représentable par un nombre à virgule flottante il est arrondi à $X \in \mathbb{F}$, avec

$$X = \mathrm{fl}(x) = s \cdot M \cdot 2^e, \tag{14.18}$$

où

$$M = \sum_{i=0}^{p} A_i 2^{-i}, \qquad A_i \in \{0,1\}. \tag{14.19}$$

Nous supposons ici que la représentation à virgule flottante est aussi normalisée, de sorte que l'exposant $e$ est le même pour $x$ et $X$. L'erreur due à cet arrondi satisfait alors

$$|X - x| = 2^{e-p} \cdot \alpha, \tag{14.20}$$

avec $p$ le nombre de bits de la mantisse $M$, et $\alpha \in [-0.5, 0.5]$ quand l'arrondi est vers le plus proche et $\alpha \in [-1, 1]$ quand l'arrondi est vers $\pm\infty$ [40]. L'erreur d'arrondi relative $|X - x|/|x|$ est ainsi égale à $2^{-p}$ au plus.

### *14.4.2 Addition (et soustraction)*

Soit $X_3 = s_3 \cdot M_3 \cdot 2^{e_3}$ le résultat à virgule flottante obtenu pour $x_3 = x_1 + x_2$. Le calcul de $X_3$ entraîne en général trois erreurs dues à des arrondis (arrondir $x_1$ pour obtenir $X_1 = s_1 \cdot M_1 \cdot 2^{e_1}$, arrondir $x_2$ pour obtenir $X_2 = s_2 \cdot M_2 \cdot 2^{e_2}$, et arrondir le résultat de l'addition de $X_1$ et $X_2$). On a donc

$$|X_3 - x_3| = |s_1 \cdot 2^{e_1-p} \cdot \alpha_1 + s_2 \cdot 2^{e_2-p} \cdot \alpha_2 + s_3 \cdot 2^{e_3-p} \cdot \alpha_3|. \tag{14.21}$$

Chaque fois que $e_1$ diffère de $e_2$, $X_1$ ou $X_2$ doit être dénormalisé (pour que $X_1$ et $X_2$ aient le même exposant) avant de renormaliser le résultat $X_3$. Il convient de distinguer deux cas :

1. Si $s_1 \cdot s_2 > 0$, ce qui signifie que $X_1$ et $X_2$ sont de même signe, alors l'exposant de $X_3$ est tel que
$$e_3 = \max\{e_1, e_2\} + \delta, \tag{14.22}$$
avec $\delta = 0$ ou 1.
2. Si $s_1 \cdot s_2 < 0$ (comme quand on soustrait deux nombres positifs), alors
$$e_3 = \max\{e_1, e_2\} - k, \tag{14.23}$$
où $k$ est un entier positif. Plus $|X_1|$ est proche de $|X_2|$, et plus $k$ devient grand. C'est une *situation potentiellement catastrophique* ; l'erreur absolue (14.21) est en $O(2^{\max\{e_1,e_2\}-p})$ et l'erreur relative en $O(2^{\max\{e_1,e_2\}-p-e_3}) = O(2^{k-p})$. Ainsi, $k$ digits significatifs ont été perdus.

### *14.4.3 Multiplication (et division)*

Quand $x_3 = x_1 \star x_2$, le même type d'analyse conduit à

$$e_3 = e_1 + e_2 + \delta. \tag{14.24}$$

Quand $x_3 = x_1/x_2$, with $x_2 \neq 0$, il conduit à

$$e_3 = e_1 - e_2 - \delta. \tag{14.25}$$

Dans les deux cas, $\delta = 0$ ou 1.

### *14.4.4 En résumé*

Les équations (14.22), (14.24) et (14.25) suggèrent que l'addition de doubles de même signe, la multiplication de doubles et la division d'un double par un double non nul ne devraient pas conduire à une perte catastrophique de digits significatifs. La soustraction de doubles proches a par contre le potentiel pour un désastre.

On peut parfois reformuler les problèmes à résoudre pour éliminer le risque d'une soustraction mortelle ; voir à titre d'illustration l'exemple 1.2 et la section 14.4.6. Ceci n'est cependant pas toujours possible. Un cas typique est l'évaluation d'une dérivée au moyen d'une approximation par différence finie, par exemple

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}, \tag{14.26}$$

puisque la définition mathématique d'une dérivée requiert que $h$ tende vers zéro. Pour éviter une explosion de l'erreur due à l'arrondi, il faut prendre un $h$ non nul, ce qui induit une erreur de méthode.

### *14.4.5 Perte de précision due à $n$ opérations arithmétiques*

Soir $r$ un résultat mathématique obtenu après $n$ opérations arithmétiques, et $R$ le résultat à virgule flottante normalisé correspondant. Pourvu que les exposants et les signes des résultats intermédiaires ne soient pas affectés par les erreurs dues aux arrondis, on peut montrer [38, 40] que

$$R = r + \sum_{i=1}^{n} g_i \cdot 2^{-p} \cdot \alpha_i + \mathrm{O}(2^{-2p}), \tag{14.27}$$

où les $g_i$ ne dépendent que des données et de l'algorithme et où $\alpha_i \in [-0.5, 0.5]$ si l'arrondi est vers le plus proche et $\alpha_i \in [-1, 1]$ si l'arrondi est vers $\pm\infty$. Le nombre $n_\mathrm{b}$ de digits binaires significatifs dans $R$ est alors tel que

$$n_\mathrm{b} \approx -\log_2 \left| \frac{R-r}{r} \right| = p - \log_2 \left| \sum_{i=1}^{n} g_i \cdot \frac{\alpha_i}{r} \right|. \tag{14.28}$$

Le terme

$$\log_2 \left| \sum_{i=1}^{n} g_i \cdot \frac{\alpha_i}{r} \right|, \tag{14.29}$$

qui approxime la *perte* de précision due aux calculs, ne dépend pas du nombre $p$ des bits dans la mantisse. La précision restante dépend de $p$, bien sûr.

### *14.4.6 Cas particulier du produit scalaire*

Le produit scalaire

$$\mathbf{v}^\mathrm{T}\mathbf{w} = \sum_i v_i w_i \tag{14.30}$$

mérite une attention particulière car ce type d'opération est extrêmement fréquent en calcul matriciel et peut impliquer des différences de termes proches. Ceci a conduit au développement d'outils variés pour maintenir l'erreur commise lors de l'évaluation d'un produit scalaire sous contrôle. On peut citer l'*accumulateur de Kulisch* [132], l'*algorithme de sommation de Kahan* et d'autres algorithmes de sommation compensée [100]. Le prix matériel ou logiciel à payer pour les mettre en œuvre

est cependant significatif, et ces outils ne sont pas toujours utilisables en pratique ni même disponibles.

## 14.5 Classes de méthodes pour évaluer les erreurs numériques

On peut distinguer deux grandes classes de méthodes. La première s'appuie sur une analyse mathématique a priori tandis que la seconde utilise l'ordinateur pour évaluer l'impact de ses erreurs sur ses résultats quand il traite des données numériques spécifiques.

### *14.5.1 Analyse mathématique a priori*

Une référence clé sur l'analyse de la précision d'algorithmes numériques est [100]. L'*analyse progressive* calcule une borne supérieure de la norme de l'erreur entre le résultat mathématique et sa représentation informatique. L'*analyse rétrograde* [253, 250] vise quant à elle à calculer la plus petite perturbation des données d'entrée qui rendrait le résultat mathématique égal à celui fourni par l'ordinateur pour les données d'entrée initiales. Il devient ainsi possible, surtout pour des problèmes d'algèbre linéaire, d'analyser les erreurs dues aux arrondis de façon théorique et de comparer des algorithmes concurrents sur le plan de leur robustesse numérique.

L'analyse mathématique a priori présente cependant deux inconvénients. D'abord, chaque nouvel algorithme doit être soumis à une étude spécifique, qui requiert des compétences sophistiquées. Ensuite, les erreurs dues aux arrondis dépendent des valeurs numériques prises par les données d'entrée du problème spécifique qu'il s'agit de résoudre, qui ne sont pas prises en compte.

### *14.5.2 Analyse par ordinateur*

Chacune des cinq approches considérées dans cette section peut être vue comme une variante a posteriori de l'analyse progressive, où l'on tient compte des valeurs numériques des données traitées.

La première approche étend la notion de conditionnement à des calculs plus généraux que ceux considérés en section 3.3. Nous verrons qu'elle ne répond que partiellement à nos attentes.

La seconde, qui repose sur une suggestion de W. Kahan [119], est de loin la plus simple à mettre en œuvre. Elle s'avère très utile mais, comme la *preuve par neuf* pour la vérification de calculs effectués à la main, elle peut ne pas détecter des erreurs graves. Il en sera de même pour l'approche détaillée en section 14.6.

La troisième, à base d'*analyse par intervalles*, calcule des intervalles qui contiennent *à coup sûr* les résultats mathématiques exacts, de sorte que les erreurs de méthode *et* celles dues aux arrondis sont prises en compte. Le prix à payer est le *conservatisme*, car les intervalles d'incertitude résultants peuvent devenir trop grands pour avoir une utilité quelconque. Des techniques existent pour atténuer la croissance de ces intervalles, mais elles requièrent une adaptation des algorithmes et ne sont pas toujours applicables.

La quatrième approche peut être vue comme une simplification de la troisième, où des bornes d'erreur approximatives sont calculées en propageant les effets des erreurs dues aux arrondis.

La cinquième exploite des perturbations aléatoires des données et des résultats intermédiaires. Sous des hypothèses qui peuvent en partie être vérifiées par la méthode elle-même, elle donne un moyen plus sophistiqué que la seconde approche pour évaluer le nombre de digits décimaux significatifs dans les résultats.

#### 14.5.2.1 Évaluer des conditionnements

La notion de *conditionnement*, introduite en section 3.3 dans le contexte de la résolution des systèmes d'équations linéaires, peut s'étendre aux problèmes non linéaires. Soit $f(\cdot)$ une fonction différentiable de $\mathbb{R}^n$ vers $\mathbb{R}$. Son argument vectoriel $\mathbf{x} \in \mathbb{R}^n$ peut correspondre aux entrées d'un programme, et la valeur prise par $f(\mathbf{x})$ peut correspondre à un résultat mathématique que le programme est chargé de calculer. Pour évaluer les conséquences sur $f(\mathbf{x})$ d'une erreur relative $\varepsilon$ sur chaque composante $x_i$ de $\mathbf{x}$, qui revient à remplacer $x_i$ par $x_i(1+\varepsilon)$, développons $f(\cdot)$ au voisinage de $\mathbf{x}$ pour obtenir

$$f(\widetilde{\mathbf{x}}) = f(\mathbf{x}) + \sum_{i=1}^{n} [\frac{\partial}{\partial x_i} f(\mathbf{x})] \cdot x_i \cdot \varepsilon + \mathrm{O}(\varepsilon^2), \tag{14.31}$$

avec $\widetilde{\mathbf{x}}$ le vecteur d'entrée perturbé.

L'erreur relative sur le résultat $f(\mathbf{x})$ est donc telle que

$$\frac{|f(\widetilde{\mathbf{x}}) - f(\mathbf{x})|}{|f(\mathbf{x})|} \leqslant \frac{\sum_{i=1}^{n} |\frac{\partial}{\partial x_i} f(\mathbf{x})| \cdot |x_i|}{|f(\mathbf{x})|} |\varepsilon| + \mathrm{O}(\varepsilon^2). \tag{14.32}$$

L'approximation au premier ordre du coefficient d'amplification de l'erreur relative est donc donné par le *conditionnement*

$$\kappa = \frac{\sum_{i=1}^{n} |\frac{\partial}{\partial x_i} f(\mathbf{x})| \cdot |x_i|}{|f(\mathbf{x})|}. \tag{14.33}$$

Si $|\mathbf{x}|$ est le vecteur des valeurs absolues des $x_i$, alors

$$\kappa = \frac{|\mathbf{g}(\mathbf{x})|^{\mathrm{T}} \cdot |\mathbf{x}|}{|f(\mathbf{x})|}, \tag{14.34}$$

où $\mathbf{g}(\cdot)$ est la fonction gradient de $f(\cdot)$.

La valeur de $\kappa$ sera grande (mauvaise) si $\mathbf{x}$ est proche d'un zéro de $f(\cdot)$ ou tel que la norme de $\mathbf{g}(\mathbf{x})$ est grande. Des fonctions bien conditionnées (telles que $\kappa$ est petit) peuvent néanmoins être numériquement instables (parce qu'elles impliquent de faire la différence entre des nombres proches). Bon conditionnement et stabilité numérique ne doivent donc pas être confondus.

#### 14.5.2.2 Commuter la direction d'arrondi

Soit $R \in \mathbb{F}$ la représentation informatique d'un résultat mathématique $r \in \mathbb{R}$. Une idée simple pour évaluer la précision de $R$ est de le calculer deux fois, avec des directions d'arrondi opposées, et de comparer les résultats. Notons $R_+$ le résultat informatique obtenu en arrondissant vers $+\infty$ et $R_-$ celui obtenu en arrondissant vers $-\infty$, et montrons comment on peut même obtenir une estimée *grossière* du nombre de chiffres décimaux significatifs.

Le nombre de chiffres décimaux significatifs dans $R$ est le plus grand entier $n_{\mathrm{d}}$ tel que

$$|r - R| \leqslant \frac{|r|}{10^{n_{\mathrm{d}}}}. \tag{14.35}$$

En pratique, $r$ est inconnu (sinon, il ne serait pas nécessaire de calculer $R$). En remplaçant $r$ dans (14.35) par sa moyenne empirique $(R_+ + R_-)/2$ et $|r - R|$ par $|R_+ - R_-|$, on obtient

$$\widehat{n}_{\mathrm{d}} = \log_{10} \left| \frac{R_+ + R_-}{2(R_+ - R_-)} \right|, \tag{14.36}$$

qu'on peut ensuite arrondir à l'entier non négatif le plus proche. des calculs similaires seront effectués en section 14.6 sur la base d'hypothèses statistiques sur les erreurs.

*Remarque 14.4.* L'estimée $\widehat{n}_{\mathrm{d}}$ fournie par (14.36) doit être manipulée avec précaution. Si $R_+$ et $R_-$ sont proches, ceci ne prouve pas qu'ils sont proches de $r$, ne serait-ce que parce que l'arrondi n'est qu'une des sources d'erreurs possibles. Si, par contre, $R_+$ et $R_-$ sont notablement différents, alors les résultats fournis par l'ordinateur doivent être regardés avec méfiance. □

*Remarque 14.5.* Il peut s'avérer difficile d'évaluer $n_{\mathrm{d}}$ par inspection visuelle de $R_+$ et $R_-$. Ainsi, 1.999999991 et 2.000000009 sont-ils très proches bien qu'ils n'aient aucun chiffre en commun, tandis que 1.21 et 1.29 sont moins proches qu'ils ne le semblent visuellement, comme on peut s'en rendre compte en les remplaçant par les approximations à deux chiffres qui en sont les plus proches. □

#### 14.5.2.3 Calculer avec des intervalles

Le calcul par intervalles a plus de deux mille ans. Il a été popularisé en informatique par le travail de R.E. Moore [160, 161, 163]. Dans sa forme de base, il opère sur des intervalles fermés

$$[x] = [x_-, x_+] = \{x \in \mathbb{R} | x_- \leqslant x \leqslant x_+\}, \tag{14.37}$$

où $x_-$ est la borne inférieure de $[x]$ et $x_+$ sa borne supérieure. Les intervalles peuvent donc être caractérisés par des paires de nombres réels $(x_-, x_+)$, tout comme les nombres complexes. Les opérations arithmétiques s'étendent aux intervalles en prenant en compte toutes les valeurs possibles des variables qui appartiennent aux opérandes intervalles. La surcharge d'opérateurs permet d'adapter facilement le sens de ceux-ci quand ils opèrent sur des intervalles. Ainsi, par exemple,

$$[c] = [a] + [b] \tag{14.38}$$

est interprété comme signifiant que

$$c_- = a_- + b_- \quad \text{et} \quad c_+ = a_+ + b_+, \tag{14.39}$$

et

$$[c] = [a] \star [b] \tag{14.40}$$

est interprété comme signifiant que

$$c_- = \min\{a_- b_-, a_- b_+, a_+ b_-, a_+ b_+\} \tag{14.41}$$

et

$$c_+ = \max\{a_- b_-, a_- b_+, a_+ b_-, a_+ b_+\}. \tag{14.42}$$

Le cas de la division est légèrement plus compliqué, car si l'intervalle au dénominateur contient zéro alors le résultat n'est plus un intervalle. Quand on prend son intersection avec un intervalle, ce résultat peut produire deux intervalles au lieu d'un.

L'image d'un intervalle par une fonction monotone est triviale à calculer. Par exemple,

$$\exp([x]) = [\exp(x_-), \exp(x_+)]. \tag{14.43}$$

Il est à peine plus difficile de calculer l'image d'un intervalle par une fonction trigonométrique ou toute autre fonction élémentaire. Tel n'est plus le cas pour une fonction générique $f(\cdot)$, mais n'importe laquelle de ses *fonctions d'inclusion* $[f](\cdot)$ permet de calculer des intervalles qui contiennent à coup sûr l'image de $[x]$ par la fonction initiale, car

$$f([x]) \subset [f]([x]). \tag{14.44}$$

Quand une expression formelle est disponible pour $f(x)$, la *fonction d'inclusion naturelle* $[f]_{\mathrm{n}}([x])$ est obtenue en remplaçant, dans l'expression formelle de $f(\cdot)$,

chaque occurrence de $x$ par $[x]$ et chaque opération ou chaque fonction élémentaire par sa contrepartie pour les intervalles.

*Exemple 14.4.* Si

$$f(x) = (x-1)(x+1), \tag{14.45}$$

alors

$$\begin{aligned} [f]_{\mathrm{n1}}([-1,1]) &= ([-1,1]-[1,1])([-1,1]+[1,1]) \\ &= [-2,0] \star [0,2] \\ &= [-4,4]. \end{aligned} \tag{14.46}$$

En réécrivant $f(x)$ sous la forme

$$f(x) = x^2 - 1, \tag{14.47}$$

et en exploitant le fait que $x^2 \geqslant 0$, nous obtenons aussi

$$[f]_{\mathrm{n2}}([-1,1]) = [-1,1]^2 - [1,1] = [0,1] - [1,1] = [-1,0]. \tag{14.48}$$

On constate que $[f]_{\mathrm{n2}}(\cdot)$ est beaucoup plus précise que $[f]_{\mathrm{n1}}(\cdot)$. C'est même une *fonction d'inclusion minimale*, car

$$f([x]) = [f]_{\mathrm{n2}}([x]). \tag{14.49}$$

Ceci est dû au fait que l'expression formelle de $[f]_{\mathrm{n2}}([x])$ ne contient qu'une occurrence de $[x]$. □

Une illustration caricaturale du pessimisme induit par les occurrences multiples de variables est l'évaluation de

$$f(x) = x - x \tag{14.50}$$

sur l'intervalle $[-1,1]$ en utilisant une fonction d'inclusion naturelle. Comme les deux occurrences de $x$ dans (14.50) sont traitées comme si elles étaient indépendantes,

$$[f]_{\mathrm{n}}([-1,1]) = [-2,2]. \tag{14.51}$$

C'est donc une bonne idée de rechercher des expressions formelles qui minimisent le nombre d'occurrences des variables. De nombreuses autres techniques sont disponibles pour réduire le pessimisme des fonctions d'inclusion.

Le calcul par intervalles s'étend facilement aux vecteurs et matrices d'intervalles. Un *vecteur d'intervalles* (ou *boîte*) $[\mathbf{x}]$ est un produit cartésien d'intervalles, et $[\mathbf{f}]([\mathbf{x}])$ est une fonction d'inclusion pour la fonction vectorielle multivariable $\mathbf{f}(\mathbf{x})$ si elle calcule une boîte $[\mathbf{f}]([\mathbf{x}])$ qui contient l'image de $[\mathbf{x}]$ par $\mathbf{f}(\cdot)$, c'est à dire si

$$\mathbf{f}([\mathbf{x}]) \subset [\mathbf{f}]([\mathbf{x}]). \tag{14.52}$$

Dans la mise en œuvre des intervalles en virgule flottante, l'intervalle réel $[x]$ est remplacé par un intervalle $[X]$ représentable en machine et obtenu par *arrondi extérieur*, c'est à dire que $X_-$ est obtenu en arrondissant $x_-$ vers $-\infty$, et $X_+$ en arrondissant $x_+$ vers $+\infty$. On peut alors remplacer le calcul sur les réels par un calcul sur des intervalles représentables en machine, qui produit des intervalles contenant à coup sûr les résultats qui auraient été obtenus en calculant sur les réels. Cette approche conceptuellement attirante est à peu près aussi vielle que les ordinateurs.

Il devint vite clair, cependant, que l'évaluation de l'impact des erreurs ainsi obtenue pouvait être si pessimiste qu'elle en devenait inutilisable. Ceci ne veut pas dire qu'on ne peut pas employer l'analyse par intervalles, mais le problème à résoudre doit être formulé de façon adéquate et des algorithmes spécifiques doivent être utilisés. Des ingrédients clés de ces algorithmes sont

- l'élimination de boîtes en *prouvant* qu'elles ne contiennent aucune solution,
- la bissection de boîtes sur lesquelles aucune conclusion n'a pu être obtenue, dans une approche de type *diviser pour régner*,
- et la contraction de boîtes qui pourraient contenir des solutions, sans en perdre aucune.

*Exemple 14.5. Élimination*

Supposons que $\mathbf{g}(\mathbf{x})$ soit le gradient d'une fonction de coût à minimiser sans contrainte et que $[\mathbf{g}](\cdot)$ soit une fonction d'inclusion pour $\mathbf{g}(\cdot)$. Si

$$\mathbf{0} \notin [\mathbf{g}]([\mathbf{x}]), \tag{14.53}$$

alors (14.52) implique que

$$\mathbf{0} \notin \mathbf{g}([\mathbf{x}]). \tag{14.54}$$

La condition d'optimalité au premier ordre (9.6) n'est ainsi satisfaite nulle part dans la boîte $[\mathbf{x}]$, qui peut donc être éliminée des recherches ultérieures puisqu'elle ne peut pas contenir de minimiseur sans contrainte. □

*Exemple 14.6. Bissection*

Considérons encore l'exemple 14.5, mais supposons maintenant que

$$\mathbf{0} \in [\mathbf{g}]([\mathbf{x}]), \tag{14.55}$$

ce qui ne permet pas d'éliminer $[\mathbf{x}]$. On peut alors couper $[\mathbf{x}]$ en $[\mathbf{x}_1]$ et $[\mathbf{x}_2]$, et essayer d'éliminer ces boîtes plus petites. Ceci est facilité par le fait que les fonctions d'inclusion deviennent en général moins pessimistes quand la taille de leurs arguments intervalles diminue (jusqu'à ce que l'effet des arrondis extérieurs devienne prédominant). □

La malédiction de la dimension rôde bien sûr derrière la bissection. La contraction, qui permet de réduire la taille de $[\mathbf{x}]$ sans qu'aucune solution ne soit perdue, est donc particulièrement importante quand on traite des problèmes en dimension élevée.

*Exemple 14.7. Contraction*

Soit $f(\cdot)$ une fonction scalaire d'une variable, avec une dérivée première continue sur $[x]$, et soient $x^\star$ et $x_0$ deux points de $[x]$, avec $f(x^\star) = 0$. Le théorème de la valeur moyenne implique qu'il existe $c \in [x]$ tel que

$$\dot{f}(c) = \frac{f(x^\star) - f(x_0)}{x^\star - x_0}. \tag{14.56}$$

Autrement dit,

$$x^\star = x_0 - \frac{f(x_0)}{\dot{f}(c)}. \tag{14.57}$$

Si une fonction d'inclusion $[\dot{f}](\cdot)$ est disponible pour $\dot{f}(\cdot)$, alors

$$x^\star \in x_0 - \frac{f(x_0)}{[\dot{f}]([x])}. \tag{14.58}$$

Comme $x^\star$ appartient aussi à $[x]$, on peut écrire

$$x^\star \in [x] \cap \left( x_0 - \frac{f(x_0)}{[\dot{f}]([x])} \right), \tag{14.59}$$

et le membre de droite de (14.59) peut être beaucoup plus petit que $[x]$. Ceci suggère d'itérer

$$[x_{k+1}] = [x_k] \cap \left( x_k - \frac{f(x_k)}{[\dot{f}]([x_k])} \right), \tag{14.60}$$

où $x_k$ est un point de $[x_k]$, par exemple son centre. Toute solution appartenant à $[x_k]$ appartient aussi à $[x_{k+1}]$.

La méthode de *Newton sur les intervalles* qui en résulte est plus compliquée qu'il ne semble à mettre en œuvre, car l'intervalle $[\dot{f}]([x_k])$ au dénominateur peut contenir zéro, de sorte que $[x_{k+1}]$ peut en fait être constitué de deux intervalles, qui devront alors être traités tous les deux lors de l'itération suivante. La méthode de Newton sur les intervalles peut être étendue à la recherche d'approximations par des boîtes de toutes les solutions de systèmes d'équations non linéaires en plusieurs inconnues [171]. □

*Remarque 14.6.* De façon similaire, les calculs sur les intervalles peuvent servir à obtenir des bornes sur les restes de développements de Taylor, ce qui permet de borner des erreurs de méthode. Considérons, par exemple, le développement de Taylor à l'ordre $k$ d'une fonction scalaire $f(\cdot)$ d'une variable au voisinage de $x_c$

$$f(x) = f(x_c) + \sum_{i=1}^{k} \frac{1}{i!} f^{(i)}(x_c) \cdot (x - x_c)^i + r(x, x_c, \xi), \tag{14.61}$$

où

$$r(x, x_c, \xi) = \frac{1}{(k+1)!} f^{(k+1)}(\xi) \cdot (x - x_c)^{k+1} \tag{14.62}$$

est le *reste de Taylor*. L'équation (14.61) est vraie pour un $\xi$ inconnu dans $[x, x_c]$. Une fonction d'inclusion $[f](\cdot)$ pour $f(\cdot)$ est donc

$$[f]([x]) = f(x_c) + \sum_{i=1}^{k} \frac{1}{i!} f^{(i)}(x_c) \cdot ([x] - x_c)^i + [r]([x], x_c, [x]), \tag{14.63}$$

avec $[r](\cdot,\cdot,\cdot)$ une fonction d'inclusion pour $r(\cdot,\cdot,\cdot)$ et $x_c$ un point quelconque de $[x]$, par exemple son centre. □

Grâce à ces concepts, on peut trouver des solutions approchées mais *garanties* à des problèmes tels que

— trouver *toutes* les solutions d'un système d'équations non linéaires [171],
— caractériser un ensemble défini par des inégalités non linéaires [115],
— trouver *tous* les minimiseurs *globaux* d'une fonction de coût non convexe [191, 96],
— résoudre un problème de Cauchy pour une EDO non linéaire pour laquelle on ne connaît pas de solution explicite [16, 168, 167].

Des applications en ingénierie sont présentées dans [115].

L'analyse par intervalles fait l'hypothèse que les erreurs commises à chaque étape du calcul peuvent être aussi néfastes qu'il est possible. Heureusement, la situation n'est en général pas si grave, car certaines erreurs en compensent d'autres en partie. C'est ce qui motive le remplacement d'une telle analyse dans le pire des cas par une analyse probabiliste des résultats obtenus quand les mêmes calculs sont répétés avec des réalisations différentes des erreurs dues aux arrondis, comme dans la section 14.6.

#### 14.5.2.4 Analyser l'erreur courante

L'analyse de l'erreur courante (*running error analysis*) [252, 100, 258] propage une évaluation de l'effet des erreurs d'arrondi lors des calculs en virgule flottante. Soit $\varepsilon_x$ une borne de l'erreur absolue sur $x$, telle que

$$|X - x| \leqslant \varepsilon_x. \tag{14.64}$$

Quand l'arrondi est vers le double le plus proche, comme il est d'usage, des bornes approchées sur les résultats des opérations arithmétiques sont calculées comme suit :

$$z = x + y \Rightarrow \varepsilon_z = u|\mathrm{fl}(X+Y)| + \varepsilon_x + \varepsilon_y, \tag{14.65}$$
$$z = x - y \Rightarrow \varepsilon_z = u|\mathrm{fl}(X-Y)| + \varepsilon_x + \varepsilon_y, \tag{14.66}$$
$$z = x \star y \Rightarrow \varepsilon_z = u|\mathrm{fl}(X \star Y)| + \varepsilon_x|Y| + \varepsilon_y|X|, \tag{14.67}$$
$$z = x/y \Rightarrow \varepsilon_z = u|\mathrm{fl}(X/Y)| + \frac{\varepsilon_x|Y| + \varepsilon_y|X|}{Y^2}. \tag{14.68}$$

Les premiers termes des membres les plus à droite de (14.65)-(14.68) sont déduits de (14.16). Les termes suivants propagent les erreurs d'entrée vers la sortie tout en

négligeant les produits de termes d'erreur. La méthode est beaucoup plus simple à mettre en œuvre que l'approche par intervalles de la section 14.5.2.3, mais les bornes résultantes sur l'effet des erreurs dues aux arrondis sont approximatives et les erreurs de méthode ne sont pas prises en compte.

#### 14.5.2.5 Perturber aléatoirement les calculs

Cette méthode trouve ses origines dans les travaux de M. La Porte et J. Vignes [133, 236, 238, 237, 182]. Connue initialement sous l'acronyme CESTAC (pour Contrôle et Estimation STochastique des Arrondis de Calcul), elle est maintenant mise en œuvre dans le logiciel CADNA (pour *Control of Accuracy and Debugging for Numerical Applications*), disponible à www-pequan.lip6.fr/cadna/. CESTAC/CADNA, décrite plus en détail dans la section qui suit, peut être vue comme une méthode de Monte-Carlo. Le même calcul est effectué plusieurs fois en tirant au hasard l'erreur due à l'arrondi, et des caractéristiques statistiques de la population des résultats ainsi obtenus sont évaluées. Si les résultats fournis par l'ordinateur varient considérablement à cause de perturbations aussi minuscules, alors ils manquent clairement de crédibilité. De façon plus quantitative, chacun de ces résultats sera fourni avec une estimée de son nombre de chiffres décimaux significatifs.

## 14.6 CESTAC/CADNA

La présentation de la méthode est suivie d'une discussion de ses conditions de validité, qui peuvent être vérifiées en partie par la méthode elle même.

### *14.6.1 Méthode*

Soient $r \in \mathbb{R}$ une quantité à évaluer par un programme et $R_i \in \mathbb{F}$ le résultat à virgule flottante correspondant, tel que fourni par la $i$-ème exécution de ce programme ($i = 1, \cdots, N$). Lors de chaque exécution, le résultat de chaque opération est arrondi aléatoirement vers $+\infty$ ou $-\infty$, avec la même probabilité. Chaque $R_i$ est ainsi une approximation de $r$. L'*hypothèse fondamentale* à la base de CESTAC/CADNA est que ces $R_i$ sont indépendamment et identiquement distribués suivant une loi gaussienne de moyenne $r$.

Soit $\mu$ la moyenne arithmétique des résultats fournis par l'ordinateur en $N$ exécutions

$$\mu = \frac{1}{N} \sum_{i=1}^{N} R_i. \tag{14.69}$$

Comme $N$ est fini, $\mu$ n'est pas égal à $r$, mais en est typiquement plus proche que les $R_i$ (sous l'hypothèse fondamentale, $\mu$ est l'estimée de $r$ au sens du maximum de vraisemblance). Soit $\sigma$ l'écart-type empirique des $R_i$

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(R_i-\mu)^2}, \tag{14.70}$$

qui caractérise la dispersion des $R_i$ autour de leur moyenne. Le *test de Student* permet de calculer un intervalle centré en $\mu$ et ayant une probabilité $\beta$ donnée de contenir $r$

$$\text{Prob}\left[|\mu - r| \leqslant \frac{\tau\sigma}{\sqrt{N}}\right] = \beta. \tag{14.71}$$

Dans (14.71), la valeur de $\tau$ dépend de la valeur de $\beta$ (à choisir par l'utilisateur) et du nombre de degrés de liberté, qui est égal à $N-1$ puisqu'il y a $N$ données $R_i$ liées à $\mu$ par la contrainte d'égalité (14.69). On choisit typiquement $\beta = 0.95$, ce qui revient à accepter de se tromper dans 5% des cas, et $N = 2$ ou 3, pour que le volume des calculs reste gérable. D'après (14.35), le nombre $n_{\rm d}$ des chiffres décimaux significatifs dans $\mu$ est tel que

$$10^{n_{\rm d}} \leqslant \frac{|r|}{|\mu - r|}. \tag{14.72}$$

Remplaçons $|\mu - r|$ par $\tau\sigma/\sqrt{N}$ et $r$ par $\mu$ pour obtenir une estimée de $n_{\rm d}$ comme l'entier non négatif le plus proche de

$$\widehat{n}_{\rm d} = \log_{10}\frac{|\mu|}{\frac{\tau\sigma}{\sqrt{N}}} = \log_{10}\frac{|\mu|}{\sigma} - \log_{10}\frac{\tau}{\sqrt{N}}. \tag{14.73}$$

Pour $\beta = 0.95$, il vient

$$\widehat{n}_{\rm d} \approx \log_{10}\frac{|\mu|}{\sigma} - 0.953 \quad \text{si} \quad N = 2, \tag{14.74}$$

et

$$\widehat{n}_{\rm d} \approx \log_{10}\frac{|\mu|}{\sigma} - 0.395 \quad \text{si} \quad N = 3. \tag{14.75}$$

*Remarque 14.7.* Supposons que $N = 2$ et notons $R_+$ et $R_-$ les résultats des deux exécutions. On a alors

$$\log_{10}\frac{|\mu|}{\sigma} = \log_{10}\frac{|R_+ + R_-|}{|R_+ - R_-|} - \log_{10}\sqrt{2}, \tag{14.76}$$

de sorte que

$$\widehat{n}_{\rm d} \approx \log_{10}\frac{|R_+ + R_-|}{|R_+ - R_-|} - 1.1. \tag{14.77}$$

Ceci est à comparer avec (14.36), qui est tel que

$$\widehat{n}_{\mathrm{d}} \approx \log_{10} \frac{|R_+ + R_-|}{|R_+ - R_-|} - 0.3. \tag{14.78}$$

□

Sur la base de cette analyse, on peut maintenant présenter chaque résultat dans un format qui ne montre que les chiffres jugés significatifs. Un cas particulièrement spectaculaire est quand le nombre de chiffres significatifs devient nul, ce qui revient à dire qu'on ne sait rien du résultat, pas même son signe. Ceci a conduit au concept de *zéro informatique* (ZI) : Le résultat d'un calcul numérique est un ZI s'il est nul ou ne contient aucun chiffre significatif. Un nombre à virgule flottante très grand peut se révéler être un ZI alors qu'un autre de toute petite magnitude peut ne pas en être un.

L'application de cette approche dépend du type d'algorithme considéré, comme défini en section 14.2.

Pour les *algorithmes finis exacts*, CESTAC/CADNA peut fournir chaque résultat avec une estimée de son nombre de chiffres décimaux significatifs. Quand l'algorithme comporte des branchements conditionnels, il faut être très prudent avec l'évaluation par CESTAC/CADNA de la précision des résultats, car les exécutions perturbées peuvent ne pas toutes suivre la même branche du code, ce qui rendrait l'hypothèse d'une distribution gaussienne des résultats particulièrement discutable. Ceci suggère d'analyser non seulement la précision des résultats finaux mais aussi celle de tous les résultats intermédiaires à virgule flottante (ou au moins de ceux impliqués dans les conditions de tests). Ceci peut être fait via deux ou trois exécutions de l'algorithme *en parallèle*. La surcharge d'opérateurs permet d'éviter d'avoir à modifier lourdement le code à tester. Il suffit de déclarer que les variables à surveiller sont de type *stochastique*. Pour plus de détails, voir www.lip6.fr/cadna. Dès qu'un ZI est détecté, tous les résultats des calculs qui suivent doivent être examinés avec soin. On peut même décider d'arrêter là les calculs et de rechercher une formulation alternative du problème.

Pour les *algorithmes itératifs exacts*, CESTAC/CADNA fournit aussi des règles d'arrêt rationnelles. De nombreux algorithmes de ce type sont vérifiables (au moins partiellement) et devraient être stoppés quand une quantité (éventuellement vectorielle) devient nulle. Quand on recherche une racine du système d'équations non linéaires $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, par exemple, cette quantité pourra être $\mathbf{f}(\mathbf{x}^k)$. Quand on recherche un minimiseur sans contrainte d'une fonction de coût différentiable, ce pourra être $\mathbf{g}(\mathbf{x}^k)$, avec $\mathbf{g}(\cdot)$ la fonction gradient de cette fonction de coût. On peut ainsi décider d'arrêter quand les représentations à virgule flottante de toutes les composantes de $\mathbf{f}(\mathbf{x}^k)$ ou $\mathbf{g}(\mathbf{x}^k)$ sont devenues des ZI, c'est à dire qu'elles sont nulles ou ne contiennent plus aucun chiffre significatif. Ceci revient à dire qu'il est devenu impossible de prouver que la solution n'a pas été atteinte compte-tenu de la précision avec laquelle les calculs ont été conduits. Le choix délicat des paramètres de seuil dans les tests d'arrêt est ainsi contourné. Le prix à payer pour évaluer la précision des résultats est une multiplication par deux ou trois du volume des calculs. Ceci semble d'autant plus raisonnable que les algorithmes itératifs sont souvent arrêtés beaucoup plus tôt qu'avec des règles d'arrêt plus traditionnelles, de sorte que le vo-

lume total des calculs peut même décroître. Quand l'algorithme n'est pas vérifiable, il peut rester possible de définir une règle d'arrêt rationnelle. Si, par exemple, on veut calculer

$$S = \lim_{n\to\infty} S_n = \sum_{i=1}^{n} f_i, \tag{14.79}$$

on peut alors s'arrêter quand

$$|S_n - S_{n-1}| = \mathrm{ZI}, \tag{14.80}$$

ce qui revient à dire que l'incrément itératif n'est plus significatif. (Les fonctions transcendantes usuelles *ne sont pas* calculées par des évaluations de séries de ce type, et les procédures utilisées en pratique sont très sophistiquées [165].)

Pour les *algorithmes approximatifs*, on voudrait minimiser l'erreur globale résultant de la combinaison des erreurs de méthode et de celles dues aux arrondis. CESTAC/CADNA peut aider à trouver un bon compromis en contribuant à l'évaluation de l'effet des secondes, pourvu que les effets des premières soient évalués par une autre méthode.

### *14.6.2 Conditions de validité*

Une étude détaillée des conditions sous lesquelles cette approche fournit des résultats fiables est présentée dans [38] et [40] ; voir aussi [182]. Elle est fondée sur (14.27), qui résulte d'une analyse d'erreur progressive au premier ordre, et sur le *théorème de la limite centrale*. Sous sa forme la plus simple, ce théorème dit que la moyenne arithmétique de $n$ variables aléatoires $x_i$ *indépendantes* tend, quand $n$ tend vers l'infini, à être distribués suivant une loi *gaussienne* de moyenne $\mu$ et de variance $\sigma^2/n$, pourvu que les $x_i$ aient la *même moyenne* $\mu$ et la *même variance* $\sigma^2$. Les $x_i$ n'ont pas besoin d'être gaussien pour que ce résultat soit valide.

CESTAC/CADNA arrondit vers $+\infty$ ou $-\infty$ *aléatoirement*, ce qui assure que les $\alpha_i$ dans (14.27) sont approximativement indépendants et distribués uniformément dans $[-1,1]$, bien que les erreurs dues aux arrondis nominales soient déterministes et corrélées. Si aucun des coefficients $g_i$ dans (14.27) n'est de taille beaucoup plus grande que tous les autres et si l'analyse d'erreur au premier ordre demeure valide, alors la population des résultats fournis par l'ordinateur est approximativement gaussienne, de moyenne approximativement égale au résultat mathématique exact pourvu que le nombre d'opérations soit suffisamment grand.

Considérons tout d'abord les conditions sous lesquelles l'approximation (14.27) est valide pour des opérations arithmétiques. Les exposants et les signes des résultats intermédiaires sont supposés ne pas être affectés par les erreurs d'arrondi. En d'autres termes, aucun de ces résultats intermédiaires n'est supposé être un ZI.

Les additions et les soustractions n'introduisent pas de terme d'erreur d'ordre supérieur à un. Pour la multiplication,

$$X_1X_2 = x_1(1+\varepsilon_1)x_2(1+\varepsilon_2) = x_1x_2(1+\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2), \tag{14.81}$$

et $\varepsilon_1\varepsilon_2$, le seul terme d'erreur d'ordre supérieur à un, est négligeable si $\varepsilon_1$ et $\varepsilon_2$ sont petits par rapport à un, c'est à dire si $X_1$ et $X_2$ ne sont pas des ZI. Pour la division

$$\frac{X_1}{X_2} = \frac{x_1(1+\varepsilon_1)}{x_2(1+\varepsilon_2)} = \frac{x_1(1+\varepsilon_1)}{x_2}(1-\varepsilon_2+\varepsilon_2^2-\cdots), \tag{14.82}$$

ce qui démontre l'effet particulièrement catastrophique qu'aurait $\varepsilon_2$ si sa valeur absolue était supérieure à un. Ceci correspondrait à une division par un ZI, une première cause d'échec de l'analyse CESTAC/CADNA.

Une deuxième cause d'échec est quand la plus grande part de l'erreur finale est due à quelques opérations critiques. Tel peut être le cas, par exemple, quand une décision de branchement est prise sur la base du signe d'une quantité qui se révèle être un ZI. Suivant la réalisation des calculs, l'une ou l'autre branche de l'algorithme sera suivi. Les résultats peuvent donc être complètement différents et avoir une distribution multimodale, très loin d'une gaussienne.

Ces considérations suggèrent le conseil suivant.

> Tout résultat intermédiaire qui se révèle être un ZI doit faire douter des estimées des nombres de chiffres significatifs dans les résultats des calculs qui suivent, qui doivent être regardés avec prudence. C'est particulièrement vrai si le ZI apparaît dans la condition d'un test ou comme un diviseur.

En dépit de ses limites, cette méthode simple présente l'avantage considérable d'alerter l'utilisateur sur le manque de robustesse numérique de certaines opérations *dans le cas particulier des données en cours de traitement*. On peut donc la voir comme un débogueur numérique en ligne.

## 14.7 Exemples MATLAB

Considérons à nouveau l'exemple 1.2, qui comparait deux méthodes de résolution de l'équation polynomiale du second ordre

$$ax^2+bx+c=0, \tag{14.83}$$

à savoir les formules du lycée

$$x_1^{\text{hs}} = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad \text{et} \quad x_2^{\text{hs}} = \frac{-b-\sqrt{b^2-4ac}}{2a}. \tag{14.84}$$

et les formules plus robustes

$$q = \frac{-b - \mathrm{sign}(b)\sqrt{b^2 - 4ac}}{2}, \tag{14.85}$$

$$x_1^{\mathrm{mr}} = \frac{c}{q} \quad \text{et} \quad x_2^{\mathrm{mr}} = \frac{q}{a}. \tag{14.86}$$

Les ennuis commencent quand $b$ est très grand par rapport à $ac$ ; prenons donc $a = c = 1$ et $b = 2 \cdot 10^7$. En tapant

```
Digits:=20;
f:=x^2+2*10^7*x+1;
fsolve(f=0);
```

dans Maple, on trouve qu'une solution précise est

$$\begin{aligned} x_1^{\mathrm{as}} &= -5.0000000000000125000 \cdot 10^{-8}, \\ x_2^{\mathrm{as}} &= -1.9999999999999950000 \cdot 10^{7}. \end{aligned} \tag{14.87}$$

Cette solution servira d'étalon pour évaluer à quel point les méthodes présentées dans les sections 14.5.2.2, 14.5.2.3 et 14.6 sont efficaces dans leurs évaluations de la précision avec laquelle $x_1$ et $x_2$ sont calculés par les formules du lycée et les formules plus robustes.

### *14.7.1 Commuter la direction d'arrondi*

La mise en œuvre de la méthode par commutation de la direction d'arrondi présentée en section 14.5.2.2 requiert de contrôler les modes d'arrondi. MATLAB ne permet malheureusement pas encore de le faire directement, mais c'est possible via la *INTLAB toolbox* [195]. Une fois cette boîte à outils installée et démarrée par la commance MATLAB `startintlab`, la commande `setround(-1)` commute sur le mode d'arrondi vers $-\infty$, tandis que `setround(1)` commute sur le mode d'arrondi vers $+\infty$ et que `setround(0)` restaure l'arrondi vers le plus proche. Notons que la fonction `sqrt` de MATLAB n'est pas conforme à la norme IEEE 754 et doit être remplacée par la fonction `sqrt_rnd` d'INTLAB pour le calcul des racines carrés requises pour l'exemple.

Avec l'arrondi vers $-\infty$, les résultats sont

$$\begin{aligned} x_1^{\mathrm{hs}-} &= -5.029141902923584 \cdot 10^{-8}, \\ x_2^{\mathrm{hs}-} &= -1.999999999999995 \cdot 10^{7}, \\ x_1^{\mathrm{mr}-} &= -5.000000000000013 \cdot 10^{-8}, \\ x_2^{\mathrm{mr}-} &= -1.999999999999995 \cdot 10^{7}. \end{aligned} \tag{14.88}$$

Avec l'arrondi vers $+\infty$, ils deviennent

$$\begin{aligned} x_1^{\mathrm{hs}+} &= -4.842877388000488 \cdot 10^{-8}, \\ x_2^{\mathrm{hs}+} &= -1.999999999999995 \cdot 10^{7}, \\ x_1^{\mathrm{mr}+} &= -5.000000000000012 \cdot 10^{-8}, \\ x_2^{\mathrm{mr}+} &= -1.999999999999995 \cdot 10^{7}. \end{aligned} \tag{14.89}$$

En appliquant (14.36), nous obtenons alors

$$\begin{aligned} \widehat{n}_{\mathrm{d}}(x_1^{\mathrm{hs}}) &\approx 1.42, \\ \widehat{n}_{\mathrm{d}}(x_2^{\mathrm{hs}}) &\approx 15.72, \\ \widehat{n}_{\mathrm{d}}(x_1^{\mathrm{mr}}) &\approx 15.57, \\ \widehat{n}_{\mathrm{d}}(x_2^{\mathrm{mr}}) &\approx 15.72. \end{aligned} \tag{14.90}$$

En arrondissant ces estimées vers l'entier non négatif le plus proche, nous pouvons n'écrire que les chiffres jugés significatifs dans les résultats. Ainsi

$$\begin{aligned} x_1^{\mathrm{hs}} &= -5 \cdot 10^{-8}, \\ x_2^{\mathrm{hs}} &= -1.999999999999995 \cdot 10^{7}, \\ x_1^{\mathrm{mr}} &= -5.000000000000013 \cdot 10^{-8}, \\ x_2^{\mathrm{mr}} &= -1.999999999999995 \cdot 10^{7}. \end{aligned} \tag{14.91}$$

*Remarque 14.8.* À partir de sa version 2016a, MATLAB devrait permettre le changement direct de mode d'arrondi, déjà possible en OCTAVE. □

### *14.7.2 Calculer avec des intervalles*

La résolution de cette équation polynomiale avec la *INTLAB toolbox* est particulièrement facile. Il suffit de spécifier que $a$, $b$ et $c$ sont des intervalles (dégénérés), en écrivant

```
a = intval(1);
b = intval(20000000);
c = intval(1);
```

Les réels $a$, $b$ et $c$ sont alors remplacés par les plus petits intervalles représentables en machine qui les contiennent. Tous les calculs impliquant ces intervalles produisent des intervalles contenant à coup sûr les résultats mathématiques exacts et dont les bornes inférieures et supérieures sont représentables en machine. INTLAB peut fournir les résultats en se limitant aux chiffres partagés par les bornes inférieures et supérieures de leurs valeurs intervalles, les autres chiffres étant remplacés par des tirets bas. On obtient alors les résultats suivants

```
intval x1hs = -5._____________e-008
intval x2hs = -1.999999999999995e+007
```

```
intval x1mr = -5.00000000000001_e-008
intval x2mr = -1.999999999999995e+007
```

Ils sont cohérents avec ceux de l'approche par commutation des modes d'arrondi, et obtenus d'une façon garantie. Il ne faut cependant pas en déduire que le calcul garanti par intervalles peut toujours être utilisé à la place d'approches non garanties telles que la commutation de mode d'arrondi ou CESTAC/CADNA. Cet exemple est en fait si simple que les effets du pessimisme inhérent au calcul par intervalles ne sont pas mis en évidence, bien qu'aucun effort n'ait été fait pour les réduire. Pour des calculs plus complexes, la situation serait bien différente et les tailles des intervalles contenant les résultats pourraient vite devenir beaucoup trop grandes (à moins que des mesures spécifiques et non triviales ne soient prises).

### *14.7.3 Utiliser CESTAC/CADNA*

A défaut d'une boîte à outils MATLAB implémentant CESTAC/CADNA, nous utilisons les deux résultats obtenus en section 14.7.1 par commutation des modes d'arrondi pour estimer le nombre de chiffres significatifs grâce à (14.74). Exploitant la remarque 14.7, nous soustrayons 0.8 aux estimées (14.90) du nombre de chiffres significatifs, pour obtenir

$$\begin{aligned} \widehat{n}_{\mathrm{d}}(x_1^{\mathrm{hs}}) &\approx 0.62, \\ \widehat{n}_{\mathrm{d}}(x_2^{\mathrm{hs}}) &\approx 14.92, \\ \widehat{n}_{\mathrm{d}}(x_1^{\mathrm{mr}}) &\approx 14.77, \\ \widehat{n}_{\mathrm{d}}(x_2^{\mathrm{mr}}) &\approx 14.92. \end{aligned} \tag{14.92}$$

En arrondissant ces estimées à l'entier non négatif le plus proche et en ne gardant que les chiffres jugés significatifs, nous obtenons les résultats légèrement modifiés suivants

$$\begin{aligned} x_1^{\mathrm{hs}} &= -5 \cdot 10^{-8}, \\ x_2^{\mathrm{hs}} &= -1.99999999999999 \cdot 10^{7}, \\ x_1^{\mathrm{mr}} &= -5.00000000000001 \cdot 10^{-8} \\ x_2^{\mathrm{mr}} &= -1.99999999999999 \cdot 10^{7}. \end{aligned} \tag{14.93}$$

L'approche CESTAC/CADNA suggère donc de supprimer des chiffres que l'approche par commutation jugeait significatifs. Sur cet exemple spécifique, l'étalon (14.87) révèle que l'approche par commutation, plus optimiste, a raison puisque ces chiffres sont en effet corrects. Les deux approches, comme le calcul par intervalles, mettent clairement en évidence un problème dans le calcul de $x_1$ avec la méthode du lycée.

## 14.8 En résumé

— Passer du calcul analytique au calcul numérique avec des nombres à virgule flottante se traduit par des erreurs inévitables, dont les conséquences doivent être analysées et minimisées.
— Les opérations potentiellement les plus dangereuses sont la soustraction de nombres proches, la division par un ZI, et le branchement conditionnel sur la base de la valeur ou du signe d'un ZI.
— Parmi les méthodes proposées dans la littérature pour évaluer l'effet des erreurs dues aux arrondis, celles qui utilisent l'ordinateur pour étudier les conséquences de ses propres erreurs ont deux avantages : elles sont applicables à de vastes classes d'algorithmes, et elles tiennent compte des spécificités des données à traiter.
— Une simple commutation de la direction d'arrondi peut suffire à révéler une grande incertitude sur des résultats numériques.
— L'analyse par intervalles produit des résultats garantis avec des estimées d'erreurs qui peuvent être très pessimistes à moins que des algorithmes dédiés ne soient utilisés. Ceci limite son applicabilité, mais le fait de pouvoir fournir des bornes sur des erreurs de méthode est un avantage considérable.
— L'analyse des erreurs courantes perd cet avantage et ne fournit que des bornes approchées sur l'effet de la propagation des erreurs dues aux arrondis, mais elle est beaucoup plus simple à mettre en œuvre.
— L'approche par perturbation aléatoire CESTAC/CADNA ne souffre pas du pessimisme de l'analyse par intervalles. Il faut cependant l'utiliser avec précaution, comme une variante de la preuve par neuf, qui ne peut pas garantir que les résultats numériques fournis par l'ordinateur sont correct mais qui peut détecter qu'ils ne le sont pas. Elle peut contribuer à vérifier que ses conditions de validité sont satisfaites.

# Chapitre 15
# Aller plus loin grâce au WEB

Ce chapitre suggère des sites WEB qui donnent accès à des logiciels numériques ainsi qu'à des informations complémentaires sur les concepts et les méthodes présentées dans les autres chapitres. La plupart des ressources décrites peuvent être utilisées gratuitement. La classification retenue est discutable, car la même URL peut pointer vers plusieurs types de ressources.

## 15.1 Moteurs de recherche

Entre autres applications innombrables, les moteurs de recherche d'usage général peuvent être utilisés pour trouver les pages personnelles de chercheurs en analyse numérique. Il n'est pas rare de trouver des transparents de cours et des versions électroniques d'articles ou même de livres librement disponibles via ces pages.

*Google Scholar* (www.scholar.google.com) est un moteur de recherche plus spécialisé qui cible la littérature académique. On peut l'utiliser pour trouver qui a cité un auteur ou un article donné, ce qui permet de voir ce que fut le destin d'une idée intéressante. En se créant un profil public, on peut même se voir suggérer des articles.

*Publish or Perish* (www.harzing.com) récupère et analyse les citations académiques en s'appuyant sur Google Scholar. On peut l'utiliser pour évaluer l'impact d'une méthode, d'un auteur ou d'un journal sur la communauté scientifique.

*YouTube* (www.youtube.com) donne accès à de nombreuses vidéos pédagogiques sur des thèmes couverts par ce livre.

## 15.2 Encyclopédies

Sur à peu près n'importe quel concept ou méthode numérique mentionné dans ce livre, *Wikipedia* (www.wikipedia.org) fournit des informations complémentaires.

*Scholarpedia* (www.scholarpedia.org) est un ensemble d'encyclopédies en accès libre avec évaluation des contributions par les pairs. Cet ensemble inclut une *Encyclopedia of Applied Mathematics* avec des articles sur les équations différentielles, l'analyse numérique et l'optimisation.

L'*Encyclopedia of Mathematics* (www.encyclopediaofmath.org) est une autre source précieuse d'informations, avec un bureau éditorial géré par l'*European Mathematical Society* qui a toute autorité sur les modifications et les suppressions.

## 15.3 Entrepôts

Le classement des entrepôts à repositories.webometrics.info/en/world contient des pointeurs vers beaucoup plus d'entrepôts que ceux listés ci-dessous, dont certains sont aussi intéressants dans le contexte du calcul numérique.

*NETLIB* (www.netlib.org) est une collection d'articles, de bases de données et de logiciels mathématiques. Cet entrepôt donne par exemple accès à LAPACK, une collection de routines de classe professionnelle pour calculer

— des solutions de systèmes d'équations linéaires,
— des valeurs et vecteurs propres,
— des valeurs singulières,
— des conditionnements,
— des factorisations de matrices (LU, Cholesky, QR, SVD, etc.),
— des solutions au sens des moindres carrés de systèmes d'équations linéaires.

*GAMS* (gams.nist.gov), l'acronyme de *Guide to Available Mathematical Software*, est un entrepôt virtuel de logiciels mathématiques et statistiques avec un intéressant index croisé offert par le *National Institute of Standards and Technology of the US Department of Commerce*.

*ACTS* (acts.nersc.gov), l'acronyme de *Advanced CompuTational Software tools*, regroupe des outils développés par le *US Department of Energy*, parfois en collaboration avec d'autres agences de financement comme la DARPA ou la NSF. Il donne accès à

— AZTEC, une bibliothèque d'algorithmes pour la résolution itérative de grands systèmes creux comportant des solveurs itératifs, des préconditionneurs et des routines de produit matrice-vecteur ;
— HYPRE, une bibliothèque pour résoudre des grands systèmes d'équations linéaires creux sur des calculateurs massivement parallèles ;
— OPT++, un paquetage d'optimisation non linéaire orienté objet qui contient diverses méthodes de Newton, une méthode de gradients conjugués et une méthode à points intérieurs pour les problèmes non linéaires ;
— PETSc, qui fournit des outils pour la résolution numérique d'EDP en parallèle (aussi bien qu'en série) ; PETSc contient des solveurs pour les grands systèmes creux d'équations linéaires et non linéaires ;

— ScaLAPACK, une librairie de routines d'algèbre linéaire pour les calculateurs à mémoire distribuée et pour les réseaux de stations de travail ; ScaLAPACK est une suite du projet LAPACK ;
— SLEPc, un paquetage pour le calcul de valeurs propres et de vecteurs propres de matrices creuses de grande taille sur des calculateurs parallèles, ainsi que le calcul de décompositions en valeurs singulières ;
— SUNDIALS [106], une famille de solveurs très liés : CVODE, pour les systèmes d'équations différentielles ordinaires, CVODES, une variante de CVODE pour l'analyse de sensibilité, KINSOL, pour les systèmes d'équations algébriques non linéaires, et IDA, pour les systèmes d'équations algébro-différentiels ; ces solveurs peuvent traiter des systèmes d'extrêmement grande taille, dans des environnements série ou parallèle ;
— SuperLU, une bibliothèque d'usage général pour la résolution directe de grands systèmes d'équations linéaires creux et non symétriques via des factorisations LU ;
— TAO, un logiciel pour les grands problèmes d'optimisation (moindres carrés non linéaires, optimisation sans contrainte, optimisation avec des bornes, optimisation non linéaire générale, avec un fort accent sur la réutilisation d'outils extérieurs quand c'est approprié ; TAO peut être utilisé dans des environnements série ou parallèle.

Les pages dédiées à ces produits contiennent des pointeurs vers nombre d'autres logiciels intéressants.

*CiteSeerX* (citeseerx.ist.psu.edu) se concentre sur la littérature en informatique. On peut l'utiliser pour trouver des articles qui en citent d'autres jugés intéressants, et ce site fournit souvent un accès gratuit à des versions électroniques de ces articles.

La *Collection of Computer Science Bibliographies* abrite plus de trois millions de références, principalement à des articles de journaux et de conférences et à des rapports techniques. Environ un million de ces références contient une URL pointant sur une version en ligne du document (liinwww.ira.uka.de/bibliography).

Le *Arxiv Computing Research Repository* (arxiv.org) permet à des chercheurs de rechercher et de télécharger gratuitement des articles.

*HAL* (hal.archives-ouvertes.fr) est une autre archive multidisciplinaire à accès libre pour le dépôt et la distribution d'articles de recherche scientifique et de mémoires de thèses de doctorat.

*Interval Computation* (www.cs.utep.edu/interval-comp) est une source riche en informations sur les calculs garantis à base d'analyse par intervalles.

## 15.4 Logiciels

### *15.4.1 Langages interprétés de haut niveau*

Les langages interprétés de haut niveau sont utilisés principalement pour le prototypage et l'enseignement, ainsi que pour la conception d'interfaces commodes avec du code compilé d'exécution plus rapide.

*MATLAB* (www.mathworks.com/products/matlab) est la référence principale dans ce contexte. Une documentation intéressante sur le calcul numérique avec MATLAB due à Cleve Moler peut être téléchargée de www.mathworks.com/moler.

Malgré sa popularité méritée, MATLAB a plusieurs inconvénients :

— il est cher (particulièrement pour les utilisateurs industriels qui ne bénéficient pas des prix spéciaux éducation),
— le code source MATLAB développé ne peut pas être utilisé par d'autres (à moins qu'ils n'aient accès eux aussi à MATLAB),
— des parties du code source ne sont pas accessibles.

Pour ces raisons, ou si l'on trouve inconfortable d'avoir un fournisseur unique, les deux options suivantes méritent considération :

*GNU Octave* (www.gnu.org/software/octave) a été construit en visant la compatibilité avec MATLAB ; il donne un accès gratuit à tout son code source et est librement redistribuable sous les termes de la *General Public License* (GPL) de GNU ; (GNU est l'acronyme récursif de *GNU is Not Unix*, une blague privée pour spécialiste des systèmes d'exploitation ;)

*Scilab* (www.scilab.org), initialement développé par Inria, donne lui aussi accès à tout son code source. Il est distribué sous licence CeCILL (compatible GPL).

Si certaines des boîtes à outils MATLAB sont des produits commerciaux, d'autres sont mises à disposition gratuitement, au moins pour une utilisation non commerciale. Un bon exemple était INTLAB (www.ti3.tu-harburg.de/rump/intlab/), une boîte à outils pour le calcul numérique garanti à base d'analyse par intervalles offrant, parmi beaucoup d'autres choses, la différentiation automatique et le contrôle du mode d'arrondi. INTLAB est maintenant disponible pour une somme symbolique. Chebfun, un logiciel open-source utilisable, parmi beaucoup d'autres choses, pour de l'interpolation à haute précision avec des polynômes d'ordre élevé grâce à l'utilisation de la formule de Lagrange barycentrique et des points de Tchebychev, peut être obtenu à www2.maths.ox.ac.uk/chebfun/. DACE (l'acronyme de *Design and Analysis of Computer Experiments*, http ://www2.imm.dtu.dk/h̃bn/dace/) et STK (l'acronyme de *Small Toolbox for Kriging*, sourceforge.net/projects/kriging/) sont deux boîtes à outils gratuites mettant en œuvre le krigeage. SuperEGO, un produit en MATLAB pour l'optimisation sous contrainte à base de krigeage, peut être obtenu (seulement pour un usage académique) sur demande à P.Y. Papalambros à pyp@umich.edu. D'autres ressources gratuites peuvent être obtenues à www.mathworks.com/matlabcentral/fileexchange/.

Un autre langage qui mérite d'être mentionné est *R* (www.r-project.org), principalement utilisé par des statisticiens mais qui ne se limite pas aux statistiques.

R est un autre projet GNU. Des pointeurs à des paquetages R pour le krigeage et l'optimisation par *efficient global optimization* (EGO) sont à ls11-www.cs.uni-dortmund.de/rudolph/kriging/dicerpackage.

De nombreuses ressources pour le calcul scientifique en *Python* (y compris SciPy) sont listées à (www.scipy.org/Topical_Software). L'implémentation de Python est sous une licence open source qui le rend librement utilisable et distribuable, y compris pour un usage commercial.

### 15.4.2 Bibliothèques pour des langages compilés

GSL est l'acronyme de *GNU Scientific Library* (www.gnu.org/software/gsl/), une bibliothèque de fonctions pour les programmeurs en C et en C++, avec un statut de logiciel libre sous licence GNU GPL. GSL fournit plus de mille fonctions avec une documentation détaillée [65], qui peut être téléchargée gratuitement dans une version mise à jour. Une longue série de tests est aussi fournie. La plupart des thèmes de ce livre sont couverts.

*Numerical Recipes* (www.nr.com) distribue le code présenté dans les livres éponymes pour un coût modeste, mais avec une licence qui n'autorise pas sa redistribution.

Des produits commerciaux classiques sont *IMSL* et *NAG*.

### 15.4.3 Autres ressources pour le calcul scientifique

Le *serveur NEOS* (www.neos-server.org/neos/) peut être utilisé pour résoudre des problèmes d'optimisation éventuellement de grande taille sans avoir à acheter et gérer le logiciel requis. Les utilisateurs peuvent ainsi se concentrer sur la définition de leurs problèmes d'optimisation. NEOS est l'acronyme de *network-enabled optimization software*. Des informations sur l'optimisation sont aussi fournies à neos-guide.org.

*BARON* (archimedes.cheme.cmu.edu/baron/baron.html) est un système pour résoudre des problèmes d'optimisation non convexes. Bien qu'il en existe des versions commerciales, on peut aussi y accéder gratuitement via le serveur NEOS.

*FreeFEM++* (www.freefem.org) est un solveur à éléments finis pour les EDP. Il a déjà été utilisé sur des problèmes à plus de $10^9$ inconnues.

*FADBAD++* (www.fadbad.com/fadbad.html) met en œuvre la différentiation automatique en mode progressif et rétrograde en utilisant la surcharge d'opérateurs en C++.

*VNODE* (www.cas.mcmaster.ca/ nedialk/Software/VNODE/VNODE.shtml) est un paquetage C++ pour le calcul de bornes rigoureuses pour les solutions de problèmes aux valeurs initiales pour des EDO.

*COMSOL Multiphysics* (www.comsol.com) est un environnement éléments finis commercial pour la simulation des modèles à base d'EDP avec des conditions aux limites compliquées. Il peut par exemple traiter des problèmes impliquant de la chimie *et* des transferts de chaleur *et* de la mécanique des fluides.

## 15.5 OpenCourseWare

L'*OpenCourseWare*, ou OCW, c'est du matériel d'enseignement créé par des universités et partagé gratuitement sur Internet. Ce matériel peut inclure des vidéos, des notes de cours, des transparents, des examens et leurs solutions, etc. Parmi les institutions qui offrent des cours en mathématiques appliquées et en informatique, on peut citer

— le MIT (ocw.mit.edu),
— Harvard (www.extension.harvard.edu/open-learning-initiative),
— Stanford (see.stanford.edu), avec, par exemple, deux séries de conférences par Stephen Boyd sur les systèmes linéaires et l'optimisation convexe,
— Berkeley (webcast.berkeley.edu),
— the University of South Florida (mathforcollege.com/nm/)

Le *OCW finder* (www.opencontent.org/ocwfinder/) permet une recherche de cours sur un ensemble d'universités. Citons enfin le *Demonstrations Project* de Wolfram (demonstrations.wolfram.com), avec des thèmes sur le calcul et l'analyse numérique.

Les MOOC (l'acronyme de *Massive Open Online Courses*) peuvent être mis en temps réel à la disposition de milliers d'étudiants par Internet, avec des niveaux d'interactivité variables. Parmi les fournisseurs de MOOC on peut citer edX, Coursera et Udacity.

# Littérature


1. Acton, F. : Numerical Methods That (usually) Work, revised edn. Mathematical Association of America, Washington, DC (1990)
2. Al-Mohy, A., Higham, N. : A new scaling and squaring algorithm for the matrix exponential. SIAM J. Matrix Analysis and Applications **31**(3), 970–989 (2009)
3. Alexander, R. : Diagonally implicit Runge-Kutta methods for stiff O.D.E.'s. SIAM J. Numer. Anal. **14**(6), 1006–1021 (1977)
4. Applegate, D., Bixby, R., Chvátal, V., Cook, W. : The Traveling Salesman Problem : A Computational Study. Princeton University Press, Princeton, NJ (2006)
5. Applegate, D., Bixby, R., Chvátal, V., Cook, W., Espinoza, D., Goycoolea, M., Helsgaun, K. : Certification of an optimal TSP tour through 85,900 cities. Operations Research Letters **37**, 11–15 (2009)
6. Ascher, U., Greif, C. : A First Course in Numerical Methods. SIAM, Philadelphia, PA (2011)
7. Ashino, R., Nagase, M., Vaillancourt, R. : Behind and beyond the Matlab ODE suite. Computers and Mathematics with Applications **40**, 491–512 (2000)
8. Beichl, I., Sullivan, F. : The Metropolis algorithm. Computing in Science and Engineering **2**(1), 65–69 (2000)
9. Bekey, G., Masri, S. : Random search techniques for optimization of nonlinear systems with many parameters. Math. and Comput. in Simulation **25**, 210–213 (1983)
10. Ben-Tal, A., Ghaoui, L.E., Nemirovski, A. : Robust Optimization. Princeton University Press, Princeton, NJ (2009)
11. Benzi, M. : Preconditioning techniques for large linear systems : A survey. Journal of Computational Physics **182**, 418–477 (2002)
12. Berrut, J.P., Trefethen, L. : Barycentric Lagrange interpolation. SIAM Review **46**(3), 501–517 (2004)
13. Bertsekas, D. : Constrained Optimization and Lagrange Multiplier Methods. Athena Scientific, Belmont, MA (1996)
14. Bertsekas, D. : Nonlinear Programming. Athena Scientific, Belmont, MA (1999)
15. Bertsimas, D., Brown, D., Caramanis, C. : Theory and applications of robust optimization. SIAM Review **53**(3), 464–501 (2011)
16. Bertz, M., Makino, K. : Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. Reliable Computing **4**, 361–369 (1998)
17. Bhatti, M., Bracken, P. : Solution of differential equations in a Bernstein polynomial basis. J. of Computational and Applied Mathematics **205**, 272–280 (2007)
18. Björck, A. : Numerical Methods for Least Squares Problems. SIAM, Philadelphia, PA (1996)
19. Bogacki, P., Shampine, L. : A 3(2) pair of Runge-Kutta formulas. Applied Mathematics Letters **2**(4), 321–325 (1989)
20. Boggs, P., Tolle, J. : Sequential quadratic programming. Acta Numerica **4**, 1–51 (1995)
21. Boggs, P., Tolle, J. : Sequential quadratic programming for large-scale nonlinear optimization. J. of Computational and Applied Mathematics **124**, 123–137 (2000)
22. Bonnans, J., Gilbert, J.C., Lemaréchal, C., Sagastizabal, C. : Numerical Optimization – Theoretical and Practical Aspects. Springer, Berlin, DE (2006)
23. de Boor, C. : Package for calculating with B-splines. SIAM J. Numer. Anal. **14**(3), 441–472 (1977)
24. de Boor, C. : A Practical Guide to Splines, revised edn. Springer, New York, NY (2001)

25. de Boor, C., Swartz, B. : Comments on the comparison of global methods for linear two-point boudary value problems. Mathematics of Computation **31**(140), 916–921 (1977)
26. Borrie, J. : Stochastic Systems for Engineers. Prentice-Hall, Hemel Hempstead, UK (1992)
27. Boulier, F., Lefranc, M., Lemaire, F., Morant, P.E. : Model reduction of chemical reaction systems using elimination. Mathematics in Computer Science **5**, 289–301 (2011)
28. Boyd, S., Vandenberghe, L. : Convex Optimization. Cambridge University Press, Cambridge, UK (2004)
29. Brent, R. : Algorithms for Minimization Without Derivatives. Prentice-Hall, Englewood Cliffs, NJ (1973)
30. Bronson, R. : Operations Research. Schaum's Outline Series. McGraw-Hill, New York, NY (1982)
31. Broyden, C. : A class of methods for solving nonlinear simultaneous equations. Mathematics of Computation **19**(92), 577–593 (1965)
32. Broyden, C. : Quasi-Newton methods and their application to function minimization. Mathematics of Computation **21**(99), 368–381 (1967)
33. Bryan, K., Leise, T. : The $25,000,000,000 eigenvector : The linear algebra behind Google. SIAM Review **48**(3), 569–581 (2006)
34. Butcher, J. : Implicit Runge-Kutta processes. Mathematics of Computation **18**(85), 50–64 (1964)
35. Butcher, J., Wanner, G. : Runge-Kutta methods : some historical notes. Applied Numerical Mathematics **22**, 113–151 (1996)
36. Byrd, R., Hribar, M., Nocedal, J. : An interior point algorithm for large-scale nonlinear programming. SIAM J. Optimization **9**(4), 877–900 (1999)
37. Chandrupatla, T., Belegundu, A. : Introduction to Finite Elements in Engineering, third edn. Prentice-Hall, Upper Saddle River, NJ (2002)
38. Chesneaux, J.M. : Etude théorique et implémentation en ADA de la méthode CESTAC. Ph.D. thesis, Université Pierre et Marie Curie (1988)
39. Chesneaux, J.M. : Study of the computing accuracy by using probabilistic approach. In : C. Ullrich (ed.) Contribution to Computer Arithmetic and Self-Validating Methods, pp. 19–30. J.C. Baltzer AG, Amsterdam, NL (1990)
40. Chesneaux, J.M. : L'arithmétique stochastique et le logiciel CADNA. Habilitation à diriger des recherches, Université Pierre et Marie Curie (1995)
41. Chilès, J.P., Delfiner, P. : Geostatistics. Wiley, New York, NY (1999)
42. Ciarlet, P. : Introduction to Numerical Linear Algebra and Optimization. Cambridge University Press, Cambridge, UK (1989)
43. Collette, Y., Siarry, P. : Multiobjective Optimization. Springer, Berlin, DE (2003)
44. Conn, A., Gould, N., Toint, P. : A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds. SIAM J. Numer. Anal. **28**(2), 545–572 (1991)
45. Conn, A., Gould, N., Toint, P. : A globally convergent augmented Lagrangian barrier algorithm for optimization with general constraints and simple bounds. Tech. Rep. 92/07 (2nd revision), IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA (1995)
46. Cressie, N. : Statistics for Spatial Data. Wiley, New York, NY (1993)
47. Dahlquist, G. : A special stability problem for linear multistep methods. BIT Numerical Mathematics **3**(1), 27–43 (1963)
48. Demmel, J. : The probability that a numerical analysis problem is difficult. Mathematics of Computation **50**(182), 449–480 (1988)

49. Demmel, J. : Applied Numerical Linear Algebra. SIAM, Philadelphia, PA (1997)
50. Demmel, J., Kahan, W. : Accurate singular values of bidiagonal matrices. SIAM J. Scientific and Statistical Computing **11**(5), 873–912 (1990)
51. Dennis, JR, J., Moré, J. : Quasi-Newton methods, motivations and theory. SIAM Review **19**(1), 46–89 (1977)
52. Didrit, O., Petitot, M., Walter, E. : Guaranteed solution of direct kinematic problems for general configurations of parallel manipulators. IEEE Trans. on Robotics and Automation **14**(2), 259–266 (1998)
53. Diez, P. : A note on the convergence of the secant method for simple and multiple roots. Applied Mathematics Letters **16**, 1211–1215 (2003)
54. Dixon, L. : Quasi Newton techniques generate identical points II : The proofs of four new theorems. Mathematical Programming **3**, 345–358 (1972)
55. Dongarra, J., Sullivan, F. : Guest editors' introduction to the top 10 algorithms. Computing in Science and Engineering **2**(1), 22–23 (2000)
56. Dorato, P., Abdallah, C., Cerone, V. : Linear-Quadratic Control, An Introduction. Prentice-Hall, Englewood Cliffs, NJ (1995)
57. Dorigo, M., Stützle, T. : Ant Colony Optimization. MIT Press, Cambridge, MA (2004)
58. Dormand, J., Prince, P. : A family of embedded Runge-Kutta formulae. J. of Computational and Applied Mathematics **6**(1), 19–26 (1980)
59. Dormand, J., Prince, P. : A reconsideration of some embedded Runge-Kutta formulae. J. of Computational and Applied Mathematics **15**, 203–211 (1986)
60. Droste, S., Jansen, T., Wegener, I. : Perhaps not a free lunch but at least a free appetizer. In : Proc. 1st Genetic and Evolutionary Computation Conf., pp. 833–839. Orlando, FL (1999)
61. Duchêne, P., Rouchon, P. : Kinetic sheme reduction, attractive invariant manifold and slow/fast dynamical systems. Chem. Eng. Science **53**, 4661–4672 (1996)
62. Farouki, R. : The Bernstein polynomial basis : A centennial retrospective. Computer Aided Geometric Design **29**, 379–419 (2012)
63. Floudas, C., Pardalos, P. (eds.) : Encyclopedia of Optimization, second edn. Springer, New York, NY (2009)
64. Fortin, A. : Numerical Analysis for Engineers. Ecole Polytechnique de Montréal, Montréal, CAN (2009)
65. Galassi et al., M. : GNU Scientific Library Reference Manual, third edn. NetworkTheory Ltd, Bristol, UK (2009)
66. Gander, M., Wanner, G. : From Euler, Ritz, and Galerkin to modern computing. SIAM Review **54**(4), 627–666 (2012)
67. Gander, W., Gautschi, W. : Adaptive quadrature - revisited. BIT **40**(1), 84–101 (2000)
68. Gear, C. : Numerical Initial Value Problems in Ordinary Differential Equations. Prentice-Hall, Englewood Cliffs, NJ (1971)
69. Gertz, E. : A quasi-Newton trust-region method. Mathematical Programming **100**(3), 447–470 (2004)
70. Gilbert, J., Moler, C., Schreiber, R. : Sparse matrices in MATLAB : Design and implementation. SIAM J. Matrix Analysis and Applications **13**, 333–356 (1992)
71. Gilbert, J., Vey, G.L., Masse, J. : La différentiation automatique de fonctions représentées par des programmes. Tech. Rep. 1557, INRIA (1991)
72. Gill, P., Murray, W., Saunders, M., Tomlin, J., Wright, M. : On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method. Math. Prog. **36**, 183–209 (1986)

73. Gill, P., Murray, W., Wright, M. : Practical Optimization. Elsevier, London, UK (1986)
74. Gladwell, I., Shampine, L., Brankin, R. : Locating special events when solving ODEs. Applied Mathematics Letters **1**(2), 153–156 (1988)
75. Goldberg, D. : Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, MA (1989)
76. Goldberg, D. : What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys **23**(1), 5–48 (1991)
77. Golub, G., Kahan, W. : Calculating the singular values and pseudo-inverse of a matrix. J. Soc. Indust. Appl. Math : Series B, Numerical Analysis **2**(2), 205–224 (1965)
78. Golub, G., O'Leary, D. : Some history of the conjugate gradient and Lanczos algorithms : 1948-1976. SIAM Review **31**(1), 50–102 (1989)
79. Golub, G., Reinsch, C. : Singular value decomposition and least squares solution. Numer. Math. **14**, 403–420 (1970)
80. Golub, G., Van Loan, C. : Matrix Computations, third edn. The Johns Hopkins University Press, Baltimore, MD (1996)
81. Golub, G., Welsch, J. : Calculation of Gauss quadrature rules. Mathematics of Computation **23**(106), 221–230 (1969)
82. Gonin, R., Money, A. : Nonlinear $L_p$-Norm Estimation. Marcel Dekker, New York, NY (1989)
83. Grabmeier, J., Kaltofen, E., Weispfenning, V. (eds.) : Computer Algebra Handbook : Foundations, Applications, Systems. Springer, Berlin, DE (2003)
84. Griewank, A., Corliss, G. (eds.) : Automatic Differentiation of Algorithms : Theory, Implementation and Applications. SIAM, Philadelphia, PA (1991)
85. Griewank, A., Walther, A. : Principles and Techniques of Algorithmic Differentiation, second edn. SIAM, Philadelphia, PA (2008)
86. Grote, M., Huckle, T. : Parallel preconditioning with sparse approximate inverses. SIAM J. Sci. Comput. **18**(3), 838–853 (1997)
87. Gupta, G., Sacks-Davis, R., Tischer, P. : A review of recent developments in solving ODEs. ACM Computing Surveys **17**(1), 5–47 (1985)
88. Gustafsson, B. : Fundamentals of Scientific Computing. Springer, Berlin, DE (2011)
89. Gutknecht, M. : A brief introduction to Krylov space methods for solving linear systems. In : Y. Kaneda, H. Kawamura, M. Sasai (eds.) Proc. Int. Symp. on Frontiers of Computational Science 2005, pp. 53–62. Springer, Berlin, DE (2007)
90. Hager, W. : Updating the inverse of a matrix. SIAM Review **31**(2), 221–239 (1989)
91. Hager, W., Zhang, H. : A survey of nonlinear conjugate gradient methods. Pacific J. of Optimization **2**(1), 35–58 (2006)
92. Hairer, E., Wanner, G. : On the instability of the BDF formulas. SIAM J. Numer. Anal. **20**(6), 1206–1209 (1983)
93. Hammer, R., Hocks, M., Kulisch, U., Ratz, D. : C++ Toolbox for Verified Computing. Springer, Berlin, DE (1995)
94. Hamming, R. : Numerical Methods for Scientists and Engineers. Dover, New York, NY (1986)
95. Han, S.P., Mangasarian, O. : Exact penalty functions in nonlinear programming. Mathematical Programming **17**, 251–269 (1979)
96. Hansen, E. : Global Optimization Using Interval Analysis. Marcel Dekker, New York, NY (1992)
97. Hestenes, M., Stiefel, E. : Methods of conjugate gradients for solving linear systems. J. of Research of the National Bureau of Standards **49**(6), 409–436 (1952)

98. Higham, D. : An algorithmic introduction to numerical simulation of stochastic differential equations. SIAM Review **43**(3), 525–546 (2001)
99. Higham, N. : Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation (algorithm 674). ACM Trans. on Mathematical Software **14**(4), 381–396 (1988)
100. Higham, N. : Accuracy and Stability of Numerical Algorithms, second edn. SIAM, Philadelphia, PA (2002)
101. Higham, N. : The numerical stability of barycentric Lagrange interpolation. IMA J. Numer. Anal. **24**(4), 547–556 (2004)
102. Higham, N. : Cholesky factorization. Wiley Interdiscipinary Reviews : Computational Statistics **1**(2), 251–254 (2009)
103. Higham, N. : The scaling and squaring method for the matrix exponential revisited. SIAM Review **51**(4), 747–764 (2009)
104. Higham, N. : Gaussian elimination. Wiley Interdiscipinary Reviews : Computational Statistics **3**(3), 230–238 (2011)
105. Higham, N., Tisseur, F. : A block algorihm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. SIAM J. Matrix Analysis and Applications **21**, 1185–1201 (2000)
106. Hindmarsh, A., Brown, P., Grant, K., Lee, S., Serban, R., Shumaker, D., Woodward, C. : SUNDIALS : Suite of nonlinear and differential/algebraic equation solvers. ACM Trans. on Mathematical Software **31**(3), 363–396 (2005)
107. Hiriart-Urruty, J.B., Lemaréchal, C. : Convex Analysis and Minimization Algorithms - Advanced Theory and Bundle Methods. Springer-Verlag, Berlin, DE (1993)
108. Hiriart-Urruty, J.B., Lemaréchal, C. : Convex Analysis and Minimization Algorithms - Fundamentals. Springer-Verlag, Berlin, DE (1993)
109. Ho, Y.C., Pepyne, D. : Simple explanation of the no free lunch theorem of optimization. In : Proc. 40th IEEE Conf. on Decision and Control, pp. 4409–4414. Orlando, FL (2001)
110. Hoffmann, K., Chiang, S. : Computational Fluid Dynamics, vol. 1, fourth edn. Engineering Education System, Wichita, KA (2000)
111. Horst, R., Tuy, H. : Global Optimization. Springer, Berlin, DE (1990)
112. IEEE : IEEE standard for floating-point arithmetic. Tech. Rep. IEEE Standard 754-2008, IEEE Computer Society (2008)
113. Ipsen, I. : Computing an eigenvector with inverse iteration. SIAM Review **39**, 254–291 (1997)
114. Jacquez, J. : Compartmental Analysis in Biology and Medicine. BioMedware, Ann Arbor MI (1996)
115. Jaulin, L., Kieffer, M., Didrit, O., Walter, E. : Applied Interval Analysis. Springer, London, UK (2001)
116. Jazwinski, A. : Stochastic Processes and Filtering Theory. Academic Press, New York, NY (1970)
117. Jones, D. : A taxonomy of global optimization methods based on response surfaces. Journal of Global Optimization **21**, 345–383 (2001)
118. Jones, D., Schonlau, M., Welch, W. : Efficient global optimization of expensive black-box functions. Journal of Global Optimization **13**(4), 455–492 (1998)
119. Kahan, W. : How futile are mindless assessments of roundoff in floating-point computation ? (2006). Work in progress
120. Karmarkar, N. : A new polynomial-time algorithm for linear programming. Combinatorica **4**(4), 373–395 (1984)
121. Kearfott, R. : Globsol user guide. Optimization Methods Software **24**(4-5), 687–708 (2009)

122. Kelley, C. : Iterative Methods for Optimization. SIAM, Philadelphia, PA (1999)
123. Kelley, C. : Solving Nonlinear Equations with Newton's Method. SIAM, Philadelphia, PA (2003)
124. Kennedy, J., Eberhart, R., Shi, Y. : Swarm Intelligence. Morgan Kaufmann, San Francisco, CA (2001)
125. Kershaw, D. : A note on the convergence of interpolatory cubic splines. SIAM J. Numer. Anal. **8**(1), 67–74 (1971)
126. Khachiyan, L. : A polynomial algorithm in linear programming. Soviet Mathematics Doklady **20**, 191–194 (1979)
127. Kierzenka, J., Shampine, L. : A BVP solver based on residual control and the MATLAB PSE. ACM Trans. on Mathematical Software **27**(3), 299–316 (2001)
128. Kiountouzis, E. : Linear programming techniques in regression analysis. J. Royal Statistical Society. Series C (Applied Statistics) **22**(1), 69–73 (1973)
129. Klopfenstein, R. : Numerical differentiation formulas for stiff systems of ordinary differential equations. RCA Review **32**, 447–462 (1971)
130. Knuth, D. : The Art of Computer Programming : 2 Seminumerical Algorithms, third edn. Addison-Wesley, Reading, MA (1997)
131. Krige, D. : A statistical approach to some basic mine valuation problems on the Witwatersrand. J. of the Chemical, Metallurgical and Mining Society **52**, 119–139 (1951)
132. Kulisch, U. : Very fast and exact accumulation of products. Computing **91**, 397–405 (2011)
133. La Porte, M., Vignes, J. : Algorithmes numériques, analyse et mise en œuvre, 1 : Arithmétique des ordinateurs. Systèmes linéaires. Technip, Paris, FR (1974)
134. van Laarhoven, P., Aarts, E. : Simulated Annealing : Theory and Applications. Kluwer, Dordrecht (1987)
135. Lagarias, J., Poonen, B., Wright, M. : Convergence of the restricted Nelder-Mead algorithm in two dimensions. SIAM J. Optimization **22**(2), 501–532 (2012)
136. Lagarias, J., Reeds, J., Wright, M., Wright, P. : Convergence properties of the Nelder-Mead simplex method in low dimensions. SIAM J. Optimization **9**(1), 112–147 (1998)
137. Langville, A., Meyer, C. : Google's PageRank and Beyond. Princeton University Press, Princeton, NJ (2006)
138. Lapidus, L., Pinder, G. : Numerical Solution of Partial Differential Equations in Science and Engineering. Wiley, New York, NY (1999)
139. Lawson, C., Hanson, R. : Solving Least Squares Problems. Classics in Applied Mathematics. SIAM, Philadelphia, PA (1995)
140. Le Roux, N., Schmidt, M., Bach, F. : A stochastic gradient method with an exponential convegence rate for strongly-convex optimization with finite training sets. In : Neural Information Processing Systems (NIPS 2012). Lake Tahoe, NE (2012)
141. Levenberg, K. : A method for the solution of certain non-linear problems in least squares. Quart. Appl. Math. **2**, 164–168 (1944)
142. LeVeque, R. : Finite Difference methods for Ordinary and Partial Differential Equations. SIAM, Philadelphia, PA (2007)
143. Lewis, R., Torczon, V. : A globally convergent augmented Lagrangian pattern algorithm for optimization with general constraints and simple bounds. Tech. Rep. 98-31, NASA - ICASE, NASA Langley Research Center, Hampton, VA (1998)
144. Linfield, G., Penny, J. : Numerical Methods Using MATLAB, 3rd edn. Academic Press – Elsevier, Amsterdam, NL (2012)
145. Lotkin, M. : The treatment of boundary problems by matrix methods. The American Mathematical Monthly **60**(1), 11–19 (1953)

146. Lowan, A., Davids, N., Levenson, A. : Table of the zeros of the Legendre polynomials of order 1-16 and the weight coefficients for Gauss' mechanical quadrature formula. Bulletin of the American Mathematical Society **48**(10), 739–743 (1942)

147. Lowan, A., Davids, N., Levenson, A. : Errata to "Table of the zeros of the Legendre polynomials of order 1-16 and the weight coefficients for Gauss' mechanical quadrature formula". Bulletin of the American Mathematical Society **49**(12), 939–939 (1943)

148. Makino, K., Bertz, M. : Suppression of the wrapping effect by Taylor model-based verified integrators : Long-term stabilization by preconditioning. Int. J. of Differential Equations and Applications **10**(4), 353–384 (2005)

149. Makino, K., Bertz, M. : Suppression of the wrapping effect by Taylor model-based verified integrators : The single step. Int. J. of Pure and Applied Mathematics **36**(2), 175–196 (2007)

150. Marquardt, D. : An algorithm for least-squares estimation of nonlinear parameters. J. Soc. Indust. Appl. Math **11**(2), 431–441 (1963)

151. Marzat, J., Walter, E., Piet-Lahanier, H. : Worst-case global optimization of black-box functions through Kriging and relaxation. Journal of Global Optimization **55**(4), 707–727 (2013)

152. Mathews, J., Fink, K. : Numerical Methods Using MATLAB, fourth edn. Prentice-Hall, Upper Saddle River, NJ (2004)

153. Matousek, J., Gärtner, B. : Understanding and Using Linear Programming. Springer, Berlin, DE (2007)

154. Mattheij, R., Rienstra, S., ten Thije Boonkkamp, J. : Partial Differential Equations – Modeling, Analysis, Computation. SIAM, Philadelphia, PA (2005)

155. Minoux, M. : Mathematical Programming - Theory and Algorithms. Wiley, New York, NY (1986)

156. Mitra, D., Romeo, F., Sangiovanni-Vincentelli, A. : Convergence and finite-time behavior of simulated annealing. Adv. Appl. Prob. **18**, 747–771 (1986)

157. Mockus, J. : Bayesian Approach to Global Optimization. Kluwer, Dordrecht, NL (1989)

158. Moler, C. : Numerical Computing with MATLAB, revised reprinted edn. SIAM, Philadelphia, PA (2008)

159. Moler, C., Van Loan, C. : Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. SIAM Review **45**(1), 3–49 (2003)

160. Moore, R. : Automatic error analysis in digital computation. Technical Report LMSD-48421, Lockheed Missiles and Space Co, Palo Alto, CA (1959)

161. Moore, R. : Interval Analysis. Prentice-Hall (1966)

162. Moore, R. : Mathematical Elements of Scientific Computing. Holt, Rinehart and Winston, New York, NY (1975)

163. Moore, R. : Methods and Applications of Interval Analysis. SIAM, Philadelphia, PA (1979)

164. Morokoff, W., Caflisch, R. : Quasi-Monte Carlo integration. Journal of Computational Physics **122**, 218–230 (1995)

165. Muller, J.M. : Elementary Functions, Algorithms and Implementation, second edn. Birkhäuser, Boston, MA (2006)

166. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S. : Handbook of Floating-Point Arithmetic. Birkhäuser, Boston, MA (2010)

167. Nedialkov, N. : VNODE-LP, a validated solver for initial value problems in ordinary differential equations. Tech. Rep. CAS-06-06-NN, Department of Computing and Software, McMaster University, Hamilton, Ontario, CAN (2006)

168. Nedialkov, N., Jackson, K., Corliss, G. : Validated solutions of initial value problems for ordinary differential equations. Applied Mathematics and Computation **105**(1), 21–68 (1999)

169. Nesterov, Y. : Introductory Lectures on Convex Optimization : A Basic Course. Kluwer, Boston, MA (2004)
170. Nesterov, Y. : Primal-dual subgradient methods for convex problems. Math. Program., Ser. B **120**, 221–259 (2009)
171. Neumaier, A. : Interval Methods for Systems of Equations. Cambridge University Press, Cambridge, UK (1990)
172. Nievergelt, Y. : A tutorial history of least squares with applications to astronomy and geodesy. Journal of Computational and Applied Mathematics **121**, 37–72 (2000)
173. Nocedal, J., Wright, S. : Numerical Optimization. Springer, New York, NY (1999)
174. Owen, A. : Monte Carlo variance of scrambled net quadratures. SIAM J. Numer. Anal. **34**(5), 1884–1910 (1997)
175. Paige, C., Saunders, M. : Solution of sparse indefinite systems of linear equations. SIAM J. Numer. Anal. **12**(4), 617–629 (1975)
176. Papalambros, P., Wilde, D. : Principles of Optimal Design. Cambridge University Press, Cambridge, UK (1988)
177. Parlett, B. : The QR algorithm. Computing in Science and Engineering **2**(1), 38–42 (2000)
178. Paschos, V. (ed.) : Applications of Combinatorial Optimization. Wiley-ISTE, Hoboken, NJ (2010)
179. Paschos, V. (ed.) : Concepts of Combinatorial Optimization. Wiley-ISTE, Hoboken, NJ (2010)
180. Paschos, V. (ed.) : Paradigms of Combinatorial Optimization : Problems and New Approaches. Wiley-ISTE, Hoboken, NJ (2010)
181. Petzold, L. : Differential/algebraic equations are not ODE's. SIAM J. Scientific and Statistical Computing **3**(3), 367–384 (1982)
182. Pichat, M., Vignes, J. : Ingénierie du contrôle de la précision des calculs sur ordinateur. Editions Technip, Paris, FR (1993)
183. Polak, E. : Computational Methods in Optimization. Academic Press, New York, NY (1971)
184. Polak, E. : Optimization – Algorithms and Consistent Approximations. Springer, New York, NY (1997)
185. Polyak, B. : Introduction to Optimization. Optimization Software, New York, NY (1987)
186. Press, W., Flannery, B., Teukolsky, S., Vetterling, W. : Numerical Recipes. Cambridge University Press, Cambridge, UK (1986)
187. Prince, P., Dormand, J. : High order embedded Runge-Kutta formulae. J. of Computational and Applied Mathematics **7**(1), 67–75 (1981)
188. Pronzato, L., Walter, E. : Eliminating suboptimal local minimizers in nonlinear parameter estimation. Technometrics **43**(4), 434–442 (2001)
189. Pronzato, L., Walter, E., Venot, A., Lebruchec, J.F. : A general purpose global optimizer : implementation and applications. Math. and Comput. in Simulation **26**, 412–422 (1984)
190. Rall, L., Corliss, G. : Introduction to automatic differentiation. In : M. Bertz, C. Bischof, G. Corliss, A. Griewank (eds.) Computational Differentiation Techniques, Applications, and Tools. SIAM, Philadelphia, PA (1996)
191. Ratschek, H., Rokne, J. : New Computer Methods for Global Optimization. Ellis Horwood, Chichester, UK (1988)
192. Reddien, G. : Projection methods for two-point boundary value problems. SIAM Review **22**(2), 156–171 (1980)
193. Rice, J. : A theory of condition. SIAM J. Numer. Anal. **3**(2), 287–310 (1966)
194. Robert, C., Casella, G. : Monte Carlo Statistical Methods. Springer, New York, NY (2004)

195. Rump, S. : INTLAB - INTerval LABoratory. In : T. Csendes (ed.) Developments in Reliable Computing, pp. 77 – 104. Kluwer Academic Publishers (1999)

196. Rump, S. : Verification methods : Rigorous results using floating-point arithmetic. Acta Numerica pp. 287–449 (2010)

197. Russel, R., Shampine, L. : A collocation method for boundary value problems. Numer. Math. **19**, 1–28 (1972)

198. Russell, R., Varah, J. : A comparison of global methods for linear two-point boundary value problems. Mathematics of Computation **29**(132), 1007–1019 (1975)

199. Rustem, B., Howe, M. : Algorithms for Worst-Case Design and Applications to Risk Management. Princeton University Press, Princeton, NJ (2002)

200. Saad, Y. : Preconditioning techniques for nonsymmetric and indefinite linear systems. J. of Computational and Applied Mathematics **24**, 89–105 (1988)

201. Saad, Y. : Iterative Methods for Sparse Linear Systems, second edn. SIAM, Philadelphia, PA (2003)

202. Saad, Y. : Numerical Methods for Large Eigenvalue Problems, second edn. SIAM, Philadelphia, PA (2011)

203. Saad, Y., Schultz, M. : GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Scientific and Statistical Computing **7**(3), 856–869 (1986)

204. Sacks, J., Welch, W., Mitchell, T., Wynn, H. : Design and analysis of computer experiments (with discussion). Statistical Science **4**(4), 409–435 (1989)

205. Santner, T., Williams, B., Notz, W. : The Design and Analysis of Computer Experiments. Springer, New York, NY (2003)

206. Sasena, M. : Flexibility and efficiency enhancements for constrained global design optimization with kriging approximations. Ph.D. thesis, University of Michigan (2002)

207. Sasena, M., Papalambros, P., Goovaerts, P. : Exploration of metamodeling sampling criteria for constrained global optimization. Engineering Optimization **34**(3), 263–278 (2002)

208. Segel, L., Slemrod, M. : The quasi-steady-state assumption : A case study in perturbation. SIAM Review **31**(3), 446–477 (1989)

209. Shampine, L. : What everyone solving differential equations numerically should know. In : I. Gladwell, D. Sayers (eds.) Computational Techniques for Ordinary Differential Equations. Academic Press, London, UK (1980)

210. Shampine, L. : Numerical Solution of Ordinary Differential Equations. Chappman & Hall, New York, NY (1994)

211. Shampine, L. : Error estimation and control for ODEs. J. of Scientific Computing **25**(1/2), 3–16 (2005)

212. Shampine, L. : Vectorized solution of ODEs in MATLAB. Scalable Computing : Practice and Experience **10**(4), 337–345 (2009)

213. Shampine, L., Gear, C. : A user's view of solving stiff ordinary differential equations. SIAM Review **21**(1), 1–17 (1979)

214. Shampine, L., Gladwell, I., Thompson, S. : Solving ODEs in MATLAB. Cambridge University Press, Cambridge, UK (2003)

215. Shampine, L., Kierzenka, J., Reichelt, M. : Solving boundary value problems for ordinary differential equations in MATLAB with bvp4c. Available at http ://www.mathworks.com/ (2000)

216. Shampine, L., Reichelt, M. : The MATLAB ODE suite. SIAM J. Sci. Comput. **18**(1), 1–22 (1997)

217. Shampine, L., Thompson, S. : Event location for ordinary differential equations. Computers and Mathematics with Applications **39**, 43–54 (2000)

218. Shewchuk, J. : An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1994)

219. Shimizu, K., Aiyoshi, E. : Necessary conditions for min-max problems and algorithms by a relaxation procedure. IEEE Trans. Autom. Control **AC-25**(1), 62–66 (1980)

220. Shor, N. : Minimization Methods for Non-differentiable Functions. Springer, Berlin, DE (1985)

221. Skufca, J. : Analysis still matters : A surprising instance of failure of Runge-Kutta-Felberg ODE solvers. SIAM Review **46**(4), 729–737 (2004)

222. Speelpening, B. : Compiling fast partial derivatives of functions given by algorithms. Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana Champaign, IL (1980)

223. Steihaug, T., Wolfbrandt, A. : An attempt to avoid exact Jacobian and nonlinear equations in the numerical solution of stiff differential equations. Mathematics of Computation **33**(146), 521–534 (1979)

224. Stewart, G. : On the early history of the singular value decomposition. SIAM Review **35**(4), 551–566 (1993)

225. Stewart, G. : Afternotes on Numerical Analysis. SIAM, Philadelphia, PA (1996)

226. Stewart, G. : The decomposition approach to matrix computation. Computing in Science and Engineering **2**(1), 50–59 (2000)

227. Stoer, J., Bulirsch, R. : Introduction to Numerical Analysis. Springer, New York, NY (1980)

228. Storn, R., Price, K. : Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization **11**, 341–359 (1997)

229. Strang, G. : Introduction to Applied Mathematics. Wellesley - Cambridge Press, Wellesley, MA (1986)

230. Theodoridis, S., Slavakis, K., Yamada, I. : Adaptive learning in a world of projections. IEEE Signal Processing Mag. **28**(1), 97–123 (2011)

231. Trefethen, L. : Six myths of polynomial interpolation and quadrature. Mathematics Today **47**, 184–188 (2011)

232. Trefethen, L. : Approximation Theory and Approximation Practice. SIAM, Philadelphia, PA (2013)

233. Varah, J. : On the numerical solution of ill-conditioned linear systems with applications to ill-posed problems. SIAM J. Numer. Anal. **10**(2), 257–267 (1973)

234. Vazquez, E., Walter, E. : Estimating derivatives and integrals with Kriging. In : Proc. 44th IEEE Conf. on Decision and Control (CDC) and European Control Conference (ECC), pp. 8156–8161. Seville, Spain (2005)

235. Vetter, W. : Derivative operations on matrices. IEEE Trans. Autom. Control **15**, 241–244 (1970)

236. Vignes, J. : New methods for evaluating the validity of the results of mathematical computations. Math. and Comput. in Simulation **20**(4), 227–249 (1978)

237. Vignes, J. : A stochastic arithmetic for reliable scientific computation. Math. and Comput. in Simulation **35**, 233–261 (1993)

238. Vignes, J., Alt, R., Pichat, M. : Algorithmes numériques, analyse et mise en œuvre, 2 : équations et systèmes non linéaires. Technip, Paris, FR (1980)

239. van der Vorst, H. : Bi-CGSTAB : A fast and smoothly convergent variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM J. Scientific and Statistical Computing **13**(2), 631–644 (1992)

240. van der Vorst, H. : Krylov subspace iteration. Computing in Science and Engineering **2**(1), 32–37 (2000)

241. Wackernagel, H. : Multivariate Geostatistics, third edn. Springer-Verlag, Berlin, DE (2003)

242. Wahba, G. : Spline Models for Observational Data. SIAM, Philadelphia, PA (1990)

243. Walter, E. : Identifiability of State Space Models. Springer, Berlin, DE (1982)

244. Walter, E., Pronzato, L. : Identification of Parametric Models. Springer, London, UK (1997)

245. Walters, F., Parker, L., Morgan, S., Deming, S. : Sequential Simplex Optimization. CRC Press, Boca Raton, FL (1991)

246. Watkins, D. : Understanding the QR algorithm. SIAM Review **24**(4), 427–440 (1982)

247. Watson, L., Bartholomew-Biggs, M., Ford, J. : Optimization and Nonlinear Equations. J. of Computational and Applied Mathematics **124**(1-2), 1–373 (2000)

248. Whitley, L. (ed.) : Foundations of Genetic Algorithms 2. Morgan Kaufmann, San Mateo, CA (1993)

249. Wilkinson, J. : Convergence of the LR, QR, and related algorithms. The Computer Journal **8**, 77–84 (1965)

250. Wilkinson, J. : Modern error analysis. SIAM Review **13**(4), 548–568 (1971)

251. Wilkinson, J. : The perfidious polynomial. In : G. Golub (ed.) Studies in Numerical Analysis, *Studies in Mathematics*, vol. 24, chap. The perfidious polynomial, pp. 1–28. Mathematical Association of America (1984)

252. Wilkinson, J. : Error analysis revisited. IMA Bulletin **22**(11/12), 192–200 (1986)

253. Wilkinson, J. : Rounding Errors in Algebraic Processes, reprinted edn. Dover, New York, NY (1994)

254. Wolpert, D., Macready, W. : No free lunch theorems for optimization. IEEE Trans. on Evolutionary Computation **1**(1), 67–82 (1997)

255. Wright, M. : The interior-point revolution in optimization : History, recent developments, and lasting consequences. Bulletin of the American Mathematical Society **42**(1), 39–56 (2004)

256. Young, D. : Iterative methods for solving partial difference equations of elliptic type. Ph.D. thesis, Harvard University, Cambridge, MA (1950)

257. Ypma, T. : Historical development of the Newton-Raphson method. SIAM Review **37**(4), 531–551 (1995)

258. Zahradnicky, T., Lorencz, R. : FPU-supported running error analysis. Acta Polytechnica **50**(2), 30–36 (2010)

259. Zaslavski, A. : A sufficient condition for exact penalty in constrained optimization. SIAM J. Optimization **16**, 250–262 (2005)

260. Zedan, H. : Modified Rosenbrock-Wanner methods for solving systems of stiff ordinary differential equations. Ph.D. thesis, University of Bristol, Bristol, UK (1982)

# Index