Prolégomènes à l'algorithmique

Les quelques nouveaux qui réussissent à passer le font tout au moins en partie parce qu'ils ont de la chance, mais surtout parce qu'ils font ce qu'il faut : ils excellent à construire des machines à survivre.

R. Dawkins (1976), *Le Gène Égoïste*. [157]

Avec ce chapitre nous commençons l'étude des algorithmes de résolution des problèmes d'optimisation. Quelques réponses aux questions communes à tout algorithme, à leur implémentation ou utilisation, ont été rassemblées ici. Celles relatives à la comparaison des algorithmes en fonction de la vitesse de convergence des suites qu'ils génèrent ou du nombre d'opérations qu'ils requièrent sont respectivement abordées aux sections 5.1 et 5.2. La section 5.3 fait quelques remarques sur le (pré-)conditionnement des problèmes d'optimisation, c'est-à-dire sur les moyens de les transformer analytiquement de manière à les rendre plus faciles à résoudre numériquement et moins sensibles aux erreurs d'arrondi, lesquelles sont inévitablement commises en calcul flottant. Le calcul d'un gradient est une opération fondamentale, mécanique et fastidieuse, de l'optimisation numérique. La méthode de l'état adjoint, qui est souvent la bonne méthode à utiliser dans les problèmes de commande optimale, est exposée à la section 5.4. Il ne faudra jamais perdre de vue qu'il existe une méthode générale, théoriquement efficace, pour évaluer un gradient : la différentiation automatique. Nous détaillons les modes de différentiation direct et inverse à la section 5.5. Nous concluons avec la section 5.6 qui discute de deux aspects des codes d'optimisation : des différentes structures que peut avoir un code utilisant un module d'optimisation et des profils de performance qui apportent une image graphique de l'efficacité relative de solveurs sur un banc d'essai de problèmes-tests.

5.1 Vitesse de convergence des suites

In those days, we needed faster convergence to get results in the few hours between expected failures of the Avidac's large roomful of a few thousand bytes of temperamental electrostatic memory.

W.C. Davidon, dans la nouvelle introduction de son article fondateur des méthodes de quasi-Newton, écrit en 1959, mais qui ne fut publié qu'en 1991, lors de la parution du premier numéro de la revue *SIAM Journal on Optimization* [155, 156].

Les algorithmes d'optimisation génèrent des suites $\{x_k\}_{k\geqslant 1}$ convergeant vers une solution x_* du problème (dans les bons cas !). Ils procèdent ainsi parce que, dans la plupart des problèmes que nous allons rencontrer, il n'est pas possible de calculer une solution en un nombre fini d'opérations arithmétiques. Les éléments x_k de la suite générée sont appelés des *itérés*. Étant donné un itéré x_k, on calcule l'itéré suivant x_{k+1} de telle sorte que celui-ci soit, si possible, plus proche de la solution cherchée x_* que ne l'est x_k.

Se pose alors le problème de comparer l'efficacité des algorithmes par l'examen des suites qu'ils génèrent, dans le pire des cas. La notion de vitesse de convergence permet de qualifier le comportement asymptotique (pour x_k proche de x_*) d'une suite. On distingue deux classes de vitesses de convergence : les vitesses de convergence en quotient (section 5.1.1) et en racine (section 5.1.2). Ces notions sont motivées par des considérations pratiques : on peut en effet démontrer que les suites générées par les algorithmes étudiés dans cet ouvrage ont de telles vitesses de convergence.

5.1.1 Vitesses de convergence en quotient

Dans cette section, on cherche à qualifier la vitesse de convergence d'une suite $\{x_k\}$ d'un espace normé $\mathbb{E}$, de norme notée $\|\cdot\|$, en comparant la norme de l'*erreur* $x_k - x_*$ de deux itérés successifs, c'est-à-dire en comparant $\|x_k - x_*\|$ et $\|x_{k+1} - x_*\|$. On supposera toujours que l'erreur ne s'annule pas ($x_k \neq x_*$, pour tout indice k). Cette hypothèse est raisonnable, car dans les algorithmes bien conçus, dès que $x_k = x_*$, la suite devient stationnaire après x_k (tous les itérés suivants sont égaux à x_*) et il n'y a plus de sens à parler de vitesse de convergence. On s'intéresse donc au *quotient*

$$\frac{\|x_{k+1} - x_*\|}{\|x_k - x_*\|^\alpha}, \tag{5.1}$$

où α est un entier non nul. L'intérêt pour ce quotient provient du fait qu'on peut souvent l'estimer en faisant un développement de Taylor autour de x_* des fonctions définissant le problème que l'on cherche à résoudre et dont x_* est solution.

Les noms des vitesses de convergence de cette section seront préfixés par « q- » pour rappeler qu'il s'agit de vitesse de convergence en quotient.

Numériquement, plus rapide est la convergence, plus vite augmente le *nombre de chiffres significatifs corrects* de x_k, c'est-à-dire de chiffres significatifs identiques à ceux de x_*. Donnons une définition plus précise de cette notion. Si x_k est un vecteur, on ne peut pas définir par un scalaire la correction des chiffres significatifs de toutes

ses composantes, mais on peut le faire *en moyenne* au sens de la norme sur $\mathbb{E}$. On suppose que $x_* \neq 0$ car on ne peut définir ce que sont les chiffres significatifs de zéro. Si $\|x_k - x_*\|/\|x_*\|$ vaut 10^{-4}, on dira que x_k a 4 chiffres significatifs corrects. Ceci conduit à la définition suivante.

Définition 5.1 (nombre de chiffres significatifs corrects) Le *nombre de chiffres significatifs corrects* de x_k par rapport à $x_* \neq 0$ est le nombre réel défini par

$$\sigma_k := -\log_{10} \frac{\|x_k - x_*\|}{\|x_*\|}. \qquad \square$$

Lorsque $x_* \neq 0$, on peut exprimer les vitesses de convergence en quotient en utilisant σ_k plutôt que le quotient (5.1), ce que nous ferons.

Il est parfois intéressant de vérifier numériquement que les suites générées par un algorithme ont bien la vitesse de convergence attendue, celle démontrée théoriquement pour l'algorithme considéré. Bien sûr, c'est une manière de vérifier que l'algorithme est bien implémenté, mais il y a une autre motivation. Par exemple, sous certaines hypothèses de régularité, on verra que l'algorithme de Newton converge q-quadratiquement (théorème 10.2) ; cet algorithme procède par linéarisation de la fonction qu'il cherche à annuler ; vérifier que la convergence des suites générées est bien q-quadratique est alors une indication sur la correction du calcul des dérivées.

Comme on ne connaît pas la solution, on ne peut vérifier la vitesse de convergence en quotient attendue par l'examen du quotient (5.1), qu'en résolvant deux fois le problème ; la première fois pour calculer une approximation précise de la solution x_*, la seconde pour faire l'examen des quotients susmentionnés ; c'est dommage. On pourrait aussi mémoriser tous les itérés et considérer que le dernier itéré est la solution, mais en grande dimension, cette une opération est bien trop gourmande en place mémoire. On peut éviter cette double résolution ou la mémorisation des itérés si l'on arrive à exprimer la vitesse de convergence en termes d'une quantité dont la limite est connue, typiquement nulle. Il en est ainsi si l'algorithme cherche à annuler une fonction

$$F : \mathbb{E} \to \mathbb{E},$$

pourvu que

$$F(x_*) = 0 \qquad \text{et} \qquad F(x) \sim (x - x_*). \tag{5.2}$$

L'écriture $F(x) \sim (x - x_*)$ signifie ici que, pour une norme $|||\cdot|||$ sur $\mathbb{E}$,

$$\exists\, C \geqslant 1,\ \forall\, x \text{ voisin de } x_* :\ C^{-1}|||F(x)||| \leqslant |||x - x_*||| \leqslant C|||F(x)|||. \tag{5.3}$$

En dimension finie, cette propriété ne dépend pas du choix de la norme $|||\cdot|||$ et est vérifiée dans les codnitions énoncées dans la proposition suivante.

Proposition 5.2 (équivalence asymptotique) *Si F est différentiable en x_*, si $F(x_*) = 0$ et si $F'(x_*)$ est inversible, alors (5.3) a lieu.*

Démonstration. Par la différentiablité de F en x_* et la nullité de $F(x_*)$, on a

$$F(x) = F'(x_*) \cdot (x - x_*) + o(\|x - x_*\|).$$

On en déduit directement que $\|F(x)\| = O(\|x - x_*\|)$ pour x voisin de x_*. Inversement, grâce à l'inversibilité de $F'(x_*)$, on a $x - x_* = F'(x_*)^{-1}F(x) + o(\|x - x_*\|)$, d'où l'on déduit que $\|x - x_*\| = O(\|F(x)\|)$. □

Les vitesses de convergence d'une suite $\{x_k\}$ présentées ci-dessous seront également exprimées en termes du logarithme de $\|F(x_k)\|$ pour une fonction F vérifiant (5.2) et une autre norme $\|\cdot\|$, de manière à permettre une vérification numérique de cette convergence.

La mise en évidence de l'équivalence (5.2) n'est pas toujours aussi aisée que dans la démonstration de la proposition 5.2, en particulier lorsque F n'est pas différentiable en x_* dans le sens classique de Fréchet. C'est le cas, par exemple, pour la fonction que l'on cherche à annuler par l'algorithme newtonien en optimisation avec contraintes (chapitre 15). Pour certaines vitesses de convergence de $\{x_k\}$ vers x_*, on dispose d'un autre moyen de les mettre en évidence théoriquement ou numériquement sans la connnaissance de la limte x_*, à savoir par l'examen de la vitesse de convergence vers zéro de la suite $\{s_k\}$ des déplacements

$$s_k := x_{k+1} - x_k. \tag{5.4}$$

L'intérêt de cette suite est, en l'occurrence, de converger vers une limite connue, zéro, lorsque $x_k \to x_*$. L'estimation de la vitesse de convergence peut alors se faire au cours des itérations.

Nous rencontrerons essentiellement trois vitesses de convergence en quotient : les vitesses de convergence q-linéaire, q-superlinéaire et q-quadratique. Le préfixe « q » utilisé dans ces appellations, qui rappelle le mot « quotient », on l'a dit, est parfois omis.

Convergence q-linéaire

Définition 5.3 (convergence q-linéaire) On dit qu'une suite $\{x_k\} \subseteq \mathbb{E}$ converge *q-linéairement* vers x_* s'il existe une norme $\|\cdot\|$, un scalaire $\tau \in [0, 1[$ et un indice $k_1 \geqslant 1$, tels que pour tout $k \geqslant k_1$ on ait

$$\|x_{k+1} - x_*\| \leqslant \tau \|x_k - x_*\|. \tag{5.5}$$

Le paramètre τ est appelé le *taux de convergence linéaire*. □

Il faut donc que la norme de l'erreur décroisse strictement à chaque itération à partir d'une certaine itération, avec un taux de convergence τ strictement plus petit que 1. Cette propriété dépend du choix de la norme que l'on utilise pour mesurer l'erreur, car l'estimation (5.5) peut être vraie pour une norme et, malgré l'équivalence des normes, peut ne plus être vérifiée avec $\tau < 1$ pour une autre norme.

Le résultat suivant fait le lien entre la convergence q-linéaire et le nombre σ_k de chiffres significatifs corrects des itérés (définition 5.1). Sa démonstration est proposée à l'exercice 5.2.

Proposition 5.4 (convergence q-linéaire en termes de σ_k) *La suite $\{x_k\}_{k\geqslant 1}$ converge q-linéairement vers $x_* \neq 0$ pour une norme $\|\cdot\|$ si, et seulement si, il existe une constante $\sigma > 0$ et un indice $k_1 \geqslant 1$ tels que pour tout $k \geqslant k_1$ on ait*

$$\sigma_{k+1} \geqslant \sigma_k + \sigma,$$

où σ_k est défini avec la norme $\|\cdot\|$.

Il est difficile d'établir un lien entre la convergence linéaire de la suite $\{x_k\}$ et celle de la suite $\{F(x_k)\}$ où la fonction F vérifie (5.2), à cause de la constante C intervenant dans (5.3) et du taux de convergence τ qui doit être strictement inférieur à 1. On trouve la même difficulté pour établir une équivalence entre la convergence linéaire de $\{x_k\}$ et celle des déplacements $s_k := x_{k+1} - x_k$.

En général, on s'attend à ce qu'un algorithme d'optimisation différentiable calcule des suites convergeant plus rapidement que q-linéairement.

Exemple d'algorithme générant des suites q-linéairement convergentes

- L'algorithme du gradient pour minimiser une fonction quadratique strictement convexe (proposition 7.2).

Convergence q-superlinéaire

Définition 5.5 (convergence q-superlinéaire) On dit qu'une suite $\{x_k\} \subseteq \mathbb{E}$ converge *q-superlinéairement* vers x_* si pour tout $\tau > 0$, il existe un indice $k_\tau \geqslant 1$, tels que pour tout $k \geqslant k_\tau$ on ait

$$\|x_{k+1} - x_*\| \leqslant \tau \|x_k - x_*\|.$$

Il revient au même de dire que $\|x_{k+1} - x_*\|/\|x_k - x_*\| \to 0$ ou encore que

$$\|x_{k+1} - x_*\| = o(\|x_k - x_*\|). \qquad \square$$

Cette propriété est indépendante du choix de la norme. Clairement, une suite convergeant q-superlinéairement converge q-linéairement.

Le résultat suivant fait le lien entre la convergence q-superlinéaire et le nombre σ_k de chiffres significatifs corrects des itérés (définition 5.1). Sa démonstration est proposée à l'exercice 5.2.

Proposition 5.6 (convergence q-superlinéaire en termes de σ_k) *La suite $\{x_k\}_{k\geqslant 1}$ converge q-superlinéairement vers $x_* \neq 0$ pour une norme $\|\cdot\|$ si, et seulement si,*

$$\sigma_{k+1} - \sigma_k \to +\infty,$$

où σ_k est défini avec la norme $\|\cdot\|$.

Voici une manière de vérifier numériquement la convergence q-superlinéaire d'une suite par l'intermédiaire d'une fonction s'annulant au point limite. La démonstration de la proposition est proposée à l'exercice 5.3.

Proposition 5.7 (convergence q-superlinéaire en termes *F*) *Soit* $F : \mathbb{E} \to \mathbb{E}$ *une fonction vérifiant* (5.2)*. La suite* $\{x_k\}$ *converge q-superlinéairement vers* x_* *si, et seulement si,*

$$\log \|F(x_{k+1})\| - \log \|F(x_k)\| \to -\infty.$$

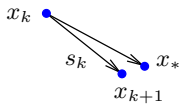
On peut le dire autrement : la suite $\{x_k\}$ converge q-superlinéairement si, et seulement si, le tracé $k \mapsto \log \|F(x_k)\|$ a une « pente » estimée par la différence $\log \|F(x_{k+1})\| - \log \|F(x_k)\|$ qui tend vers $-\infty$.

Pour des suites $\{z_k\}$ et $\{z'_k\}$ dans un espace normé, convergeant vers zéro, on utilisera occasionnellement la notation

$$z_k \sim z'_k \tag{5.6}$$

pour dire qu'il existe une constante $C \geqslant 1$ telle que, pour tout indice k, on a $C^{-1}\|z_k\| \leqslant \|z'_k\| \leqslant C\|z_k\|$, ce que l'on peut aussi écrire $z_k = O(\|z'_k\|)$ et $z'_k = O(\|z_k\|)$. En mots simples : $\{z_k\}$ et $\{z'_k\}$ convergent vers zéro à la même « vitesse ».

Si $\{x_k\}$ converge superlinéairement vers x_*, asymptotiquement, la norme de l'erreur à l'itération suivante $\|x_{k+1} - x_*\|$ est très petite devant celle de l'itération courante $\|x_k - x_*\|$, ce qui implique que le déplacement $s_k := x_{k+1} - x_k$ est proche de $x_* - x_k$ (voir le dessin ci-contre). Ceci a pour conséquence que $(x_k - x_*) \sim s_k$, comme le montre le lemme ci-dessous.



Lemme 5.8 (suites équivalentes) *Si la suite* $\{x_k\}$ *converge superlinéairement vers* x_**, alors* $(x_k - x_*) \sim s_k$*.*

Démonstration. On a

$$s_k = (x_{k+1} - x_*) - (x_k - x_*) = -(x_k - x_*) + o(\|x_k - x_*\|).$$

On en déduit le résultat. □

Cette équivalence entre la suite des erreurs $\{x_k - x_*\}$ et celle des déplacements $\{s_k\}$ permet de montrer que celles-ci convergent superlinéairement vers leur limite respective simultanément.

Proposition 5.9 (convergence q-superlinéaire en termes de s_k)
1) *Si une suite $\{x_k\}$ converge superlinéairement vers x_*, alors la suite des déplacements $\{s_k\}$, définie par (5.4), converge superlinéairement vers zéro.*
2) *Inversement, si, pour une suite donnée $\{x_k\}$, la suite des déplacements $\{s_k\}$ converge superlinéairement vers zéro, alors $\{x_k\}$ est une suite de Cauchy qui converge superlinéairement vers sa limite.*

DÉMONSTRATION. 1) D'après le lemme 5.8, la convergence superlinéaire de x_k vers x_* implique que $s_k \sim (x_k - x_*)$. On déduit alors de $x_{k+1} - x_* = o(\|x_k - x_*\|)$ que $s_{k+1} = o(\|s_k\|)$, qui est la marque de la convergence superlinéaire de s_k vers zéro.

2) On peut supposer que $s_k \neq 0$ pour tout $k \geqslant 1$ (si $s_k = 0$ avec k assez grand, la convergence superlinéaire de $\{s_k\}$ implique alors que $s_i = 0$ pour tout $i \geqslant k$ et donc que la suite $\{x_k\}$ est stationnaire, ce qui implique immédiatement le résultat). Du fait de la convergence superlinéaire de $\{s_k\}$ vers zéro et du fait que l'on s'intéresse au comportement asymptotique des suites, on peut aussi supposer que, pour tout $k \geqslant 1$, on a

$$\varepsilon_k := \sup_{i \geqslant k} \frac{\|s_{i+1}\|}{\|s_i\|} < 1. \tag{5.7}$$

Observons que la suite $\{\varepsilon_k\}$ tend vers zéro en décroissant.

Commençons par montrer que, pour $k \geqslant 1$ et $i \geqslant 1$, on a

$$\|s_{k+i}\| \leqslant \varepsilon_k^i \|s_k\|. \tag{5.8}$$

Fixons $k \geqslant 1$ et montrons l'inégalité par récurrence sur i. Par (5.7), l'inégalité a lieu pour $i = 1$. Supposons maintenant qu'elle ait lieu pour un $i \geqslant 1$ et montrons la pour $i + 1$. Par (5.7), la décroissance de $\{\varepsilon_k\}$ et l'hypothèse de récurrence, nous pouvons conclure

$$\|s_{k+i+1}\| \leqslant \varepsilon_{k+i}\|s_{k+i}\| \leqslant \varepsilon_k \|s_{k+i}\| \leqslant \varepsilon_k^{i+1}\|s_k\|.$$

Montrons à présent que $\{x_k\}$ est une suite de Cauchy, qui converge donc vers une limite que l'on note x_*, et que, pour tout $\varepsilon > 0$, il existe un k_ε tel que, pour tout $k \geqslant k_\varepsilon$, on a

$$\|x_k - x_*\| \leqslant (1 + \varepsilon)\|s_k\|. \tag{5.9}$$

Soit $\varepsilon > 0$. Pour $l > k$ et k suffisamment grand pour que $1/(1 - \varepsilon_k) \leqslant 1 + \varepsilon$, on a

$$\begin{aligned}
\|x_k - x_l\| &= \|\textstyle\sum_{i=0}^{l-k-1} s_{k+i}\| \\
&\leqslant \textstyle\sum_{i=0}^{l-k-1} \|s_{k+i}\| \\
&\leqslant (\textstyle\sum_{i=0}^{l-k-1} \varepsilon_k^i)\|s_k\| \qquad [(5.8)] \\
&\leqslant \tfrac{1}{1-\varepsilon_k}\|s_k\| \qquad [\varepsilon_k < 1] \\
&\leqslant (1 + \varepsilon)\|s_k\| \qquad [1/(1 - \varepsilon_k) \leqslant 1 + \varepsilon].
\end{aligned}$$

Ceci montre que $\{x_k\}$ est une suite de Cauchy, dont la limite est notée x_*. En faisant tendre $l \to \infty$ dans l'inégalité ci-dessus, on obtient (5.9).

Montrons à présent que, pour tout $\varepsilon > 0$, il existe un k_ε tel que, pour tout $k \geqslant k_\varepsilon$, on a

$$\|x_k - x_*\| \geqslant (1-\varepsilon)\|s_k\|. \tag{5.10}$$

Soit $\varepsilon > 0$. Pour $l > k$ et k suffisamment grand pour que $\varepsilon_k/(1-\varepsilon_k) \leqslant \varepsilon$, on a

$$\begin{aligned}
\|x_k - x_l\| &= \|\textstyle\sum_{i=0}^{l-k-1} s_{k+i}\| \\
&\geqslant \|s_k\| - \|\textstyle\sum_{i=1}^{l-k-1} s_{k+i}\| \\
&\geqslant \|s_k\| - \textstyle\sum_{i=1}^{l-k-1} \|s_{k+i}\| \\
&\geqslant (1 - \textstyle\sum_{i=1}^{l-k-1} \varepsilon_k^i)\|s_k\| \qquad [(5.8)] \\
&\geqslant (1 - \tfrac{\varepsilon_k}{1-\varepsilon_k})\|s_k\| \qquad [\varepsilon_k < 1] \\
&\geqslant (1-\varepsilon)\|s_k\| \qquad [\varepsilon_k/(1-\varepsilon_k) \leqslant \varepsilon].
\end{aligned}$$

En faisant tendre $l \to \infty$, on obtient (5.10).

Les inégalités (5.9) et (5.10), avec $\varepsilon \in]0,1[$, montrent que $s_k \sim (x_k - x_*)$. Alors la convergence superlinéaire de $\{s_k\}$ vers zéro, $\|s_{k+1}\| = o(\|s_k\|)$, implique celle de $\{x_k\}$ vers x_*. □

Exemple d'algorithme générant des suites q-superlinéairement convergentes

- Les algorithmes de quasi-Newton en optimisation (proposition **??**) ou pour résoudre un système d'équations non linéaires.

Convergence q-quadratique

Définition 5.10 (convergence q-quadratique) On dit qu'une suite $\{x_k\} \subseteq \mathbb{E}$ converge *q-quadratiquement* vers x_* s'il existe une constante $C \geqslant 0$, telle que pour tout $k \geqslant 1$ on ait

$$\|x_{k+1} - x_*\| \leqslant C\|x_k - x_*\|^2. \qquad \square$$

Le quotient des erreurs successives $\|x_{k+1} - x_*\|/\|x_k - x_*\| \leqslant C\|x_k - x_*\|$ d'une suite quadratiquement convergente tend vers zéro ; une telle suite converge donc q-superlinéairement.

Le résultat suivant fait le lien entre la convergence q-quadratique et le nombre σ_k de chiffres significatifs corrects des itérés (définition 5.1). Sa démonstration est proposée à l'exercice 5.2.

Proposition 5.11 (convergence q-quadratique en termes de σ_k) *La suite $\{x_k\}_{k\geqslant 1}$ converge q-quadratiquement vers $x_* \neq 0$ pour une norme $\|\cdot\|$ si, et seulement si, il existe une constante $C \in \mathbb{R}$ telle que*

$$\sigma_{k+1} \geqslant 2\sigma_k + C,$$

où σ_k est défini avec la norme $\|\cdot\|$. Dans ce cas

$$\liminf_{k\to\infty} \frac{\sigma_{k+1}}{\sigma_k} \geqslant 2. \tag{5.11}$$

Remarque 5.12 (odublement du nombre de chiffres significatifs corrects à chaque itération) De manière imagée, on peut exprimer verbalement l'inégalité (5.11) en disant qu'« une suite $\{x_k\}$ convergeant q-quadratiquement a des éléments x_k dont le nombre de chiffres significatifs corrects double à chaque itération *asymptotiquement* ». C'est une convergence très rapide puisque l'on atteint alors très vite le nombre maximal de chiffres significatifs qu'un ordinateur donné peut représenter (15..16 pour des nombres en double précision pour la *norme IEEE 754*). □

Voici à présent deux manières de vérifier numériquement la convergence q-quadratique d'une suite, sans connaître le point limite x_*. La première méthode (proposition 5.13) utilise une fonction s'annulant au point limite et vérifiant (5.2), ce qui requiert l'existence d'une telle fonction (voir la proposition 5.2). La seconde méthode (proposition 5.14) utilise la suite des déplacements $\{s_k\}$, où $s_k := x_{k+1} - x_k$. Les démonstrations de ces propositions sont proposées aux exercices 5.3 et 5.4.

Proposition 5.13 (convergence q-quadratique en termes F) *Soit $F : \mathbb{E} \to \mathbb{E}$ une fonction vérifiant (5.2). La suite $\{x_k\}$ converge q-quadratiquement vers x_* si, et seulement si, il existe une constante $C \in \mathbb{R}$ telle que*

$$\log \|F(x_{k+1}\| \leqslant 2 \log \|F(x_k)\| + C.$$

Dans ce cas

$$\liminf_{k\to\infty} \frac{\log \|F(x_{k+1})\|}{\log \|F(x_k)\|} \geqslant 2.$$

Proposition 5.14 (convergence q-quadratique en termes de s_k)
1) *Si une suite $\{x_k\}$ converge quadratiquement vers x_*, alors la suite des déplacements $\{s_k\}$, définie par (5.4), converge quadratiquement vers zéro.*
2) *Inversement, si, pour une suite donnée $\{x_k\}$, la suite des déplacements $\{s_k\}$ converge quadratiquement vers zéro, alors $\{x_k\}$ est une suite de Cauchy qui converge quadratiquement vers sa limite.*

Exemple d'algorithme générant des suites q-quadratiquement convergentes

- L'algorithme de Newton en optimisation ou pour résoudre un système d'équations non linéaires (théorèmes 10.2 et 10.3).

Typiquement, ce sont donc les algorithmes qui procèdent par linéarisation des équations à résoudre qui génèrent des suites convergeant q-quadratiquement.

Convergence avec q-ordre plus élevé

Définition 5.15 (convergence avec q-ordre) On dit qu'une suite $\{x_k\} \subseteq \mathbb{E}$ converge avec un *q-ordre* $\alpha \in \mathbb{R}_{++}$ vers x_* s'il existe une constante $C \geqslant 0$, telle que pour tout $k \geqslant 1$ on ait

$$\|x_{k+1} - x_*\| \leqslant C\|x_k - x_*\|^\alpha.$$

On dit que la convergence est *q-cubique* si $\alpha = 3$, qu'elle est *q-quartique* si $\alpha = 4$ et qu'elle est *q-quintique* si $\alpha = 5$. □

Clairement, la convergence avec un q-ordre α implique la convergence avec un q-ordre $\alpha' \in]0, \alpha]$.

Une vitesse de convergence avec un q-ordre $\alpha > 2$ est garantie par certains algorithmes, mais ceux-ci ne se rencontrent pas souvent. D'ailleurs, on cherche rarement à construire un algorithme assurant une telle vitesse de convergence à ses suites, parce que la convergence q-quadratique ($\alpha = 2$) permet d'avoir une solution très précise en très peu d'itérations (remarque 5.12) et requiert déjà une itération qui peut être couteuse en place mémoire et temps de calcul lorsque la dimension du problème est grande.

5.1.2 Vitesses de convergence en racine ▲

On dit que $\{x_k\}$ converge *r-linéairement* vers x_* s'il existe un scalaire $\tau \in [0, 1[$ tel que

$$\limsup_{k\to\infty} \|x_k - x_*\|^{\frac{1}{k}} \leqslant \tau. \tag{5.12}$$

Le scalaire τ est appelé le *taux de convergence* r-linéaire.

Contrairement à la convergence q-linéaire, la notion de convergence r-linéaire ne dépend pas de la norme utilisée pour mesurer l'erreur. En effet, si $|\cdot|$ est une autre norme, il existe une constante $C > 0$ telle que $C|\cdot| \leqslant \|\cdot\|$ (équivalence des normes). Comme $C^{1/k} \to 1$ lorsque $k \to \infty$, (5.12) implique que l'on a aussi $\limsup_{k\to\infty} |x_k - x_*|^{1/k} \leqslant \tau$.

D'autre part, une suite convergeant q-linéairement converge r-linéairement, comme cela se voit sur l'estimation $\|x_k - x_*\| \leqslant r^{k-k_1}\|x_{k_1} - x_*\|$ que l'on a pour les indices assez grand ($> k_1$) d'une suite q-linéairement convergente. Nous avons donc introduit une notion plus faible que la convergence q-linéaire. En fait la convergence r-linéaire est liée à la convergence q-linéaire d'une suite majorant la norme de l'erreur $\|x_k - x_*\|$. C'est parce que l'on peut parfois montrer la convergence q-linéaire d'un majorant de l'erreur que cette notion est utile.

Proposition 5.16 *Soit $\{x_k\}$ une suite convergeant vers x_*. La convergence est r-linéaire si, et seulement si, il existe une suite $\{\beta_k\}$ convergeant q-linéairement vers 0 telle que $\|x_k - x_*\| \leqslant \beta_k$ pour tout $k \geqslant 1$.*

Démonstration. Supposons que $\{x_k\}$ vérifie (5.12) et choisissons $r_1 \in]r, 1[$. Alors il existe un indice k_1 tel que pour tout $k \geqslant k_1$: $\|x_k - x_*\|^{1/k} \leqslant r_1$. En définissant $\beta_k := \|x_k - x_*\|$ si $k < k_1$ et $\beta_k := r_1^k$ si $k \geqslant k_1$, on a $\|x_k - x_*\| \leqslant \beta_k$ et la suite $\{\beta_k\}$ converge q-linéairement vers 0.

Inversement, si $\|x_k - x_*\| \leqslant \beta_k$, avec $\{\beta_k\}$ convergeant q-linéairement vers 0, on a pour k plus grand qu'un indice k_1 et pour un nombre $r \in [0, 1[$:

$$\|x_k - x_*\|^{\frac{1}{k}} \leqslant \beta_k^{\frac{1}{k}} \leqslant r^{\frac{k-k_1}{k}} \beta_{k_1}^{\frac{1}{k}} \to r < 1.$$

□

5.2 Notions de complexité ▲

On peut trouver la solution de certains problèmes d'optimisation, ou une solution à une précision $\varepsilon > 0$ donnée, en un nombre fini d'étapes. Il en va ainsi des problèmes d'optimisation linéaire ou plus généralement d'optimisation quadratique convexe et même de certains problèmes d'optimisation convexe ou non convexe. Dans ce cas, la notion de vitesse de convergence introduite à la section 5.1 n'est plus très pertinente, puisque la suite générée est stationnaire à partir d'un certain indice. Pour apprécier la qualité des algorithmes de résolution de tels problèmes on préfère compter le nombre d'étapes qu'ils requièrent. Une étape de l'algorithme peut être une itération (concept un peu flou), une opération arithmétique, une opération sur un bit, *etc.* Il faudra toujours bien préciser cet unité élémentaire du calcul. La *théorie de la complexité* s'intéresse à ces questions. Dans cette section, nous passons en revue les notions de cette théorie les plus souvent rencontrées en optimisation. Des présentations allant plus en profondeur sont données par Garey et Johnson [229] et Papadimitriou et Steiglitz [473]. Nous nous plaçons ici dans l'esprit du livre de Vavasis [606] qui présente cette théorie en vue de son application à l'optimisation.

5.2.1 Famille de problèmes, machine de Turing

Dans la théorie de la complexité, on regroupe les problèmes en familles dans le but de pouvoir comparer l'efficacité des algorithmes en fonction de la « taille » des problèmes d'une même famille. On parlera par exemple de la *famille* OL *des problèmes d'optimisation linéaire* (ensemble des problèmes qui consistent à minimiser une fonction linéaire sur un polyèdre convexe, voir les chapitres 17 et 18), de la *famille* OQ *des problèmes d'optimisation quadratique* (ensemble des problèmes qui consistent à minimiser une fonction quadratique sur un polyèdre convexe, voir le chapitre **??**), *etc.* Un problème d'une famille est spécifié par un jeu de données, dont la structure dépend de la famille considérée et dont la représentation ou l'*encodage* doit être précisé.

Étant donné la diversité des problèmes et la variété des questions que l'on peut se poser sur un problème donné, les spécialistes de la théorie de la complexité ont ressenti le besoin de décrire plus formellement ce qu'est une famille de problèmes. Un problème d'une famille doit pouvoir être décrit par un nombre fini de caractères d'un alphabet donné. Un problème ainsi décrit est appelé une *instance* de la famille de problèmes. De manière formelle, une *famille de problèmes* est alors une application

$$F : I \to S$$

entre la collection I des instances de la famille et un certain ensemble S qui pourra dépendre de la famille considérée et de ce que l'on considère comme une solution du problème. En réalité, « évaluer F en i » signifie le plus souvent « résoudre le problème i de la famille ». Considérons par exemple, la famille OL des problèmes d'optimisation linéaire.

Famille 5.17 (OL) On se donne deux entiers $n \geqslant 1$ et $m \geqslant 0$, $c \in \mathbb{Q}^n$, $A \in \mathbb{Q}^{m \times n}$ et $b \in \mathbb{Q}^m$. On considère le problème d'optimisation

$$\begin{cases} \inf\ c^{\mathsf{T}}x \\ Ax = b \\ x \geqslant 0. \end{cases}$$

Une instance i de la famille OL pourra prendre la forme d'un quintuplet $(n, m, c, A, b) \in \mathbb{N}^* \times \mathbb{N} \times \mathbb{Q}^n \times \mathbb{Q}^{mn} \times \mathbb{Q}^m$. L'ensemble S pourra être $\mathbb{Q} \cup \{-\infty, +\infty\}$, si l'on s'intéresse à la valeur optimale $F(i)$ du problème et au fait qu'il peut ne pas être réalisable ($F(i) = +\infty$) ou ne pas être borné ($F(i) = -\infty$). Il pourra aussi être $\mathbb{Q}^n \cup \{-\infty, +\infty\}$, si l'on s'intéresse à la solution x du problème et au fait que ce problème peut ne pas être réalisable ou borné. □

Dans la famille OL, on a choisi de décrire les problèmes en prenant l'ensemble des nombres rationnels $\mathbb{Q}$ comme alphabet. On aurait aussi pu choisir l'ensemble $\{0, 1\}$ des bits représentant ces rationnels. Les nombres réels $\mathbb{R}$ sont parfois utilisés [67], mais il n'est pas clair que l'on puisse alors bien faire la distinction entre la minimisation de problèmes d'optimisation quadratiques convexes et non convexes [606]. Quant aux nombres flottants, utilisés par les ordinateurs, ils semblent peu adaptés à l'étude de la complexité pour des raisons qui sont discutées dans [606 ; page 33 et chapitre 3].

Plus un problème est de grande dimension, plus il faut de caractères pour le décrire. Comme on est particulièrement intéressé par l'influence de la dimension d'un problème sur le nombre d'opérations à effectuer pour le résoudre, la longueur de l'instance jouera un rôle important. On la note

$$L(i) \quad \text{ou} \quad L,$$

selon qu'il est important ou pas de préciser l'instance considérée.

On dira qu'une famille $F : I \to S$ est formée de *problèmes de décision* si S est formé de deux éléments, traditionnellement dénommés `vrai` et `faux`. La théorie de la complexité traite principalement de ces familles de problèmes. Un problème d'optimisation « $\inf\{f(x) : x \in X\}$ » n'est cependant pas un problème de décision. Pour ce ramener à cette notion, on considère parfois la *version décisionnelle* d'un problème d'optimisation : étant donnés f, X et un seuil σ, il s'agit de déterminer si $\inf\{f(x) : x \in X\} \leqslant \sigma$.

Pour parler de la complexité d'un problème calculatoire et donc compter le nombre d'opérations à réaliser pour le résoudre, il faut spécifier la machine effectuant les calculs en précisant les opérations qui peuvent y être exécutées. Le nombre d'opérations à effectuer pour évaluer $F(i)$ sera d'autant plus grand que le nombre d'opérations licites est réduit. Par exemple, on peut considérer que l'addition de deux entiers est une unique opération ou que ce sont les opérations sur les bits représentant ces entiers qui sont les opérations élémentaires. Dans le premier cas, le nombre d'opérations élémentaires pour réaliser une addition est égal à un; dans le second cas, il dépendra de la grandeur ou longueur des entiers.

On a imaginé différents modèles de machines, mais celui qui semble le mieux refléter les capacités des ordinateurs actuels pour résoudre les problèmes que l'on se pose est, selon les spécialistes de ces questions, celui de la *Machine de Turing* [610 ; 2009]. Dans cette brève introduction, nous n'aurons jamais besoin de savoir ce que peut faire exactement une telle machine, ni quelles sont ses opérations licites, si bien que nous renvoyons le lecteur aux ouvrages cités au début de cette section pour une description précise de ce calculateur. On peut toutefois le décrire approximativement.

Il prend en entrée la description de l'instance i du problème à résoudre et il fournit en sortie la réponse $F(i)$. Le calculateur dispose d'une *mémoire linéaire* et d'un *pointeur* adressant une case-mémoire. Il agit en fonction de son *état*, qui fait partie d'une liste *finie*, et du *symbole* de la case-mémoire adressée par le pointeur. Il y a une *liste finie de règles*, définissant ce que le calculateur doit faire en fonction de chaque couple état-symbole possible, ce qui signifie que la machine est *déterministe*. Les actions du calculateur comprennent la modification du contenu de la case-mémoire sous le pointeur, le déplacement du pointeur d'un nombre fini de positions, la modification de ses états, *etc.*

5.2.2 Classes P et NP

On dit qu'une famille de problèmes $F : I \to S$ peut être *évaluée en temps polynomial* pour une machine de Turing de référence, si celle-ci peut évaluer cette fonction en toute instance $i \in I$, et que cette évaluation se fait en un nombre fini d'opérations qui est majoré par une fonction polynomiale de $L(i)$.

Définition 5.18 (classe P, problème polynomial) On dit qu'une famille de problèmes de décision $F : I \to \{\texttt{vrai}, \texttt{faux}\}$ est dans P, on dit aussi que ses problèmes sont *polynomiaux*, si F peut être évaluée en temps polynomial (pour une machine de Turing de référence). □

Beaucoup de familles de problèmes sont dans P et on peut voir là une première raison d'introduire cette classe. Très souvent le degré du polynôme est faible, si bien que ces problèmes peuvent être résolus pour des dimensions (ou plus précisément, des longueurs d'instance L) assez élevées. Les propriétés des polynômes rendent aussi cette notion très souple. Ainsi, parce qu'une composition de polynômes est un polynôme, un algorithme sera polynomial s'il se décompose en un nombre polynomial de modules pouvant chacun être exécuté en temps polynomial de degré constant.

Exemples 5.19 *Familles de problèmes* P : OL (optimisation linéaire, voir le chapitre 18), OQC (optimisation quadratique convexe). □

Pour un alphabet $\mathcal{A}$, on note $\mathcal{A}^*$ l'ensemble de toutes les chaînes finies formées de caractères de $\mathcal{A}$.

Définition 5.20 (classe NP) On dit qu'une famille de problèmes de décision $F : I \to \{\texttt{vrai}, \texttt{faux}\}$ est dans NP, on dit aussi que ses problèmes sont *NP*, s'il existe un polynôme p, un alphabet fini $\mathcal{A}$ et une fonction $G : I \times \mathcal{A}^* \to \{\texttt{vrai}, \texttt{faux}\}$ qui peut être évaluée en temps polynomial, tels que

(NP$_1$) $\forall i \in I$ tel que $F(i) = \texttt{vrai}$, $\exists j \in \mathcal{A}^*$ tel que $L(j) \leqslant p(L(i))$ et $G(i, j) = \texttt{vrai}$,
(NP$_2$) $\forall i \in I$ tel que $F(i) = \texttt{faux}$, $\forall j \in \mathcal{A}^*$, on a $G(i, j) = \texttt{faux}$. □

Cette définition compliquée mérite quelques éclaircissements.

1. On ne demande pas ici que F soit polynomial (F serait alors dans la classe P), mais que G le soit.

2. On peut interpréter l'évaluation de $F(i)$ comme la recherche d'une solution de l'instance $i \in I$, alors que l'évaluation de $G(i,j)$ peut être interprétée comme une vérification que $j \in \mathcal{A}^*$ est une solution de i : si i a une solution (c.-à-d., $F(i) = \texttt{vrai}$), disons $j \in \mathcal{A}^*$ (avec la condition $L(j) \leqslant p(L(i))$, de manière à ce que l'on ne puisse pas prendre pour j, hormis cas triviaux, la réunion de tous les candidats-solutions possibles), on peut vérifier en temps polynomial qu'il en est ainsi (c.-à-d., $G(i,j) = \texttt{vrai}$); si i n'a pas de solution (c.-à-d., $F(i) = \texttt{faux}$), on peut vérifier en temps polynomial qu'un candidat-solution arbitraire $j \in \mathcal{A}^*$ n'est en fait pas une solution (c.-à-d., $G(i,j) = \texttt{faux}$). De manière approximative (en négligeant (NP_2) en particulier), on peut dire qu'on ne demande pas que les instances des familles de la classe NP soient résolubles en temps polynomial, mais que l'on puisse vérifier en temps polynomial qu'une solution en est effectivement une (c'est (NP_1), cette description passe aussi sous silence la condition $L(j) \leqslant p(L(i))$).
3. On a
$$\mathsf{P} \subseteq \mathsf{NP}.$$
En effet, si F est polynomial, on peut prendre pour $\mathcal{A}$ un alphabet fini quelconque et définir G par $G(i,j) = F(i)$ pour tout $(i,j) \in I \times \mathcal{A}^*$.
4. On ne sait pas si $\mathsf{P} = \mathsf{NP}$ (beaucoup pense aujourd'hui que $\mathsf{P} \neq \mathsf{NP}$). Beaucoup de familles de problèmes sont dans NP, sans que l'on sache si elles sont dans P. Cela justifie l'introduction de cette notion.

La définition de la classe NP traite des familles de problèmes de décision. Il ne semble pas simple de l'étendre aux problèmes qui ne le soient pas.

5.2.3 Problèmes NP-complets et NP-ardus

On dit qu'une famille de problèmes de décision $F_1 : I_1 \to \{\texttt{vrai}, \texttt{faux}\}$ est *polynomialement réductible* en une autre famille de problèmes de décision $F_2 : I_2 \to \{\texttt{vrai}, \texttt{faux}\}$, s'il existe une fonction $\rho : I_1 \to I_2$ transformant *toutes* les instances de I_1 en instances de I_2 avec les deux propriétés suivantes :

(PR_1) $\forall i_i \in I_1$, $\rho(i_1)$ peut être évaluée en temps polynomial,
(PR_2) $\forall i_i \in I_1$, $F_1(i_1) = \texttt{vrai}$ si, et seulement si, $F_2(\rho(i_1)) = \texttt{vrai}$.

L'opérateur ρ est appelé une *réduction*. On peut montrer que cette relation est transitive : si F_1 est polynomialement réductible en F_2 et si F_2 est polynomialement réductible en F_3, alors F_1 est polynomialement réductible en F_3. D'autre part, il est clair que si F_1 est polynomialement réductible en F_2 et si $F_2 \in \mathsf{P}$, alors $F_1 \in \mathsf{P}$.

Définition 5.21 (classe NPC, problème NP-complet) On dit qu'une famille de problèmes de décision $F : I \to \{\texttt{vrai}, \texttt{faux}\}$ est dans NPC, on dit aussi que ses problèmes sont *NP-complets*, si elle est dans NP et si toute famille de problèmes dans NP est polynomialement réductible en F. □

En quelque sorte, c'est la sous-classe des familles de problèmes « les plus difficiles » de NP. Si on pouvait montrer qu'une famille de problèmes de NPC est dans P, alors on aurait $\mathsf{P} = \mathsf{NP}$. D'autre part, grâce à la relation de transitivité énoncée ci-dessus,

$$\left.\begin{array}{r} F_1 \in \mathsf{NPC} \\ F_2 \in \mathsf{NP} \\ F_1 \text{ polynomialement réductible en } F_2 \end{array}\right\} \quad \Longrightarrow \quad F_2 \in \mathsf{NPC}.$$

Pour montrer qu'une famille F est dans NPC, on utilise souvent cette dernière implication : on montre que $F \in \mathsf{NP}$ et on montre qu'une famille de NPC est polynomialement réductible en F. Il faut pour cela connaître des problèmes NPC. En voici quelques-uns.

Exemples 5.22 *Familles de problèmes* NP*-complets* : une version décisionnelle de OQ (proposition **??**), CLIQUE, SOUS-SOMME (famille 5.24). □

Famille 5.23 (SAT) Un *littéral* est une variable booléenne v (pouvant prendre la valeur 0 ou 1) ou sa négation (notée $\neg v := 1 - v$). Une *clause* est un OU (noté $\vee$; $v_1 \vee v_2 = 1$ sauf si $v_1 = v_2 = 0$) entre un nombre fini de littéraux (par exemple $v_1 \vee \neg v_2$). Enfin, une *forme normale conjonctive* est un ET (noté $\wedge$; $v_1 \wedge v_2 = 1$ si, et seulement si, $v_1 = v_2 = 1$) entre un nombre fini de clauses; par exemple

$$(v_1) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_3).$$

Un problème SAT consiste à déterminer s'il existe une affectation des variables booléennes v_i d'une forme normale conjonctive qui lui donne la valeur 1. Il s'agit d'une famille de problèmes NP*-complets*. □

Famille 5.24 (SOUS-SOMME) On se donne un nombre fini $n \geqslant 1$ d'entiers $a_1, \ldots, a_n$, un entier b et on se pose la question de savoir si l'on peut trouver un sous-ensemble d'indices $I \subseteq \{1, \ldots, n\}$ tel que

$$\sum_{i \in I} a_i = b.$$

Il s'agit donc d'un problème de décision. □

Beaucoup de problèmes de calcul scientifique, en particulier les problèmes d'optimisation, ne sont pas des problèmes de décision. Afin d'éviter de passer par une version décisionnelle de ces problèmes, qui est parfois éloignée du problème que l'on se pose au départ, on introduit des notions de complexité qui leurs sont propres, comme la suivante.

Définition 5.25 (classe NP-ardu, problème NP-ardu) On dit qu'une famille de problèmes $F : I \to S$ est dans NP-ardu, on dit aussi que ses problèmes sont *NP-ardus*, si une (ou n'importe quelle) famille de problèmes de NPC peut être évaluée en temps polynomial par une machine de Turing utilisant F comme un oracle (c'est-à-dire que cette machine est supposée évaluer F en une étape). □

Par conséquent, une famille F de NP-ardu doit être formée de problèmes suffisamment difficiles à résoudre pour que, lorsque évaluer $F(i)$ devient une opération élémentaire du calculateur, les problèmes de NPC deviennent polynomiaux pour de tels calculateurs. Évidemment, on a

$$\mathsf{NPC} \subseteq \mathsf{NP\text{-}ardu}$$

puisque si $F \in \mathsf{NPC}$, on peut évaluer F (un problème de NPC) en une opération (donc en temps polynomial) en utilisant F comme oracle.

Exemple 5.26 *Famille de problèmes* NP-*ardus* : OQ (proposition **??**). □

Nous avons représenté à la figure 5.1 les liens entre les différentes classes de familles

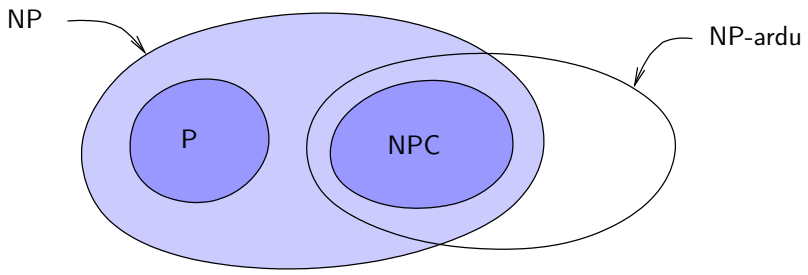


Fig. 5.1. Quelques classes de familles de problèmes si P ≠ NP

de problèmes présentées dans cette section, en supposant que P ≠ NP (ce qui peut être remis en question dans l'avenir). Si P = NP, les trois classes P, NP et NPC sont confondues.

5.3 Conditionnement d'un problème d'optimisation ▲

5.3.1 Notions de conditionnement

Le calcul numérique des solutions d'un problème (d'optimisation ou pas) se fait sur des calculateurs qui utilisent des nombres flottants, avec une précision finie donc. Le *conditionnement* d'un problème est un indicateur de l'effet de cette précision finie sur le résultat. On cherche par cette mesure à savoir si une petite perturbation des données (celle due à cette présicion finie par exemple) peut avoir une grande incidence sur la ou les solutions. Cette incidence dépendra du problème, mais peut aussi dépendre de l'algorithme qui le résout. On comprend alors pourquoi il n'y a pas une unique définition du conditionnement, ou que l'on fait parfois allusion à celui-ci de manière peu précise, ou encore que celui-ci sert de bouc émissaire permettant de justifier le mauvais comportement d'une approche algorithmique !

Résoudre un problème d'optimisation quadratique convexe sans contrainte revient à résoudre sa condition d'optimalité qui est un système linéaire (chapitre 8). Le conditionnement d'un tel problème d'optimisation est, par définition, celui défini pour un système linéaire $Ax = b$. L'effet d'une perturbation de b sur la solution x est estimé par la proposition suivante.

Proposition 5.27 (conditionnement d'un système linéaire) *Soit* $\|\cdot\|$ *une norme matricielle subordonnée à une norme vectorielle notée de la même manière*

re. Si A est inversible, $b \neq 0$, $x = A^{-1}b$ et $x' = A^{-1}b'$, alors

$$\frac{\|x' - x\|}{\|x\|} \leqslant \|A\| \, \|A^{-1}\| \, \frac{\|b' - b\|}{\|b\|}.$$

DÉMONSTRATION. Clairement $x' - x = A^{-1}(b' - b)$, si bien que $\|x' - x\| \leqslant \|A^{-1}\| \, \|b' - b\|$. On obtient le résultat en utilisant l'inégalité $\|b\| \leqslant \|A\| \, \|x\|$. □

Le scalaire $\|A\| \, \|A^{-1}\|$ permet donc d'avoir une estimation de l'erreur *relative* commise sur x à partir de l'erreur relative sur b. On le prend comme définition du conditionnement de A.

Définition 5.28 (conditionnement d'une matrice) *Le conditionnement d'une matrice inversible A pour une norme matricielle $\|\cdot\|$ est le nombre*

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

On note $\kappa_p(A)$ le conditionnement de A pour la norme matricielle ℓ_p.

Si $\|\cdot\|$ est subordonnée à une norme vectorielle, $\kappa(A) \geqslant 1$ (en effet $1 = \|I\| = \|AA^{-1}\| \leqslant \|A\| \, \|A^{-1}\|$).

C'est la même notion que pour la résolution d'un système linéaire. La définir comme distance à l'ensemble des problèmes « singuliers ». Par exemple, Kahan [340 ; 1966] montre que le conditionnement ℓ_p mesure la distance relative en norme ℓ_p à l'ensemble des matrices singulières :

$$\kappa_p(A) = \min_{X \text{ singular}} \frac{\|X - A\|_p}{\|A\|_p}.$$

On appelle *préconditionnement*, toute méthode destinée à améliorer le conditionnement du problème. En général, on transforme le problème en un problème équivalent dont le conditionnement est meilleur.

Noter ce qu'en dit Moré [438 ; 1983; Section 2: scaling and preconditioning]. Dans N et qN, noter que ces algorithmes sont invariants par changement de variables, localement.

5.4 Calcul de dérivées par état adjoint

La méthode de l'état adjoint est une technique souvent utilisée pour calculer les dérivées d'une fonction dépendant *implicitement* des variables par rapport auxquelles on veut la dériver. Elle permet d'éviter le calcul coûteux de la dérivée de la fonction implicite et est d'une utilisation constante dans les problèmes de commande optimale [384, 398]. Nous allons dans un premier temps donner un cadre formel au problème que nous nous posons, ainsi que des exemples de problèmes d'optimisation

où cette technique peut être utile (section 5.4.1). À la section 5.4.2, nous montrons comment l'état adjoint peut être utilisé pour calculer un gradient et ceci est illustré sur des problèmes de commande optimale à la section 5.4.3. Enfin, le calcul de produits hessienne-vecteur par état adjoint est abordé à la section 5.4.4.

5.4.1 Position du problème

Soient $\mathbb{Y}$, $\mathbb{U}$ et $\mathbb{Z}$ trois espaces vectoriels et

$$F : \mathbb{Y} \times \mathbb{U} \to \mathbb{Z} : (y, u) \mapsto F(y, u)$$

une fonction régulière utilisée pour décrire un système par ce que l'on appelle son *équation d'état*

$$F(y, u) = 0. \tag{5.13}$$

Cela peut être l'équation d'équilibre d'un système statique ou l'équation d'évolution d'un modèle dynamique. Dans le cadre des problèmes de commande optimale, les variables u sont appelées *variables de commande* du système et les variables y sont appelées *variables d'état* du système.

Ce qui distingue y et u est le rôle que jouent ces variables dans l'équation d'état. Pour une valeur donnée aux variables de commande u, on suppose que l'équation (5.13) détermine l'état y du système. Ceci sera certainement vrai, localement, dans le voisinage d'un couple état-commande (y_0, u_0) vérifiant (5.13), si F est C^1 dans un voisinage de (y_0, u_0) et si la jacobienne de F en (y_0, u_0) par rapport à y

$$F'_y(y_0, u_0) \text{ est inversible.} \tag{5.14}$$

En effet, on sait alors, en vertu du théorème des fonctions implicites (théorème C.14), qu'il existe une unique *fonction implicite*

$$u \in U_0 \mapsto y(u) \in Y_0 \tag{5.15}$$

définie dans un voisinage U_0 de u_0 et à valeurs dans un voisinage Y_0 de y_0. Celle-ci vérifie

$$\begin{cases} y(u_0) = y_0 \\ F(y(u), u) = 0, \quad \forall u \in U_0. \end{cases} \tag{5.16}$$

Cette fonction implicite décrit l'état $y(u)$ du système lorsque la commande u varie dans U_0.

Supposons à présent que l'on donne une fonction scalaire φ dépendant de y et de u :

$$\varphi : \mathbb{Y} \times \mathbb{U} \to \mathbb{R} : (y, u) \mapsto \varphi(y, u).$$

Si y est fonction implicite de u pour l'équation d'état (5.13), on peut considérer φ comme une fonction de u seul, ou plus précisément considérer la fonction $\psi : U_0 \to \mathbb{R}$ définie par

$$\psi(u) = \varphi(y(u), u).$$

Le problème que l'on se pose est de calculer, *de manière efficace*, le gradient de ψ par rapport à u.

Commençons par calculer les dérivées directionnelles de ψ. Si $v \in \mathbb{U}$, on a

$$\psi'(u)\cdot v = \varphi'_y(y(u),u)\cdot y'(u)\cdot v + \varphi'_u(y(u),u)\cdot v. \tag{5.17}$$

Donnons nous un produit scalaire $\langle\cdot,\cdot\rangle$ sur $\mathbb{U}$ pour pouvoir calculer le gradient de ψ. Nous aurons également besoin d'un produit scalaire sur $\mathbb{Y}$ que l'on notera de la même manière. On note alors A^* l'adjoint d'un opérateur linéaire $A \in \mathcal{L}(\mathbb{U},\mathbb{Y})$ pour ces produits scalaires. Avec $y = y(u)$, on a

$$\nabla\psi(u) = y'(u)^*\nabla_y\varphi(y,u) + \nabla_u\varphi(y,u).$$

Si cette formule contenait $y'(u)$ au lieu de $y'(u)^*$, il s'agirait simplement d'évaluer une dérivée directionnelle de y, ce qui n'est pas une opération coûteuse : si la direction est w, $y'(u)\cdot w$ s'obtient en résolvant un seul système linéaire

$$F'_y(y,u)\,(y'(u)\cdot w) = -F'_u(y,u)\cdot w.$$

Mais avec $y'(u)^*$, il semble bien qu'il faille évaluer $y'(u)$ complètement, ce qui demande a priori le calcul de $\dim\mathbb{U}$ dérivées directionnelles ou encore de $\dim\mathbb{U}$ systèmes linéaires :

$$F'_y(y,u)y'(u) = -F'_u(y,u). \tag{5.18}$$

La méthode de l'état adjoint que nous présentons ci-après permet d'éviter ces $\dim\mathbb{U}$ résolutions. La présence de l'adjoint d'un opérateur de dérivation montre en fait qu'il s'agit d'une « dérivation cotangente », situation à laquelle la méthode de l'état adjoint est bien adaptée.

Avant cela donnons deux exemples dans le domaine de l'optimisation où cette technique peut être utilisée.

Optimisation avec contraintes d'égalité

Supposons que l'on cherche à résoudre le problème d'optimisation sous contrainte d'égalité suivant

$$\begin{cases} \min f(x) \\ F(x) = 0, \end{cases}$$

où x se partitionne en $x = (y,u)$ et $F'_y(y,u)$ est inversible. L'approche est particulièrement efficace lorsque F est linéaire en y, puisque dans ce cas l'évaluation de la fonction implicite $u \mapsto y(u)$ peut se faire en résolvant un unique système linéaire.

Dans ce cas, il est possible et souvent préférable de remplacer le problème avec contraintes d'égalité en x ci-dessus par le problème en u sans contrainte équivalent

$$\min_u f(y(u),u),$$

où $y(u)$ est la fonction implicite donnée par l'équation d'état $F(y,u) = 0$. Ce problème est en effet d'une dimension ($\dim\mathbb{U}$) qui peut être beaucoup plus petite que celle du problème original ($\dim\mathbb{Y} + \dim\mathbb{U}$).

Lors de la résolution numérique de ce problème, le calcul du gradient de $u \mapsto \psi(u) = f(y(u),u)$ se fera sans difficulté par la technique de l'état adjoint (avec $\varphi = f$) si le calcul de l'état $y(u)$ associé à une commande u est aisé, ce qui est souvent le cas lorsque F est linéaire en y. Si F n'est pas linéaire en y, il sera souvent nécessaire d'utiliser un processus itératif pour calculer $y(u)$, si bien que la technique de l'état adjoint s'impose moins.

Optimisation avec contraintes d'égalité et d'inégalité

La même idée peut être utilisée si l'on a en plus des contraintes d'inégalité *en petit nombre* :

$$\begin{cases} \min f(x) \\ F(x) = 0 \\ c(x) \leqslant 0, \end{cases}$$

où c est à valeurs vectorielles. On suppose toujours que x se partitionne en $x = (y, u)$ et que F'_y est inversible. Ici aussi on aura intérêt à ce que F soit linéaire en y.

Dans ce cas également, il pourra être préférable de remplacer le problème original par le problème équivalent en u seulement :

$$\begin{cases} \min f(y(u), u) \\ c(y(u), u) \leqslant 0. \end{cases}$$

Les gradients de $u \mapsto f(y(u), u)$ et des $u \mapsto c_i(y(u), u)$ se calculeront par la technique de l'état adjoint (avec $\varphi = f$ ou un des c_i), d'où la nécessité de ne pas avoir trop de contraintes. Il n'est pas nécessaire que c soit linéaire en y.

5.4.2 Calcul de gradients

Étant donné l'importance pratique de cette méthode, très simple, nous donnons deux approches permettant de déduire les formules. La première est la plus rapide dans ce cadre formel, mais fait intervenir l'expression de la dérivée de la fonction implicite donnée par (5.18), qu'il n'est pas toujours aisé d'écrire. Celle-ci n'est pas nécessaire dans la seconde approche, qui s'avère souvent la plus utile dans les applications.

Calcul direct

On remplace dans (5.17) la valeur de $y'(u)$ donnée par (5.18), à savoir

$$y'(u) = -F'_y(y, u)^{-1} F'_u(y, u).$$

Ceci donne

$$\psi'(u) \cdot v = \varphi'_y(y(u), u) \cdot \left(-F'_y(y, u)^{-1} F'_u(y, u) \cdot v \right) + \varphi'_u(y(u), u) \cdot v.$$

Avec le produit scalaire $\langle \cdot, \cdot \rangle$ supposé donné et en simplifiant les notations par $y \equiv y(u)$, cette relation s'écrit aussi

$$\begin{aligned} \langle \nabla\psi(u), v \rangle &= -\langle \nabla_y \varphi(y, u), F'_y(y, u)^{-1} F'_u(y, u) \cdot v \rangle + \langle \nabla_u \varphi(y, u), v \rangle \\ &= -\langle F'_u(y, u)^* \left(F'_y(y, u)^{-1} \right)^* \nabla_y \varphi(y, u), v \rangle + \langle \nabla_u \varphi(y, u), v \rangle \\ &= \langle F'_u(y, u)^* p, v \rangle + \langle \nabla_u \varphi(y, u), v \rangle, \end{aligned}$$

si p est solution du système linéaire

$$F'_y(y, u)^* p = -\nabla_y \varphi(y, u). \tag{5.19}$$

Cette équation est appelée l'*équation de l'état adjoint* ou plus simplement l'*équation adjointe* (elle fait intervenir l'opérateur adjoint de $F'_y(y, u)$). Elle est linéaire en p.

Pour u donné et $y = y(u)$ calculé, elle permet de déterminer $p = p(u)$, appelé *état adjoint*. Alors, d'après ce qui précède, le gradient de ψ en u s'écrit comme fonction de u, $y = y(u)$ et de $p = p(u)$:

$$\nabla\psi(u) = F'_u(y,u)^* p + \nabla_u \varphi(y,u). \tag{5.20}$$

Utilisation d'un lagrangien

Pour $p \in \mathbb{Z}$, on introduit le *lagrangien*

$$\ell(y,u,p) = \varphi(y,u) + \langle p, F(y,u)\rangle.$$

La valeur du vecteur p sera choisie plus tard. Elle dépendra du point u_0 où l'on veut calculer le gradient de ψ, mais dans la démarche qui suit on considère p indépendant de u. Si $u \mapsto y(u)$ est une fonction implicite pour (5.13), on a grâce à (5.16)

$$\psi(u) = \ell(y(u),u,p), \quad \forall u \in U_0,$$

si bien que le gradient de ψ peut aussi s'obtenir en calculant celui de $u \mapsto \ell(y(u),u,p)$. On choisira p de manière à ce que ce dernier calcul soit simple.

Pour $v \in \mathbb{U}$, on a en $y = y(u)$

$$\begin{aligned} \psi'(u)\cdot v &= \varphi'_y(y,u)\cdot y'(u)\cdot v + \varphi'_u(y,u)\cdot v + \langle p, F'_y(y,u)\cdot y'(u)\cdot v + F'_u(y,u)\cdot v\rangle \\ &= \langle \nabla_u\varphi(y,u), v\rangle + \langle F'_u(y,u)^* p, v\rangle + \langle \nabla_y\varphi(y,u) + F'_y(y,u)^* p, y'(u)\cdot v\rangle, \end{aligned}$$

où on a regroupé les termes dépendant de $y'(u)\cdot v$. L'idée maîtresse de cette approche est maintenant de faire disparaître les termes où $y'(u).v$ intervient (on se rappelle qu'on essaye d'éviter le calcul de $y'(u)$!), en *choisissant* p de telle sorte que le dernier terme s'annule :

$$F'_y(y,u)^* p = -\nabla_y \varphi(y,u).$$

On retrouve l'équation adjointe (5.19). Alors, d'après ce qui précède le gradient de ψ en u s'écrit comme fonction de u, $y = y(u)$ et de $p = p(u)$:

$$\nabla\psi(u) = \nabla_u\varphi(y,u) + F'_u(y,u)^* p.$$

On retrouve (5.20).

Résumé des opérations

Pour résumer, la méthode de l'état adjoint pour le calcul de $\nabla\psi(u_0)$ se déroule en trois étapes. On suppose $u_0 \in \mathbb{U}$ donné. Ensuite, on on procède de la manière suivante.

Schéma 5.29 (calcul d'un gradient par état adjoint)

1. On calcule l'état y_0, solution de l'équation d'état (5.13), avec $u = u_0$, qui peut être non linéaire.

2. On calcule l'état adjoint p_0, solution de l'équation adjointe (5.19), avec $u = u_0$ et $y = y_0$, qui est toujours linéaire en p.
3. On calcule le gradient de ψ en u_0 par la formule (5.20), avec $u = u_0$, $y = y_0$ et $p = p_0$.

Comme annoncé, ce calcul ne fait pas intervenir $y'(u_0)$.

On se rappelle souvent les formules (5.19) et (5.20), en remarquant que l'équation adjointe s'écrit également

$$\nabla_y \ell(y_0, u_0, p) = 0$$

et que le gradient est aussi donné par

$$\nabla\psi(u_0) = \nabla_u \ell(y_0, u_0, p_0).$$

Cependant, dans certains cas, il est préférable de reprendre la démarche suivie pour arriver à ces formules plutôt que de les appliquer directement. En particulier, si les équations d'état sont des équations aux dérivées partielles ou des équations d'évolution, les conditions aux limites ou initiales se trouvent plus facilement en reprenant la démarche que nous avons exposée, plutôt qu'en appliquant les formules de dérivation du lagrangien (voir la section 5.4.3).

Une interprétation de l'état adjoint

Comme un multiplicateur de Lagrange, l'état adjoint p a une interprétation marginaliste, que nous énonçons dans la proposition 5.30 ci-dessous. Pour cela, considérons le système perturbé par le vecteur λ :

$$F(y, u) + \lambda = 0.$$

Si $F'_y(y_0, u_0)$ est inversible, on peut encore exprimer la solution de ce système perturbé dans un voisinage de $(u, \lambda) = (u_0, 0)$ par une fonction implicite

$$(u, \lambda) \mapsto \tilde{y}(u, \lambda).$$

Comme $F(\tilde{y}(u, 0), u) = 0$ pour tout u voisin de u_0, l'unicité de la fonction implicite implique que $\tilde{y}(u, 0) = y(u)$ est la valeur en u de la fonction implicite du problème non perturbé. On peut aussi considérer la fonction

$$\tilde{\psi}(u, \lambda) = \varphi(\tilde{y}(u, \lambda), u).$$

Proposition 5.30 *L'état adjoint p associé à (y, u), solution du système non perturbé, donne la variation de la valeur de $\tilde{\psi}$ en $(u, 0)$ par rapport à une perturbation des équations d'état :*

$$p = \nabla_\lambda \tilde{\psi}(u, 0).$$

Démonstration. Pour montrer cela, on calcule $\nabla_\lambda \tilde{\psi}(u, 0)$ par la méthode de l'état adjoint. On introduit le lagrangien du problème perturbé

$$\tilde{\ell}(y,u,\lambda,p) = \varphi(y,u) + \langle p, F(y,u) + \lambda \rangle.$$

Soit u donné. L'état du système perturbé correspondant à ce u et à $\lambda = 0$ est l'état y du système non perturbé correspondant à la commande u. L'état adjoint du système perturbé en $(y,u,0)$ s'obtient par

$$\nabla_y \tilde{\ell}(y,u,0,p) \equiv \nabla_y \varphi(y,u) + F'_y(y,u)^* p = 0.$$

Il est donc identique à l'état adjoint correspondant à (y,u) dans le système non perturbé. Enfin, le gradient de $\tilde{\psi}$ par rapport à λ en $(u,0)$ s'obtient par

$$\nabla_\lambda \tilde{\psi}(u,0) = \nabla_\lambda \tilde{\ell}(y,u,0,p) = p.$$

D'où le résultat. □

5.4.3 Exemples

Nous donnons ci-dessous deux exemples illustrant l'utilisation de la méthode de l'état adjoint. Ils sont paradigmatiques. Dans le premier exemple, l'équation d'état est une équation différentielle ordinaire. On est donc dans une situation plus générale que dans le cadre abstrait ci-dessus, puisque les objets manipulés appartiennent à un espace vectoriel de dimension infinie, mais les principes à appliquer sont identiques. Dans le second exemple, l'équation d'état est une équation différentielle ordinaire discrétisée. On pourrait alors appliquer les formules de la section 5.4.2 précédente pour obtenir le gradient, mais, comme nous l'avons déjà dit, il est plus efficace d'appliquer le principe ayant conduit à ces formules, de manière à en dégager la structure. Dans certains cas, il faut un peu de doigté pour aboutir à une formulation opérationnelle. La démarche exposée peut aussi s'appliquer lorsque les équations d'état sont des équations aux dérivées partielles. Ce sujet sort du cadre de cette monographie. Il est traité en détail dans [398].

Cas où l'équation d'état est une équation différentielle

On suppose que l'état du système est une fonction dépendant du temps, définie sur un intervalle $[0,T]$ de $\mathbb{R}$ $(T > 0)$, à valeurs dans $\mathbb{R}^n$:

$$y : t \in [0,T] \to y(t) \in \mathbb{R}^n.$$

Cet état est supposé déterminé par une équation différentielle ordinaire et sa condition initiale :

$$\begin{cases} \dot{y}(t) = \varphi(y(t),t), & \text{pour tout } t \in]0,T[\\ y(0) = Au, \end{cases} \tag{5.21}$$

appelée *équation d'état* du système. Dans (5.21), $\dot{y}$ désigne la dérivée de y par rapport à t, $\varphi : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ est une application régulière, A est une matrice $n \times m$ et $u \in \mathbb{R}^m$ est la variable de commande. Dans certains problèmes de commande optimale, on cherche à déterminer la commande u, qui agit ici sur la condition initiale du système, de manière à minimiser un critère de la forme

$$f(u) = \int_0^T J(y(t,u),t)\,\mathrm{d}t + J^u(u) + J^T(y(T,u)),$$

où J est une fonction de $\mathbb{R}^n \times \mathbb{R}$ dans $\mathbb{R}$ représentant un coût distribué dans le temps, J^u est une fonction de $\mathbb{R}^m$ dans $\mathbb{R}$ représentant un coût sur la commande du système et J^T est une fonction de $\mathbb{R}^n$ dans $\mathbb{R}$ représentant un coût sur l'état final du système. Nous avons écrit $y(t) \equiv y(t,u)$ et $y(T) \equiv y(T,u)$ pour expliciter la dépendance de y en u, qui provient de l'équation d'état (5.21).

On s'intéresse ici au calcul de $\nabla f(u)$, le gradient de f en u. Il se déduira aisément de $f'(u)\cdot v$, la dérivée directionnelle de f en u dans la direction $v \in \mathbb{R}^m$. La difficulté est de prendre en compte la dépendance de y en u. Le calcul proposé est formel, car il ne précise pas les espaces fonctionnels, dont le choix sort du cadre de cette monographie.

Comme dans le cadre abstrait ci-dessus, on écrit $f(u)$ comme un lagrangien au moyen d'un état adjoint, qui est ici une fonction $p : t \in [0,T] \to p(t) \in \mathbb{R}^n$:

$$\begin{aligned} f(u) &= \int_0^T J(y(t,u),t)\,\mathrm{d}t + J^u(u) + J^T(y(T,u)) \\ &\quad + \int_0^T p(t)^{\mathsf{T}} \left(\dot{y}(t,u) - \varphi(y(t,u),t)\right)\,\mathrm{d}t. \end{aligned}$$

Cette identité a lieu quel que soit p, pourvu que y vérifie l'équation d'état (5.21). On choisira p ultérieurement, de manière à éliminer la dérivée de la fonction implicite $u \mapsto y(\cdot,u)$. On a

$$\begin{aligned} f'(u)\cdot v &= \int_0^T J'_y(y,t)\cdot(y'(t,u)\cdot v)\,\mathrm{d}t \\ &\quad + (J^u)'(u)\cdot v + (J^T)'(y(T,u))\cdot(y'(T,u)\cdot v) \\ &\quad + \int_0^T p(t)^{\mathsf{T}} \left(\dot{y}'(t,u)\cdot v - \varphi'_y(y,t)\cdot(y'(t,u)\cdot v)\right)\,\mathrm{d}t. \end{aligned}$$

Pour faire disparaître la dérivée en temps de $y'(t,u)$, on intègre par parties :

$$\begin{aligned} &\int_0^T p(t)^{\mathsf{T}} \left(\dot{y}'(t,u)\cdot v\right)\,\mathrm{d}t \\ &= -\int_0^T \dot{p}(t)^{\mathsf{T}} \left(y'(t,u)\cdot v\right)\,\mathrm{d}t + p(t)^{\mathsf{T}} \left(y'(t,u)\cdot v\right)\Big|_0^T \\ &= -\int_0^T \dot{p}(t)^{\mathsf{T}} \left(y'(t,u)\cdot v\right)\,\mathrm{d}t + p(T)^{\mathsf{T}} \left(y'(T,u)\cdot v\right) - p(0)^{\mathsf{T}} Av, \end{aligned}$$

où on a utilisé le fait que $y(0,u) = Au$ pour tout $u \in \mathbb{R}^n$ et donc $y'(0,u)\cdot v = Av$. On aurait pu faire cette intégration par parties dans l'expression de $f(u)$, avant dérivation. En regroupant les termes contenant $y'(t,u)\cdot v$ et $y'(T,u)\cdot v$ dans $f'(u)\cdot v$, on obtient

$$\begin{aligned} f'(u)\cdot v &= (J^u)'(u)\cdot v \\ &\quad + \int_0^T \left(-\dot{p}(t) - \varphi'_y(y,t)^{\mathsf{T}} p(t) + \nabla_y J(y,t)\right)^{\mathsf{T}} \left(y'(t,u)\cdot v\right)\,\mathrm{d}t \\ &\quad + \Big(p(T) + \nabla J^T(y(T))\Big)^{\mathsf{T}} \Big(y'(T,u)\cdot v\Big) - p(0)^{\mathsf{T}} Av. \end{aligned}$$

On a désigné de la même façon la dérivée $\varphi'_y(y,t)$ et la matrice jacobienne associée. L'étape importante de ce calcul est maintenant d'éliminer $y'(t,u)\cdot v$ et $y'(T,u)\cdot v$, qui sont coûteux à calculer, en annulant leur facteur par un choix adéquat de p. Ceci conduit à l'*équation adjointe*, qui est une équation différentielle ordinaire linéaire en p, avec une condition *finale*

$$\left\{\begin{array}{l} -\dot p(t) = \varphi'_y(y(t),t)^{\mathsf T} p(t) - \nabla_y J(y(t),t), \quad \text{pour tout } t\in\,]0,T[\\ p(T) = -\nabla J^T(y(T)). \end{array}\right. \tag{5.22}$$

On obtient alors

$$f'(u)\cdot v = (J^u)'(u)\cdot v - p(0)^{\mathsf T} A v$$

et donc

$$\nabla f(u) = \nabla J^u(u) - A^{\mathsf T} p(0). \tag{5.23}$$

Pour résumer, le gradient $\nabla f(u)$ de f en u s'obtient en déterminant d'abord y comme solution de l'équation d'état (5.21), qui est souvent non linéaire, ensuite p comme solution de l'équation adjointe (5.22), toujours linéaire, et enfin le gradient par (5.23). Ce calcul permet d'éviter la détermination des m trajectoires $t\mapsto y'(t,u)\cdot e_i$, pour $i=1,\dots,m$, formant la dérivée de la fonction implicite $u\mapsto y(\cdot,u)$.

On observera que, si l'équation d'état est progressive (c.-à-d., elle s'intègre à partir d'une condition initiale), l'équation adjointe est rétrograde (c.-à-d., elle s'intègre à partir d'une condition finale). Notons également que, pour intégrer l'équation adjointe, il faut connaître l'état y à chaque instant. En pratique (sur les modèles discrétisés), il faut donc mémoriser cet état, ce qui peut demander beaucoup d'espace-mémoire : l'efficacité du calcul est donc contrebalancée par un besoin de mémoire. On retrouvera cette propriété dans le mode inverse de différentiation (section 5.5.3).

Cas où l'équation d'état est une équation différentielle discrétisée

Supposons à présent que l'état $y : t\in[0,T]\to y(t)\in\mathbb{R}^n$ du système considéré soit décrit par l'équation différentielle

$$\left\{\begin{array}{l} \dot y(t) = \varphi(y(t),u(t),t), \quad \text{pour tout } t\in\,]0,T[\\ y(0) = y_0, \end{array}\right. \tag{5.24}$$

dans laquelle $y_0\in\mathbb{R}^n$ est la valeur donnée de l'état à l'instant initial et la commande est à présent une fonction $u : t\in[0,T]\to u(t)\in\mathbb{R}^m$ agissant à chaque instant sur le système par l'intermédiaire de $\varphi : \mathbb{R}^n\times\mathbb{R}^m\times\mathbb{R}\to\mathbb{R}^n$. Le critère que l'on cherche à minimiser par rapport à u est le suivant :

$$\int_0^T J(y(t,u),u(t),t)\,\mathrm{d}t + J^T(y(T,u)), \tag{5.25}$$

où $J : \mathbb{R}^n\times\mathbb{R}^m\times\mathbb{R}\to\mathbb{R}$ représente le coût instantané et $J^T : \mathbb{R}^n\to\mathbb{R}$ représente le coût associé à l'état final du système. Nous avons encore écrit $y(t)\equiv y(t,u)$ pour insister sur le fait que y dépend de u par l'intermédiaire de l'équation d'état (5.24).

On s'intéresse cette fois à la forme discrétisée de ce modèle. Pour cela, on décompose l'intervalle d'intégration $[0,T]$ en N pas de temps $\Delta t_i > 0$, $1\leqslant i\leqslant N$, tels que

$T = \sum_{i=1}^{N} \Delta t_i$. On note $t_0 = 0$, puis $t_i = t_{i-1} + \Delta t_i$, pour $1 \leqslant i \leqslant N$, (donc $t_N = T$) et $u_i = u(t_i) \in \mathbb{R}^m$. Pour simplifier, on supposera que l'approximation $y_i \in \mathbb{R}^n$ de $y(t_i)$ est obtenue par le *schéma d'Euler implicite* (pour d'autres schémas d'intégration, on pourra consulter [230 ; 1997] par exemple) :

$$y_{i+1} = y_i + \Delta t_i \, \varphi(y_{i+1}, u_{i+1}, t_i), \quad \text{pour } 0 \leqslant i \leqslant N-1, \tag{5.26}$$

démarrant sur la condition initiale y_0. On prendra comme approximation du critère continu, le critère discret suivant

$$f(\mathbf{u}) = \sum_{i=1}^{N} J(y_i(\mathbf{u}), u_i, t_i) \, \Delta t_i + J^T(y_N(\mathbf{u})),$$

où $\mathbf{u} = \{u_i\}_{i=1}^N \subseteq \mathbb{R}^m$ et où la notation $y_i \equiv y_i(\mathbf{u})$ met en évidence la dépendance de y_i par rapport à $\mathbf{u}$. On cherche à calculer le gradient de f.

Comme dans le cadre abstrait et le cas continu ci-dessus, on écrit $f(\mathbf{u})$ comme un lagrangien au moyen d'un état adjoint, qui est ici un ensemble de $N+1$ vecteurs $\mathbf{p} = \{p_i\}_{i=0}^N \subseteq \mathbb{R}^n$ (le vecteur $p_N \in \mathbb{R}^n$ apparaîtra plus loin) :

$$\begin{aligned} f(\mathbf{u}) \;=\; & \sum_{i=1}^{N} J(y_i(\mathbf{u}), u_i, t_i) \, \Delta t_i + J^T(y_N(\mathbf{u})) \\ & + \sum_{i=0}^{N-1} p_i^{\mathsf{T}} \Big(y_{i+1} - y_i - \Delta t_i \, \varphi(y_{i+1}, u_{i+1}, t_i) \Big). \end{aligned}$$

Cette identité a lieu quel que soit $\mathbf{p}$, pourvu que $\mathbf{y} = \{y_i\}_{i=0}^N$ vérifie l'équation d'état (5.26). On choisira $\mathbf{p}$ ultérieurement, de manière à éliminer la dérivée de la fonction implicite $\mathbf{u} \mapsto \mathbf{y}(\mathbf{u})$. Pour varier, commençons par faire une « somme par parties » (équivalent de l'intégration par parties de l'exemple précédent), avant dérivation. On a

$$\sum_{i=0}^{N-1} p_i^{\mathsf{T}}(y_{i+1} - y_i) = \sum_{i=1}^{N} p_{i-1}^{\mathsf{T}} y_i - \sum_{i=0}^{N-1} p_i^{\mathsf{T}} y_i = \sum_{i=1}^{N} (p_{i-1} - p_i)^{\mathsf{T}} y_i + p_N^{\mathsf{T}} y_N - p_0^{\mathsf{T}} y_0.$$

Il en résulte

$$\begin{aligned} f(\mathbf{u}) \;=\; & \sum_{i=1}^{N} J(y_i(\mathbf{u}), u_i, t_i) \, \Delta t_i + J^T(y_N(\mathbf{u})) \\ & + \sum_{i=1}^{N} \Big((p_{i-1} - p_i)^{\mathsf{T}} y_i - \Delta t_{i-1} \, p_{i-1}^{\mathsf{T}} \varphi(y_i, u_i, t_{i-1}) \Big) \\ & + p_N^{\mathsf{T}} y_N - p_0^{\mathsf{T}} y_0. \end{aligned}$$

On peut à présent calculer la dérivée directionnelle de f en $\mathbf{u}$ dans une direction $\mathbf{v} = \{v_i\}_{i=1}^N \subseteq \mathbb{R}^m$:

$$
\begin{aligned}
f'(\mathbf{u})\cdot\mathbf{v} \;=\; & \sum_{i=1}^{N}\Big(J'_y(y_i,u_i,t_i)\cdot(y'_i(\mathbf{u})\cdot\mathbf{v})+J'_u(y_i,u_i,t_i)\cdot v_i\Big)\Delta t_i\\
& +(J^T)'(y_N)\cdot(y'_N(\mathbf{u})\cdot\mathbf{v})\\
& +\sum_{i=1}^{N}\Big((p_{i-1}-p_i)^{\mathsf{T}}(y'_i(\mathbf{u})\cdot\mathbf{v})\\
& \qquad -\Delta t_{i-1}\,p_{i-1}^{\mathsf{T}}\varphi'_y(y_i,u_i,t_{i-1})\cdot(y'_i(\mathbf{u})\cdot\mathbf{v})\\
& \qquad -\Delta t_{i-1}\,p_{i-1}^{\mathsf{T}}\varphi'_u(y_i,u_i,t_{i-1})\cdot v_i\Big)\\
& +p_N^{\mathsf{T}}(y'_N(\mathbf{u})\cdot\mathbf{v}).
\end{aligned}
$$

Comme précédemment, l'étape importante du calcul consiste à éliminer les dérivées $y'_i(\mathbf{u})\cdot\mathbf{v}$ $(1\leqslant i\leqslant N)$, qui sont coûteuses à évaluer, en annulant ses facteurs par un choix approprié de $\mathbf{p}$. Ceci conduit à l'*équation adjointe* suivante

$$
\left\{\begin{array}{l}
\Big(I-\Delta t_{i-1}\,\varphi'_y(y_i,u_i,t_{i-1})^{\mathsf{T}}\Big)p_{i-1}=p_i-\Delta t_i\,\nabla_y J(y_i,u_i,t_i),\quad \text{pour } i=N,\dots,1\\
p_N=-\nabla J^T(y_N).
\end{array}\right. \tag{5.27}
$$

On obtient alors

$$
\nabla f(\mathbf{u})=\Big\{\Delta t_i\,\nabla_u J(y_i,u_i,t_i)-\Delta t_{i-1}\,\varphi'_u(y_i,u_i,t_{i-1})^{\mathsf{T}}p_{i-1}\Big\}_{i=1}^{N}, \tag{5.28}
$$

qui est un vecteur de $\mathbb{R}^{Nm}$.

Résumons les opérations. Le gradient $\nabla f(\mathbf{u})$ de f en $\mathbf{u}=\{u_i\}_{i=1}^N\in\mathbb{R}^{Nm}$ s'obtient en déterminant d'abord $\mathbf{y}=\{y_i\}_{i=1}^N\in\mathbb{R}^{Nn}$ à partir de sa condition initiale y_0 comme solution de l'équation d'état (5.26), qui est souvent non linéaire. Ensuite $\mathbf{p}=\{p_i\}_{i=0}^{N-1}\in\mathbb{R}^{Nn}$ est déterminé à partir de sa condition finale $p_N=-\nabla J^T(y_N)$ comme solution de l'équation adjointe (5.27), toujours linéaire. Enfin le gradient est obtenu par (5.28). Comme dans le cas continu analysé précédemment, l'équation d'état est progressive et l'équation adjointe est rétrograde.

5.4.4 Calcul de produits hessienne-vecteur

On se place à nouveau dans le cadre de la section 5.4.1, où des variables y et u sont reliées par une *équation d'état*

$$F(y,u)=0,$$

permettant d'écrire localement y comme une fonction de u et où l'on cherche à dériver la fonction

$$u\mapsto\psi(u):=\varphi(y(u),u).$$

On s'intéresse à présent au produit de la hessienne de ψ en u (pour un produit scalaire $\langle\cdot,\cdot\rangle$ donné) par un vecteur v.

Le schéma de calcul 5.29 peut être vu comme évaluant successivement, à partir de u donné, les variables y, puis p, et enfin $\nabla\psi$, toutes vues comme des fonctions de u.

Il suffit donc de calculer la dérivée directionnelle de ces quantités dans la direction v, pour obtenir finalement $\nabla^2\psi(u)v$. On note $\dot{y} = y'(u)$ et $\dot{p} = p'(u)$. La dérivation de l'équation d'état donne une équation linéaire permettant de calculer $\dot{y}$:

$$F'_y(y,u)\,\dot{y} = -F'_u(y,u).$$

La dérivation de l'équation adjointe donne une équation linéaire permettant de calculer $\dot{p}$:

$$F'_y(y,u)^*\,\dot{p} = -\left(F''_{yy}(y,u)\cdot\dot{y} + F''_{yu}(y,u)\cdot v\right)^*\,p - \left(\varphi''_{yy}(y,u)\cdot\dot{y} + \varphi''_{yu}(y,u)\cdot v\right).$$

Il reste à dériver l'équation donnant le gradient de ψ pour obtenir le produit hessienne-vecteur :

$$\psi''(u)\cdot v = \left(F''_{yu}(y,u)\cdot\dot{y} + F''_{uu}(y,u)\cdot v\right)^*\,p + \left(\varphi''_{yu}(y,u)\cdot\dot{y} + \varphi''_{uu}(y,u)\cdot v\right).$$

On observera que les systèmes linéaires intervenant dans le calcul de $\nabla\psi(u)$ ou $\nabla^2\psi(u)v$, utilisent tous la matrice $F'_y(y,u)$ ou son adjointe. Si une factorisation de cette matrice s'impose, celle-ci ne devra être faite qu'une seule fois par valeur de u.

5.5 Différentiation automatique

En optimisation, l'objet numérique de base est le gradient. Les méthodes numériques qui peuvent fonctionner sans calcul de gradients sont lentes. On peut alors se demander si les n quantités formant le gradient d'une fonction scalaire définie sur $\mathbb{R}^n$ peuvent se calculer en un temps raisonnable, en particulier si n est très grand. On pourrait en effet penser que cela prend n fois plus de temps que le calcul de la fonction elle-même. S'il en était ainsi, les techniques numériques utilisant le gradient devraient être cantonnées à l'optimisation de problèmes de taille petite ou moyenne. Nous montrerons dans cette section (proposition 5.32) que, par le mode inverse de différentiation automatique, on peut toujours calculer le gradient d'une fonction en un temps qui est du même ordre que celui nécessaire au calcul de la fonction elle-même (sans facteur proportionnel à n). Il s'agit donc d'un résultat fondamental pour l'optimisation numérique.

La *différentiation automatique*[1] est un ensemble de techniques permettant d'obtenir les dérivées exactes (aux erreurs d'arrondi près) d'une fonction représentée par un programme informatique, écrit par exemple en Fortran ou en C. Si c'est un programme qui représente la fonction à dériver, c'est aussi un programme qui est généré par le différentiateur. Il faudra l'exécuter pour obtenir la valeur de la dérivée de la fonction en un point donné.

La méthode se distingue donc de la différentiation symbolique, qui suppose que la fonction à dériver admet une représentation symbolique au moyen de formules mathématiques et qui génère une représentation, également symbolique, de la fonction dérivée. On la trouve dans des codes comme Macsyma [607 ; 1985] ou Maple [116 ;

[1] En anglais, le vocable *Automatic Differentiation* tend à être remplacé par *Computational Differentiation* ou *Algorithmic Differentiation* [288, 291]. En français, on rencontre parfois l'expression *Différentiation par Programme*.

1988]. Elle se distingue aussi de la différentiation par différences finies, méthode par laquelle les dérivées partielles d'une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ sont *approchées* en faisant varier insensiblement et successivement chacune des variables, tout en gardant les autres fixées :

$$\frac{\partial f}{\partial x_i}(x) \simeq \frac{f(x + t_i e_i) - f(x)}{t_i}, \quad i = 1, \ldots, n,$$

où e_i est le i-ième vecteur de la base canonique de $\mathbb{R}^n$ et $t_i > 0$ est « bien choisi » (ni trop grand ni trop petit !). Avec cette dernière technique les dérivées ne peuvent pas être calculées avec précision. De plus, par cette technique le calcul d'un gradient demande l'évaluation de f aux n points $x + t_i e_i$, $1 \leqslant i \leqslant n$, ce qui peut demander beaucoup de temps de calcul.

Un des attraits de la différentiation automatique est de pouvoir générer un programme calculant le gradient d'une fonction à valeurs scalaires, dont le temps d'exécution relatif à celui du programme original est indépendant du nombre de variables par rapport auxquelles on dérive. Il s'agit en fait de génération automatique de codes adjoints. Cette technique s'est concrétisée dans des codes de différentiation tels que ADOL-C [], ADIFOR [] ou TAPENADE [].

Dans cette section, nous présentons quelques aspects théoriques de la différentiation automatique. Pour plus de détails, on consultera les ouvrages et monographies cités en fin de chapitre.

5.5.1 Modèle de programme

Pour obtenir les formules permettant de générer de manière automatique un code calculant les dérivées d'une fonction représentée par un programme, on commence par considérer le cas d'un programme formé d'une suite d'instructions d'affectation. Un tel programme a l'avantage d'avoir une représentation mathématique simple avec laquelle on peut travailler. Les règles que l'on déduit de ce modèle permettent alors de comprendre les règles de transformation de codes à mettre en œuvre sur des programmes plus complexes, voire généraux, tels que ceux écrits en Fortran ou C.

On note $v_1, v_2, \ldots, v_N$ les N variables du code informatique (ou cases-mémoire) sur lesquelles travaille le programme modèle. Il n'y a aucun ordre sur ces variables. En particulier, il n'est pas nécessaire qu'elles soient évaluées dans l'ordre de leur indice. C'est simplement une manière commode de les désigner toutes. On supposera que les n *variables d'entrée* ($n \leqslant N$), celles par rapport auxquelles on dérive, encore appelée *variables indépendantes*, sont les variables $v_1, \ldots, v_n$. Les m *variables de sortie* ($m \leqslant N$), celles que l'on veut dériver, sont supposées être les variables $v_{N-m+1}, \ldots, v_N$. Des variables peuvent être à la fois d'entrée et de sortie. On dira qu'une variable v_i devient *active* lorsqu'on lui affecte une valeur qui dépend de la valeur donnée aux variables indépendantes.

Le programme modèle que nous considérerons est donc supposé être formé d'une suite de K instructions d'affectation exécutées l'une après l'autre, ce que l'on peut écrire :

$$v_{\mu_k} := \varphi_k(v_{D_k}), \quad k = 1, \ldots, K. \tag{5.29}$$

À chaque instruction k, le programme modifie la variable v_{μ_k} au moyen d'une fonction φ_k, en utilisant les variables v_i, $i \in D_k$, où D_k est une partie de $[1 : N]$. Dans ce modèle, on ne se donne aucune restriction sur $\mu_k \in [1 : N]$, qui peut en particulier faire partie

de D_k. Si on note x les variables d'entrée et f les variables de sorties, on cherche donc à différentier une fonction

$$f : x \in \mathbb{R}^n \mapsto f(x) \in \mathbb{R}^m,$$

qui est représentée par le programme modèle (5.29) et dont la valeur en un point x peut être obtenue en exécutant ce programme.

Pour fixer les idées, considérons l'exemple suivant écrit en Fortran, dans lequel on calcule une variable $f = \texttt{v5}$ à partir de la donnée d'un couple de variables $x = (\texttt{v1}, \texttt{v2})$:

```
v3 = v1 + v2**2
v4 = v1**2 * sin(v3)
v4 = v4/v3
v5 = exp(v4)
```

La correspondance entre ce programme et le modèle (5.29) est détaillée au tableau 5.1.

indice k de l'instruction	instruction	indice de la variable modifiée	fonction φ_k	indices de dépendance
1	`v3=v1+v2**2`	$\mu_1 = 3$	$\varphi_1(a,b) = a + b^2$	$D_1 = \{1, 2\}$
2	`v4=v1**2*sin(v3)`	$\mu_2 = 4$	$\varphi_2(a,b) = a^2 \sin(b)$	$D_2 = \{1, 3\}$
3	`v4=v4/v3`	$\mu_3 = 4$	$\varphi_3(a,b) = b/a$	$D_3 = \{3, 4\}$
4	`v5=exp(v4)`	$\mu_4 = 5$	$\varphi_4(a) = \exp(a)$	$D_4 = \{4\}$

Tableau 5.1. Correspondance entre l'exemple de programme et le modèle (5.29)

On désigne par $\mathcal{F}$ l'ensemble des *fonctions intermédiaires*, c'est-à-dire les fonctions φ_k utilisées dans le programme modèle (5.29). Il n'y a pas de restriction sur ces fonctions, si ce n'est qu'elles doivent être différentiables et à valeurs scalaires (le cas où elles sont à valeurs vectorielles se traite de manière analogue, voir l'exercice 5.7). Elles peuvent représenter les opérations élémentaires (`+`, `-`, `*`, `/`), les fonctions intrinsèques du langage informatique utilisé, des compositions de ces fonctions ou encore des ensembles d'instructions (sous-routines, procédures, fonctions). On notera $\mathcal{F}_{_F} = \{$`+`, `-`, `*`, `/`, `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`$\}$ la classe des fonctions utilisables en Fortran (langage choisi pour estimer la complexité des calculs ci-dessous).

5.5.2 Différentiation en mode direct

Le *mode direct* de différentiation est le plus simple et le plus intuitif. Il est bien adapté au calcul des dérivées directionnelles de la fonction f représentée par le programme.

Soit $d \in \mathbb{R}^n$ la direction dans laquelle on veut différentier f. On cherche donc à calculer $f'(x) \cdot d$. Pour cela, il est bon de voir chaque variable du code comme une fonction des variables d'entrée $x = (v_1, \ldots, v_n)$. Donc à chaque v_i $(1 \leqslant i \leqslant N)$

correspond une dérivée directionnelle $\dot{v}_i := v_i'(x)\cdot d$. Alors, en différentiant l'instruction k du programme (5.29), on obtient

$$\dot{v}_{\mu_k} = \sum_{i\in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k})\dot{v}_i.$$

On a donc une formule permettant de calculer $\dot{v}_{\mu_k} := v'_{\mu_k}(x)\cdot d$ à partir des $\dot{v}_i$, $i \in D_k$. Grâce à celle-ci, on peut « propager » le calcul des dérivées directionnelles des variables évaluées dans le code, parallèlement à leur évaluation. C'est l'idée utilisée dans le mode direct de différentiation.

Remarquons que $\dot{v}_i = d_i$, pour $1 \leqslant i \leqslant n$. Pour calculer $f'(x)\cdot d$, il suffit donc d'initialiser les dérivées directionnelles des variables indépendantes comme suit :

$$\dot{v}_i := d_i, \quad \text{pour } 1 \leqslant i \leqslant n,$$

et ensuite de propager les dérivées directionnelles dans le code par les instructions suivantes :

$$\left.\begin{array}{r} \dot{v}_{\mu_k} := \sum_{i\in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k})\dot{v}_i \\ v_{\mu_k} := \varphi_k(v_{D_k}) \end{array}\right\}, \quad k = 1,\ldots,K.$$

On calcule $\dot{v}_{\mu_k}$ avant v_{μ_k} au cas où φ_k dépendrait de v_{μ_k}. Il faut en effet évaluer les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}$ avec la valeur non modifiée de v_{μ_k}. En fin d'exécution, on récupère dans $\dot{v}_{N-m+1}, \ldots, \dot{v}_N$, la dérivée $f'(x)\cdot d$:

$$f'(x)\cdot d := (\dot{v}_{N-m+1},\ldots,\dot{v}_N).$$

Ce mode de différentiation porte le nom de *mode direct* et le code réalisant ces opérations est appelé *code linéaire tangent* de (5.29). Celui-ci peut être résumé ainsi :

$$\boxed{\begin{array}{l} \text{pour } i = 1,\ldots,n \\ \quad \dot{v}_i := d_i \ ; \\ \text{pour } k = 1,\ldots,K \\ \quad \dot{v}_{\mu_k} := \sum_{i\in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k})\dot{v}_i \ ; \\ \quad v_{\mu_k} := \varphi_k(v_{D_k}) \ ; \\ f'(x)\cdot d := (\dot{v}_{N-m+1},\ldots,\dot{v}_N) \ . \end{array}} \tag{5.30}$$

On voit que l'on a associé à chaque variable v_i du code, une variable $\dot{v}_i$ contenant sa dérivée directionnelle. Si toutes les variables sont actives, l'espace-mémoire requis est donc exactement le double de celui utilisé par le programme original. En ce qui concerne le temps de calcul $T(f, f'\cdot d)$ nécessaire à l'exécution de (5.30), on peut le comparer au temps $T(f)$ de l'exécution de (5.29).

Proposition 5.31 *Le temps $T(f, f'\cdot d)$ nécessaire au calcul de f et de sa dérivée directionnelle $f'\cdot d$ dans la direction d au moyen de l'algorithme (5.30) vérifie*

l'estimation suivante:

$$\frac{T(f, f'\cdot d)}{T(f)} \leqslant C_{\mathcal{F}},$$

où $T(f)$ est le temps nécessaire au calcul de f par l'algorithme (5.29) et $C_{\mathcal{F}}$ est une constante ne dépendant que des fonctions intermédiaires φ_k de la bibliothèque $\mathcal{F}$.

Démonstration. D'après l'algorithme (5.30), on a

$$T(f, f'\cdot d) = \sum_{k=1}^{K} T(\varphi_k, \varphi'_k).$$

Ensuite, en utilisant l'inégalité de Cauchy-Schwarz généralisée (A.9), on obtient

$$\begin{aligned} T(f, f'\cdot d) &\leqslant \max_{1\leqslant k\leqslant K} \left(\frac{T(\varphi_k, \varphi'_k)}{T(\varphi_k)}\right) \sum_{k=1}^{K} T(\varphi_k) \\ &\leqslant \max_{\varphi\in\mathcal{F}} \left(\frac{T(\varphi, \varphi')}{T(\varphi)}\right) T(f) \\ &= C_{\mathcal{F}}\, T(f), \end{aligned}$$

où la constante

$$C_{\mathcal{F}} = \max_{\varphi\in\mathcal{F}} \left(\frac{T(\varphi, \varphi')}{T(\varphi)}\right)$$

ne dépend que de la classe $\mathcal{F}$ des fonctions intermédiaires. □

Par exemple si toutes les fonctions intermédiaires φ_k sont prises dans $\mathcal{F}_{_F}$ (famille des fonctions Fortran), on montre, sous des hypothèses raisonnables sur le temps respectif de l'évaluation des fonctions de $\mathcal{F}_{_F}$, que

$$\frac{T(f, f'\cdot d)}{T(f)} \leqslant 4,$$

et ceci quel que soit le nombre m de variables dérivées (voir [243 ; 1991]).

À titre d'illustration, voyons comment s'écrit le code linéaire tangent dans le cas de l'exemple de la page 278. Supposons que les deux composantes de la direction d soient données dans les variables `d1` et `d2`. En utilisant le suffixe `tl` pour désigner les variables $\dot{v}_i$ associées aux variables v_i du code et contenant leur dérivée directionnelle, le code linéaire tangent s'écrit :

```
v1tl = d1
v2tl = d2
v3tl = v1tl + 2*v2*v2tl
v3 = v1 + v2**2
v4tl = 2*v1*v1tl*sin(v3) + v1**2*cos(v3)*v3tl
v4 = v1**2 * sin(v3)
```

```
aux = v4/v3
v4tl = (v4tl - aux*v3tl)/v3
v4 = aux
aux = exp(v4)
v5tl = aux * v4tl
v5 = aux
```

On a utilisé une variable auxiliaire `aux`, de manière à éviter de calculer plusieurs fois certaines quantités.

Ce qu'il faut retenir sur le mode direct :

1. Le mode direct de différentiation est le bon mode pour calculer la dérivée d'un grand nombre de variables (ou variables de sortie) par rapport à un petit nombre de variables indépendantes (ou variables d'entrée).
2. C'est donc aussi le bon mode pour calculer une dérivée directionnelle d'une application $f : \mathbb{R}^n \to \mathbb{R}^m$.
3. À chaque variable v_i du code, on associe une variable $\dot{v}_i$ qui contient la dérivée directionnelle de v_i dans une direction d donnée. Sa valeur est nulle (et alors $\dot{v}_i$ peut ne pas exister dans le code linéaire tangent) si v_i n'est pas influencée par les variables par rapport auxquelles on dérive (c'est-à-dire si elle n'est pas active).
4. Le code linéaire tangent peut s'obtenir en « dérivant » le code original ligne par ligne. Si dans ce dernier une instruction s'interprète comme une application $\{v_i : i \in D_k\} \mapsto v_{\mu_k}$, les instructions à ajouter pour obtenir le code linéaire tangent forment une application entre variables $\dot{v}_i$ de la forme $\{\dot{v}_i : i \in D_k\} \mapsto \dot{v}_{\mu_k}$.

5.5.3 Différentiation en mode inverse

Le *mode inverse* de différentiation est moins intuitif et plus complexe à mettre en œuvre que le mode direct. Il trouve son utilité lorsqu'on veut calculer des dérivées par rapport à un grand nombre de variables. Ce mode est une méthode de génération automatique de codes adjoints, encore appelés codes linéaires cotangents, concept familier en commande optimale.

Il y a plusieurs manières d'introduire le mode inverse. Dans l'état de l'art [243 ; 1991], quatre approches ont été recensées : l'approche du graphe de calcul [535 ; 1984] [326 ; 1984] [327 ; 1984], la méthode des substitutions rétrogrades [565 ; 1980] [353 ; 1984], la méthode de dualité et l'approche de Speelpenning [565 ; 1980]. Chacune de ces approches apporte un éclairage différent sur ce mode de différentiation, toujours surprenant, mais c'est la dernière qui est la plus simple à exposer et à adapter à diverses situations. C'est aussi celle que nous allons présenter.

D'abord, il est intéressant de voir l'instruction k de (5.29) comme une transformation agissant sur l'ensemble des variables $v_1, \ldots, v_N$ du code et qui laisse inchangées toutes ces variables sauf v_{μ_k}. On peut donc lui associer une transformation $\Phi_k : \mathbb{R}^N \to \mathbb{R}^N$, définie par

$$\left(\Phi_k(v)\right)_i = \begin{cases} v_i & \text{si } i \neq \mu_k \\ \varphi_k(v_{D_k}) & \text{si } i = \mu_k. \end{cases}$$

À présent, on peut voir le programme (5.29) comme une composition de K fonctions :

$$(\Phi_K \circ \cdots \circ \Phi_1)(v).$$

Afin de lever certaines ambiguïtés, on note $v^0 = v$ et

$$v^k = (\Phi_k \circ \cdots \circ \Phi_1)(v)$$

la valeur des variables du code après exécution des k premières instructions.

Soit alors d dans $\mathbb{R}^m$, espace d'arrivée de f. Avec le mode inverse, on cherche à calculer le gradient de l'application $x \mapsto d^{\mathsf{T}} f(x)$ pour le produit scalaire euclidien, c'est-à-dire le vecteur des dérivées partielles

$$\nabla(d^{\mathsf{T}} f)(x) = \left(\frac{\partial(d^{\mathsf{T}} f)}{\partial x_i}(x)\right)_{1 \leqslant i \leqslant n}.$$

Il s'agit donc d'une dérivation cotangente. On a en notant 0_p le vecteur nul de $\mathbb{R}^p$,

$$d^{\mathsf{T}} f(x) = \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}^{\mathsf{T}} (\Phi_K \circ \cdots \circ \Phi_2 \circ \Phi_1)(x, 0_{N-n}),$$

si bien que

$$(d^{\mathsf{T}} f)'(x) \cdot y = \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}^{\mathsf{T}} \Phi_K'(v^{K-1}) \cdots \Phi_2'(v^1) \Phi_1'(v) \begin{pmatrix} I_n \\ 0_{(N-n)\times n} \end{pmatrix} y,$$

où I_n est la matrice unité d'ordre n et $0_{p\times q}$ est la matrice nulle de type $p \times q$. On en déduit

$$\nabla(d^{\mathsf{T}} f)(x) = \begin{pmatrix} I_n & 0_{n\times(N-n)} \end{pmatrix} \Phi_1'(v)^{\mathsf{T}} \Phi_2'(v^1)^{\mathsf{T}} \cdots \Phi_K'(v^{K-1})^{\mathsf{T}} \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}.$$

Dans le produit de matrices du membre de droite, on a un vecteur à droite et une matrice à gauche. Le nombre d'opérations pour l'évaluer sera donc moindre si l'on effectue les multiplications matricielles de droite à gauche. Pour effectuer ces produits, on introduit des *variables duales* $\overline{v} \in \mathbb{R}^N$, que l'on initialise au vecteur à droite de l'expression ci-dessus

$$\overline{v} := \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}.$$

Ensuite, ces variables duales sont mises à jour en effectuant les produits matriciels sus-mentionnés, c'est-à-dire

$$\overline{v} := \Phi_k(v^{k-1})^{\mathsf{T}} \overline{v}, \quad \text{pour } k = K, \ldots, 1.$$

Le dernier produit matriciel nous apprend que le gradient $\nabla(d^{\mathsf{T}} f)(x)$ se trouve dans les n premières composantes du vecteur $\overline{v}$ ainsi obtenu.

Pour écrire cette procédure de façon précise, il reste à observer que la jacobienne de $\Phi_k(v^{k-1})$ est la matrice $N \times N$

$$\Phi_k'(v^{k-1}) = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ \frac{\partial\varphi_k}{\partial v_1}(v^{k-1}) & \cdots & \cdots & \frac{\partial\varphi_k}{\partial v_{\mu_k}}(v^{k-1}) & \cdots & \cdots & \frac{\partial\varphi_k}{\partial v_N}(v^{k-1}) & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & 1 & \end{pmatrix},$$

où seule sa μ_k-ième ligne peut ne pas être formée de vecteurs de base de $\mathbb{R}^N$. Alors la mise à jour $\overline{v} := \Phi_k(v^{k-1})^{\mathsf{T}}\overline{v}$ s'écrit

$$\left\{ \begin{array}{ll} \overline{v}_i := \overline{v}_i & \text{si } i \notin D_k \cup \{\mu_k\} \\ \overline{v}_i := \overline{v}_i + \dfrac{\partial\varphi_k}{\partial v_i}(v^{k-1})\overline{v}_{\mu_k} & \text{si } i \in D_k \setminus \{\mu_k\} \\ \overline{v}_{\mu_k} := \dfrac{\partial\varphi_k}{\partial v_{\mu_k}}(v^{k-1})\overline{v}_{\mu_k}. & \end{array} \right. \tag{5.31}$$

Il faut d'abord mettre à jour les $\overline{v}_i$ pour $i \neq \mu_k$ avant $\overline{v}_{\mu_k}$, car il est clair que c'est l'ancienne valeur de $\overline{v}_{\mu_k}$ qu'il faut utiliser dans le calcul des $\overline{v}_i$ ($i \in D_k \setminus \{\mu_k\}$).

Ce mode de différentiation porte le nom de *mode inverse* et le code réalisant ces opérations est appelé *code linéaire cotangent*. Il calcule $\nabla(d^{\mathsf{T}}f)(x)$ par le programme suivant

$$\boxed{\begin{array}{l} \text{pour } i = 1, \ldots, N-m \\ \quad \overline{v}_i := 0 \ ; \\ \text{pour } i = N-m+1, \ldots, N \\ \quad \overline{v}_i := d_i \ ; \\ \text{pour } k = K, \ldots, 1 \\ \quad \text{pour } i \in D_k \setminus \{\mu_k\} \\ \qquad \overline{v}_i := \overline{v}_i + \frac{\partial\varphi_k}{\partial v_i}(v^{k-1})\overline{v}_{\mu_k} \ ; \\ \quad \overline{v}_{\mu_k} := \frac{\partial\varphi_k}{\partial v_{\mu_k}}(v^{k-1})\overline{v}_{\mu_k} \ ; \\ \nabla(d^{\mathsf{T}}f)(x) := (\overline{v}_1, \ldots, \overline{v}_n) \ . \end{array}} \tag{5.32}$$

On voit que cette procédure s'adapte bien à la situation où on veut dériver une seule variable ($m = 1$ et d est le scalaire 1) par rapport à beaucoup de variables d'entrée (n grand). Dans ce cas, il suffit d'initialiser toutes les variables duales à 0 sauf $\overline{v}_N$ que l'on initialise à 1.

Illustrons cela en appliquant le mode inverse sur l'exemple de la page 278, lorsqu'on veut dériver la variable `v5` par rapport aux variables `v1` et `v2`. On commence par exécuter le code original en sauvegardant certaines quantités. Le choix de l'information à sauvegarder est délicat. Pour coller à la présentation faite ci-dessus, nous avons choisi de sauvegarder les dérivées partielles $\frac{\partial\varphi_k}{\partial v_i}(v^{k-1})$ dans une pile `p` munie du pointeur `ip`, sans essayer d'optimiser le code de manière à garder une certaine lisibilité.

```
ip = 1
p(ip) = 2*v2
```

```
v3 = v1 + v2**2
ip = ip + 1
p(ip) = 2*v1*sin(v3)
ip = ip + 1
p(ip) = v1**2 * cos(v3)
v4 = v1**2 * sin(v3)
ip = ip + 1
p(ip) = 1/v3
ip = ip + 1
p(ip) = -v4/v3**2
v4 = v4/v3
ip = ip + 1
p(ip) = exp(v4)
v5 = exp(v4)
```

Ensuite vient l'exécution du code linéaire cotangent. On a utilisé le suffixe `ad` pour désigner les variables duales $\overline{v}_i$ associées aux variables v_i.

```
v1ad = 0
v2ad = 0
v3ad = 0
v4ad = 0
v5ad = 1
v4ad = v4ad + p(ip)*v5ad
v5ad = 0
ip = ip - 1
v3ad = v3ad + p(ip)*v4ad
ip = ip - 1
v4ad = p(ip)*v4ad
ip = ip - 1
v3ad = v3ad + p(ip)*v4ad
ip = ip - 1
v1ad = v1ad + p(ip)*v4ad
v4ad = 0
ip = ip - 1
v2ad = v2ad + p(ip)*v3ad
v1ad = v1ad + v3ad
v3ad = 0
```

En fin d'exécution, le gradient de `v5` par rapport à `v1` et `v2` se trouve dans (`v1ad`, `v2ad`).

On peut à présent comprendre la difficulté de la mise en œuvre du mode inverse. Il faut d'abord exécuter le code direct (5.29) afin de mémoriser les « quantités » permettant de reconstituer les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ au moment où celles-ci sont utilisées dans le code linéaire cotangent (5.32). Remarquons que cette utilisation se fait en ordre inverse (de $k = K, \ldots, 1$) de l'ordre de mémorisation. Il y a donc un problème de gestion de la mémoire qui ne se posait pas avec le mode direct. Une stratégie extrême consiste à mémoriser, à chaque itération k, toute l'information nécessaire à l'évaluation des dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$, $i \in D_k$. Ceci conduit en général à un besoin en place-mémoire à peu près proportionnel au nombre d'instructions où les variables actives interviennent de façon non linéaire soit, en première approximation,

proportionnel au temps d'exécution du code original (en fait cela dépend beaucoup de ce que l'on mémorise). Cette stratégie ne peut donc fonctionner que pour les petits problèmes. Pour les grands problèmes, il est préférable de mémoriser une partie des variables du code à certains instants de l'exécution de (5.29) (voir [287 ; 1992] pour une technique indépendante de la structure du code).). Dans un problème d'évolution, par exemple, on pourra ne mémoriser que les variables d'état du problème à chaque pas de temps. Ensuite les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ sont recalculées à partir des informations mémorisées. On parvient ainsi à obtenir des codes moins gourmands en place-mémoire, au prix d'une augmentation du temps de calcul.

En ce qui concerne, le temps d'exécution du couple (code original, code linéaire cotangent) par rapport au temps d'exécution du code original, on a le résultat suivant.

Proposition 5.32 *Le temps $T(f, \nabla(d^{\mathsf{T}} f))$ nécessaire au calcul de f et de $\nabla(d^{\mathsf{T}} f)$ par (5.29) et (5.32) vérifie l'estimation suivante :*

$$\frac{T(f, \nabla(d^{\mathsf{T}} f))}{T(f)} \leqslant C'_{\mathcal{F}}, \tag{5.33}$$

où $T(f)$ est le temps nécessaire au calcul de f par (5.29) et $C'_{\mathcal{F}}$ est une constante ne dépendant que des fonctions intermédiaires φ_k de la bibliothèque $\mathcal{F}$.

Démonstration. L'examen des algorithmes (5.29) et (5.32) permet d'écrire :

$$T(f, \nabla(d^{\mathsf{T}} f)) = \sum_{k=1}^{K} T(\varphi_k, \nabla \varphi_k).$$

On ne compte pas dans $T(f, \nabla(d^{\mathsf{T}} f))$ le temps nécessaire à l'initialisation des variables p_i. Ensuite, en utilisant l'inégalité de Cauchy-Schwarz généralisée (A.9), on obtient

$$T(f, \nabla(d^{\mathsf{T}} f)) \leqslant \max_{1 \leqslant k \leqslant K} \left(\frac{T(\varphi_k, \nabla \varphi_k)}{T(\varphi_k)} \right) \sum_{k=1}^{K} T(\varphi_k) = C'_{\mathcal{F}}\, T(f),$$

où la constante

$$C'_{\mathcal{F}} := \max_{1 \leqslant k \leqslant K} \left(\frac{T(\varphi_k, \nabla \varphi_k)}{T(\varphi_k)} \right)$$

ne dépend que de la classe $\mathcal{F}$ des fonctions intermédiaires. □

On peut donner une estimation de $C'_{\mathcal{F}}$ lorsque les fonctions φ_k sont prises dans $\mathcal{F}_{_F}$ (famille des fonctions Fortran). Sous des hypothèses raisonnables concernant les temps d'exécution relatifs des fonctions Fortran, on a (voir [286 ; 1989] [243 ; 1991])

$$\frac{T(f, \nabla(d^{\mathsf{T}} f))}{T(f)} \leqslant 5.$$

Par conséquent, le calcul de $f(x)$ et de $\nabla(d^{\mathsf{T}} f(x))$ par la méthode décrite demande un temps de calcul au plus égal à 5 fois celui nécessaire au calcul de $f(x)$ et cela quel que

soit le nombre de variables et quelle que soit la complexité du programme original. Comme cela a été montré dans la démonstration ci-dessus, cette inégalité ne tient pas compte du surcoût lié aux accès-mémoire qui peuvent être importants en mode inverse. En pratique, cette estimation n'est donc pas toujours vérifiée. Cependant, même pour de grands codes, il est courant d'obtenir une borne $C'_{\mathcal{F}}$ voisine de 3.

Ce qu'il faut retenir sur le mode inverse :

1. Le mode inverse de différentiation est le bon mode pour calculer les dérivées partielles d'une seule variable (ou variable de sortie) par rapport à un grand nombre de variables (ou variables d'entrée).
2. C'est donc le bon mode pour calculer le gradient d'une application à valeurs scalaires $f : \mathbb{R}^n \to \mathbb{R}$ ou le gradient de $d^{\mathsf{T}} f$ si f est à valeurs vectorielles.
3. À chaque variable v_i du code, on associe une variable duale $\overline{v}_i$ qui contient la variation de la variable de sortie (ou de $d^{\mathsf{T}} f(x)$, pour un vecteur d donné, en cas de sortie vectorielle) par rapport à une perturbation de la variable v_i. Sa valeur est nulle si v_i n'a pas d'influence sur la variable de sortie.
4. Le code linéaire cotangent peut s'obtenir en « dualisant » le code original ligne par ligne. Cette dualisation doit se faire en ordre inverse de l'ordre d'exécution dans le code original. Si dans ce dernier une instruction s'interprète comme une application $\{v_i : i \in D_k\} \mapsto v_{\mu_k}$, les instructions correspondantes dans le code linéaire cotangent forment une application entre variables duales de la forme $(\overline{v}_i : i \in D_k \cup \{\mu_k\}) \mapsto \{\overline{v}_i : i \in D_k \cup \{\mu_k\}\}$.
5. La difficulté principale dans l'écriture de codes linéaires cotangents est la gestion-mémoire permettant de transmettre du code original au code linéaire cotangent, l'information nécessaire à l'évaluation des dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$.

5.6 Développement de codes d'optimisation

5.6.1 Communication directe et inverse ▲

La communication est bien organisée permettant à des groupes différents d'acteurs d'écrire des programmes indépendamment.

Nous appelons ci-dessous *optimiseur*, un code informatique dans lequel est implémenté un algorithme destiné à résoudre un problème d'optimisation ayant une structure bien précise. On dit que l'optimiseur est écrit en *communication directe* (CD), s'il est destiné à être utilisé comme à la figure 5.2. Dans cette figure, nous avons représenté en bleu et en pointillés la partie qui doit être écrite par l'utilisateur du code d'optimisation et en rouge et en lignes continues la partie du programme qui doit être écrite par l'auteur du code d'optimisation. L'optimiseur sera donc appelé par un programme écrit par l'utilisateur. L'optimiseur est écrit quant à lui indépendamment du problème à résoudre et lorsqu'il a besoin d'information sur ce problème, il appelle un *simulateur*, contenant le code qui évalue les objets définissant le problème et qui ont un sens pour l'optimiseur. Ainsi, en général, le simulateur évalue le critère $f(x)$ et/ou les contraintes $c(x)$ en l'itéré x fourni par l'optimiseur. C'est le fanion

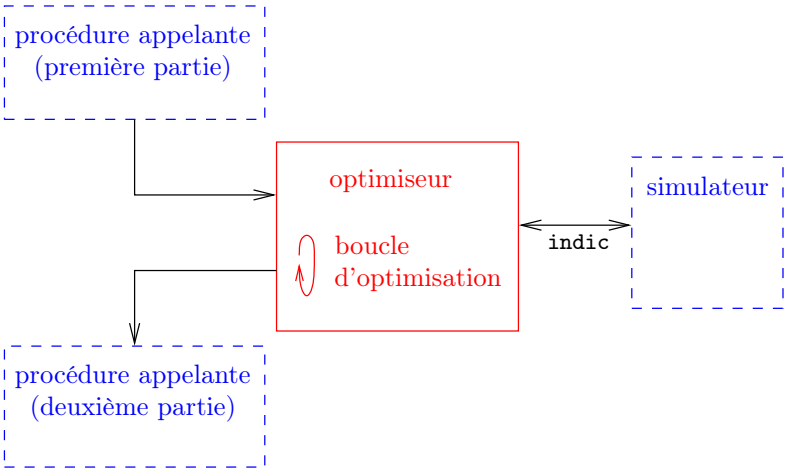


Fig. 5.2. Shéma d'un code avec un optimiseur utilisant la communication directe

`indic`, positionné par l'optimiseur, qui spécifiera ce qui lui est nécessaire. Le rôle de l'optimiseur est de mettre à jour x. Pour le dire autrement, l'optimiseur ne connaît pas le problème qu'il est occupé à résoudre et ne connaît que les objets abstraits qui décrivent la structure du problème; c'est en fait un code généraliste travaillant sur les objets mathématiques, comme nous allons le faire dans cet ouvrage. De son côté, le simulateur ne connaît pas les subtilités propres à la technique algorithmique mise en œuvre pour résoudre le problème (multiplicateur de Lagrange, fonction duale, facteur de pénalisation lui sont étrangers) et est décrit par les grandeurs physiques, chimiques, économiques ou mathématiques..., qui ont du sens pour lui.

Un optimiseur peut aussi être écrit en *communication inverse* (CI). L'organisation d'un tel code est représentée dans le schéma de la figure 5.3.

Le tableau 5.2 compare les deux types de communication. Le signe $\ominus$ signale que le mode en rapport avec ce signe est plus faible sur la caractéristique considérée que l'autre mode de communication. Le signe $\oplus$ signale le contraire : en ce qui concerne la caractéristique considérée, le mode en rapport avec ce signe est avantagé par rapport à l'autre mode. Passons en revue ces caractéristiques. L'inconvénient principal de

	communication directe (CD)	communication inverse (CI)
écriture du simulateur	$\ominus$	$\oplus$
environnement multi-langage	$\ominus$	$\oplus$
écriture de l'optimiseur	$\oplus$	$\ominus$

Tableau 5.2. Comparaison des modes de communication directe et inverse

la CD, selon nous, est de demander à l'utilisateur du code d'optimisation de devoir

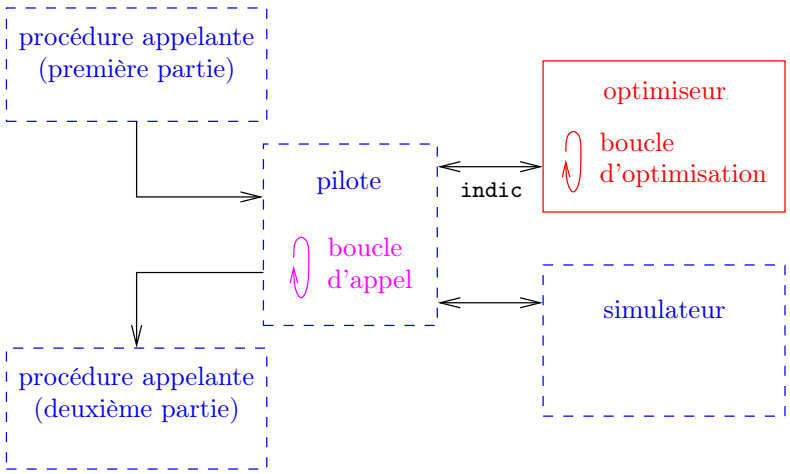


Fig. 5.3. Shéma d'un code avec un optimiseur utilisant la communication inverse

écrire un *simulateur*, c'est-à-dire une procédure à la séquence d'appel stricte réalisant les tâches demandées par l'optimiseur (calcul des fonctions définissant le problème d'optimisation et éventuellement de leurs dérivées). De plus, l'instruction d'appel de ce simulateur a sa structure figée par ce que le code d'optimisation a prévu. Cette rigidité est parfois gênante. Si des langages de programmation différents sont utilisés dans le code utilisateur et dans l'optimiseur, un seul interfaçage devra être introduit en CI et deux en CD. Il faut en effet que le programme appelant prévoit un appel de l'optimiseur dans son propre langage dans les deux modes de communication, mais en CD une interface doit aussi être prévue entre l'optimiseur et le simulateur supposés avoir été écrits dans des langages différents. Ainsi il peut être délicat d'utiliser un optimiseur écrit en Fortran et en CD dans un code Matlab.

5.6.2 Profils de performance

Comment comparer deux codes d'optimisation ? Comment savoir si une idée apporte une amélioration à un algorithme ? Ces questions ne se posent pas seulement en optimisation, mais dans tout domaine du calcul scientifique dans lequel on cherche à améliorer des solveurs généralistes. Il est coutumier de comparer les résultats des solveurs en question sur des bancs d'essai de problèmes-tests. Cela conduit à des tableaux de résultats qu'il n'est pas aisé d'analyser. Il est en effet rare qu'un solveur se comporte mieux qu'un autre sur tous les problèmes-tests de la collection considérée. Les profils de performance, que nous allons introduire, améliorent la situation en remplaçant les tableaux par des courbes dont l'interprétation est plus rapide pourvu que l'on en ait les clés de lecture.

Soient $\mathcal{S}$ un ensemble de solveurs et $\mathcal{P}$ un ensemble de problèmes-tests. Les profils de performance sont utilisés pour comparer l'efficacité *relative* des solveurs de $\mathcal{S}$ sur les problèmes-tests de $\mathcal{P}$. Soit

$$\tau_{p,s} := \text{performance du solveur } s \text{ sur le problème-test } p,$$

où la *performance* est un critère de comparaison, tel que le temps CPU ou le nombre d'évaluations de fonction ou de dérivée. Pour que ce qui suit ait un sens, il faut qu'une telle performance ait une valeur plus faible lorsque le solveur est meilleur. La *performance relative* d'un solveur s (par rapport aux autres solveurs de $\mathcal{S}$) sur un problème-test p est le rapport

$$\rho_{p,s} = \frac{\tau_{p,s}}{\tau_{p,\min}}, \quad \text{où } \tau_{p,\min} := \min\{\tau_{p,s'} : s' \in \mathcal{S}\}.$$

Bien sûr $\rho_{p,s} \geqslant 1$. Par ailleurs, un solveur s ne réussissant pas à résoudre un problème-test p devrait normalement avoir une performance $\tau_{p,s}$ infinie, ce qui ne se traite pas bien numériquement. Pour cette raison, on décide de limiter les performances relatives $\rho_{p,s}$ à une valeur maximale $\bar{\rho} > 1$ (ou de limiter les performances $\tau_{p,s}$ à $\bar{\rho}\,\tau_{p,\min}$). Dès lors, on aura $\rho_{p,s} = \bar{\rho}$, soit parce que le solveur s a en réalité sur le problème-test p une performance supérieure à $\bar{\rho}\,\tau_{p,\min}$, soit parce qu'il ne réussit pas à résoudre le problème p.

Le *profil de performance* du solveur s (par rapport aux autres solveurs de $\mathcal{S}$) est la fonction

$$t \in [1, \bar{\rho}] \mapsto \wp_s(t) := \frac{|\{p \in \mathcal{P} : \rho_{p,s} \leqslant t\}|}{|\mathcal{P}|} \in [0, 1],$$

où $|\cdot|$ désigne le cardinal d'un ensemble (le nombre de ses éléments). Cette fonction est croissante, constante par morceaux et semi-continue supérieurement. Il y a un profil pour chaque solveur de $\mathcal{S}$. La figure 5.4 donne un exemple de profils de performance,

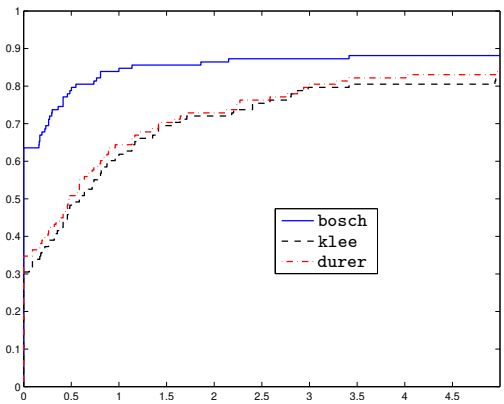


Fig. 5.4. Profils de performance typiques pour trois solveurs

correspondant à trois solveurs appelés `bosch`, `durer` et `klee`. Le solveur `bosch` apparaît plus performant que les deux autres, pour des raisons qui devraient être claires après avoir pris connaissance des clés de lecture données ci-après.

Seuls trois faits doivent être gardés à l'esprit pour avoir une interprétation correcte des profils de performance :

- $\wp_s(1)$ donne la fraction de problèmes-tests pour lesquels le solveur s est le meilleur; comme deux solveurs peuvent avoir le même score et que certains solveurs peuvent ne pas réussir à résoudre un problème-test, il n'est pas garanti que l'on ait $\sum_{s\in\mathcal{S}} \wp_s(1) = 1$;
- par définition de $\bar{\rho}$, $\wp_s(\bar{\rho}) = 1$; par ailleurs, pour $\varepsilon > 0$ petit, $\wp_s(\bar{\rho} - \varepsilon)$ donne la fraction de problèmes-tests que le solveur s peut résoudre; cette valeur est indépendante de la performance choisie pour la comparaison;
- on peut donner l'interprétation suivante de la valeur $\wp_s(t)$, obtenue en inversant l'application $t \mapsto \wp_s(t)$: pour la fraction $\wp_s(t)$ de problèmes-tests de $\mathcal{P}$, la performance du solveur s n'est jamais pire que t fois celle du meilleur solveur (celui-ci varie en général d'un problème-test à l'autre); de ce point de vue, l'argument auquel $\wp_s$ atteint sa valeur « presque » maximale $\wp_s(\bar{\rho} - \varepsilon)$ est significative.

Avec les profils de performance, l'efficacité relative des solveurs s de $\mathcal{S}$ apparaît en un coup d'œil : plus haut est le graphe de $\wp_s$, meilleur est le solveur s.

5.7 Pourquoi étudier l'optimisation numérique ?

Pourquoi est-il utile d'étudier les algorithmes d'optimisation ? La question mérite d'être posée. On pourrait en effet penser que ceci n'est plus nécessaire puisqu'il existe un certain nombre de bibliothèques de programmes d'optimisation (voir la section **??**) dans lesquels on pourra souvent trouver le code d'optimisation convenant au problème que l'on veut résoudre. Notons que ce point de vue pourrait être tenu à propos d'autres domaines de l'analyse numérique, tels que l'algèbre linéaire numérique, l'étude des schémas d'intégration des équations différentielles, les techniques de résolution des équations aux dérivées partielles, *etc.* Les remarques suivantes ne font pas le tour de la question, mais devraient permettre de nuancer cette position.

1. Face à un problème d'optimisation à résoudre, il faut faire un choix d'algorithmes. Si le problème a une structure classique (voir la section **??** pour une classification simplifiée) et ne demande pas trop de temps de calcul, ce choix pourra se faire facilement et un code de bibliothèque pourra être satisfaisant. Il est cependant nécessaire de bien comprendre les possibilités de ces codes et donc des algorithmes qui y sont implémentés pour faire un bon choix. Il faut aussi pouvoir saisir les subtilités de leur documentation pour pouvoir adapter avec pertinence leurs options au problème à résoudre, ce qui requiert également une bonne maîtrise des algorithmes implémentés.
2. Les codes d'optimisation fonctionnent rarement en boîte noire. Ceci est dû au fait qu'une solution d'un problème d'optimisation s'obtient grâce à un processus qui doit converger. Il ne s'agit donc pas d'un processus ayant un nombre déterminé d'opérations. Diagnostiquer un comportement inattendu est souvent délicat. Les codes d'optimisation bien conçus fournissent parfois ces diagnostics, mais il faut pouvoir les comprendre. Ici aussi une bonne connaissance des algorithmes est essentielle.
3. Les codes que l'on trouve dans les bibliothèques d'optimisation sont conçus pour résoudre des problèmes à la structure classique. Même si ceux-ci ont été écrits de

manière à ce qu'ils puissent être utilisés pour résoudre des problèmes variés, on gagnera souvent à les adapter au problème qui nous intéresse. Ceci est d'autant plus vrai que le problème est de grande taille, lorsque les temps de calcul sont important. Modifier les algorithmes de manière à exploiter au mieux la structure d'un problème demande de solides connaissances en algorithmique.

4. Le développement des algorithmes n'est pas terminé. Il est donc nécessaire d'enseigner ces matières aux futurs chercheurs ou ingénieurs, aux futurs numériciens qui feront progresser la discipline.

Cet ouvrage est donc destiné aux étudiants désirant se familiariser avec les techniques numériques en optimisation. Il s'adresse également aux praticiens souhaitant implémenter un algorithme ou comprendre le comportement d'un code d'optimisation sur un problème particulier. Il sera également utile aux chercheurs voulant connaître les techniques permettant d'analyser les algorithmes, d'étudier et de prévoir leur comportement. L'ouvrage contient en effet de nombreux résultats de convergence que l'on ne trouve que dans les revues spécialisées.

Notes

L'importance de la classe P a été soulignée par Edmonds [189 ; 1965] et Cobham [130 ; 1965]. Comme autres ouvrages sur la théorie de la complexité, citons [66, 475, 476].

L'utilisation de l'état adjoint pour calculer des produits hessienne-vecteur a été proposé en météorologie par Le Dimet, Navon et Daescu [383]. Le livre de Griewank (et Walther) [288, 291] et sa revue [289] constituent d'excellents états de l'art sur les techniques de différentiation automatique. Ils ont été écrits par l'un des spécialistes les plus actifs dans ce domaine. On pourra aussi se référer aux monographies [290] et [243]. Le comportement de la différentiation automatique sur des programmes contenant des processus itératifs est analysé dans [238 ; 1992]. Pour des résultats de complexité en différentiation automatique, nous renvoyons le lecteur à [449 ; 2008] et sa bibliographie.

La notion de profil de performance a été introduite par Dolan et Moré [171 ; 2002] et est maintenant couramment utilisée pour comparer l'efficacité des codes d'optimisation. Divers environnements de comparaison de codes permettent de les générer automatiquement, citons LIBOPT [242].

Exercices

5.1. Soient $\mathbb{E}$ un espace normé, dont la norme est notée $\|\cdot\|$, et $F : \mathbb{E} \to \mathbb{E}$ une application différentiable en $x_* \in \mathbb{E}$ dont la dérivée $F'(x_*)$ est inversible. Alors $F(x) \sim (x - x_*)$ au sens de (5.3).

5.2. *Vitesse de convergence en termes du nombre de chiffres significatifs corrects.* Démontrer les propositions 5.4, 5.6 et 5.11.

5.3. *Vitesse de convergence en termes d'une fonction s'annulant au point limite.* Démontrer les propositions 5.7 et 5.13.

5.4. *Convergence q-quadratique en termes de déplacements.* Démontrer la proposition 5.14.

5.5. *Suite de Fejér* [539]. Soit $\mathbb{E}$ un espace euclidien (produit scalaire $\langle\cdot,\cdot\rangle$ et norme associée $\|\cdot\|$). La *cible de Fejér* d'une suite $\{x_k\}\subseteq\mathbb{E}$ est l'ensemble

$$C(\{x_k\}) := \{x\in\mathbb{E} : \text{pour tout } k\in\mathbb{N}, \text{ on a } \|x_{k+1}-x\|\leqslant\|x_k-x\|\}.$$

On dit qu'une suite $\{x_k\}$ de $\mathbb{E}$ est une *suite de Fejér* si $C(\{x_k\})\neq\varnothing$.

1) $C(\{x_k\})$ est un convexe fermé (éventuellement vide).
2) Une suite de Fejér est bornée.
3) Si une suite $\{x_k\}$ a un point d'adhérence $\bar{x}\in C(\{x_k\})$, alors elle converge vers $\bar{x}$.
4) Si $\{x_k\}$ est suite de Fejér, alors

$$x_{k+1}\in\bigcap_{x\in C(\{x_k\})}\bar{B}(x,\|x_k-x\|).$$

Note. *Lipót Fejér* est un mathématicien hongrois (1880-1959). L'étude systématique des suites de Fejér et de leur lien avec l'optimisation a commencé avec le mathématicien russe Eremin [192, 193, 194 ; 1965-1979].

5.6. *Dérivée du critère du problème* (5.24)–(5.25). On considère le problème de commande optimale (5.24)–(5.25). Soit $f(u)$ le critère (5.25). Montrez que, pour $v:[0,T]\to\mathbb{R}^m$,

$$f'(u)\cdot v=\int_0^T\Big(J'_u(y(t),u(t),t)\cdot v(t)-p(t)^{\mathsf{T}}\varphi'_u(y(t),u(t),t)\cdot v(t)\Big)\,\mathrm{d}t,$$

où y est la solution de (5.24) et p est l'état adjoint défini comme solution de l'équation adjointe

$$\begin{cases}\dot{p}(t)=-\varphi'_y(y(t),u(t),t)^{\mathsf{T}}p(t)+\nabla_y J(y(t),u(t),t), & \text{pour tout } t\in\,]0,T[\\ p(T)=-\nabla J^T(y(T)).\end{cases}$$

5.7. *DA d'instructions à valeurs vectorielles.* Supposons que l'on ait dans le code original une instruction à valeurs vectorielles (par exemple par l'intermédiaire d'une procédure) de la forme

$$v_M:=\varphi(v_D),$$

où M et D sont des parties de $[1:N]$, pouvant être disjointes ou non et v_M désigne le vecteur de $\mathbb{R}^{|M|}$ dont les composantes sont les variables v_i du code avec indices $i\in M$ (désignation analogue pour v_D). On indice par $i\in M$ les composantes de φ. Montrez que les instructions du code linéaire tangent s'écrivent

$$\dot{v}_M:=\varphi'(v_D)\dot{v}_D,\quad v_M:=\varphi(v_D)$$

et celles du code adjoint s'écrivent

$$\overline{v}_D:=\overline{v}_{D\setminus M}+\varphi'(v_D)^{\mathsf{T}}\overline{v}_M,\quad \overline{v}_{M\setminus D}:=0. \tag{5.34}$$

5.8. *DA de solveurs linéaires.* Supposons qu'une partie d'un programme consiste à résoudre le système linéaire, ce que l'on peut écrire

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

où la matrice carrée inversible A et le vecteur $b\in\mathbb{R}^n$ peuvent dépendre des variables indépendantes, qu'il n'est pas nécessaire de spécifier ici. Montrez que les instructions du code linéaire tangent s'écrivent

$$x := A^{-1}b, \quad \dot{x} := A^{-1}(\dot{b} - \dot{A}x)$$

et celles du code adjoint s'écrivent

$$x := A^{-1}b, \quad \overline{x} := A^{-\mathsf{T}}\overline{x}, \quad \overline{b} := \overline{b} + \overline{x}, \quad \overline{A} := \overline{A} - \overline{x}x^{\mathsf{T}}, \quad \overline{x} := 0.$$

Que deviennent les instructions des codes tangent et adjoint si le code original s'écrit $b := A^{-1}b$ (la solution du système linéaire est placée dans b).

Remarque. Dans le cas où la matrice A a une *structure creuse* et où seuls les éléments d'indices $(i,j) \in N$ de A sont mémorisés, il n'est pas nécessaire de s'intéresser aux $\overline{A}_{ij}$ pour $(i,j) \notin N$ et on pourra ne mettre à jour que les éléments de $\overline{A}$ avec indices (i,j) dans N par $\overline{A}_{ij} := \overline{A}_{ij} - \overline{x}_i x_j$. Autrement dit, il ne faut pas introduire de variables duales associées aux éléments de A qui sont toujours nuls.

5.9. *DA d'une fonction quadratique.* Donner les codes direct et adjoint d'un code qui calcule la fonction quadratique

$$f := \frac{1}{2}\, x^{\mathsf{T}}Ax + b^{\mathsf{T}}x.$$