L'ENJEU DE LA DIFFÉRENTIATION AUTOMATIQUE DANS LES MÉTHODES DE NEWTON D'ORDRES SUPÉRIEURS

par

Romain Cotte

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 9 décembre 2015

Sommaire

Les méthodes plus avancées d'optimisation avec ou sans contraintes nécessitent le calcul des dérivées de la fonction. En ce sens, la différentiation automatique est devenu un outil primordial. Malgré le fait qu'il soit omniprésent, cet outil est encore en développement et en recherche. Il ne présente pas les inconvénients classiques des méthodes habituelles de dérivation mais reste complexe à utiliser. Ce travail consiste à utiliser un outil de différentiation permettant de calculer des dérivées d'ordres supérieurs afin d'obtenir des directions améliorées. Nous définirons d'abord de manière générale un type d'algorithme d'optimisation à l'aide des directions suffisamment descendantes. Leurs caractéristiques seront analysées pour modifier des méthodes de type Newton afin d'avoir une meilleure fiabilité de convergence. Nous étudierons les opérations critiques et l'ordre du coût de ces méthodes
Dans une deuxième partie, nous verrons les calculs d'algèbre linéaire requis pour nos algorithmes Ensuite, nous présenterons le fonctionnement de la différentiation automatique et en quoi c'en est un outil indispensable à ce genre de méthodes. Puis, nous expliquerons pourquoi nous avons choisi l'outil Tapenade pour la différentiation automatique et la librairie de Moré, Garbow, Hillstrom pour la collection de fonctions tests. Enfin, nous comparerons les méthodes de types Newton.

Mots-clés: différentiation automatique ; Tapenade ; optimisation ; méthode de Newton ; ordres supérieurs.

Sommaire

Remerciements

Je souhaite exprimer mes sincères remerciements à mon directeur de recherche, Jean-Pierre Dussault, qui a toujours été disponible pour moi. Son aide m'a été particulièrement précieuse et m'a fait progresser pour aller de l'avant. C'est aussi une personne avec un coté humain très agréable et avec qui il a été très intéressant et plaisant de travailler.
Ensuite, je voudrais remercier mes deux colocataires, Emmanuelle Meunier et Maggie Poudrier, qui furent très accueillantes et chaleureuses. Elles m'ont fait découvrir la région du Québec et sont à l'image de ce que sont beaucoup de gens ici.
Je voudrais surtout remercier mes parents qui m'ont toujours soutenu dans mes études et qui me soutiendraient dans n'importe quel voie. Enfin, je remercie Martin Guay, pour ses conseils et son soutient.

Remerciements

Abréviations et notations

AMPL *A Mathematical Programming Language*

BK *Bunch-Kaufman*

CUTEr *A Constrained and Unconstrained Testing Environment, revisited*

DA Différentiation Automatique

DED Demi Espace de Diminution

GAO Graphe Acyclique Orienté

INRIA Institut National de Recherche en Informatique et en Automatique

LAPACK *Linear Algebra PACKage*

MGH *Moré, Garbow et Hilstrom*

NaN *Not a Number*

RA *Recompute-All* Tout recalculer

SA *Store-All* Tout stocker

SIF *Standard Input Format*

. M caractère en majuscule pour les matrices

. c en minuscule pour les vecteurs colonnes

. c^T la transposée : $c^T = \begin{pmatrix} c_1 & c_2 & \dots & c_n \end{pmatrix}$

. $\nabla f(x)$ le gradient de f en x : $\nabla f(x) : \mathbb{R}^n \to \mathbb{R}$ une matrice ligne
$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \dots & \frac{\partial f}{\partial x_n} \end{pmatrix}$
$F(x)^T := \nabla f(x)$

- $\nabla^2 f(x) : \mathbb{R}^n \to \mathbb{R}^n$ le hessien de la fonction.

$$\nabla^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \end{pmatrix} \text{ symétrique}$$

- $\#(f)$ correspond au coût de l'évaluation de f
- `[ ]` correspond à la liste vide
- `[`a`]` la liste composée d'un élément : a
- :: opérateur de contructeur de liste a ::`[ ]`=`[`a`]`, $t :: \bar{q}$ où t est un élément et $\bar{q}$ est une liste
- $\|v\| = \|v\|_2 = \sqrt{\sum_{i=1}^n v_i^2} = \sqrt{v^T v}$
- `>$` correspond au prompt bash sous unix
- `-->` correspond à *Scilab*
- $\neg$ non logique

Table des matières

Liste des figures

Liste des tableaux

Introduction

C'est par une représentation mathématique d'un phénomène physique, économique, humain que la programmation mathématique cherche à trouver un optimum, c'est-à-dire l'état jugé le meilleur ou le plus favorable à un problème. Plus précisément, la programmation non-linéaire est une méthode permettant de résoudre des équations et inéquations qui généralement modélisent le phénomène de notre modèle. Le but est de calculer un point minimisant (ou maximisant) une fonction objectif. Le problème peut être soumis à un ensemble de contraintes, ce qui aura pour effet de réduire le domaine de réalisabilité. L'étude faite dans ce texte se limite à la programmation non-linéaire sans contraintes. Nous allons voir qu'il existe de nouveaux types d'algorithmes basés sur celui de Newton. Ils bénéficient d'un meilleur ordre de convergence, néanmoins les calculs requis pour chaque itération sont plus poussés ; calculs de dérivées d'ordre supérieur. Ce travail consiste à utiliser les progrès de la Différentiation Automatique (DA) afin d'observer si le compromis entre l'ordre de convergence et le temps de calcul est raisonnable. Le calcul de dérivées est un domaine complexe, d'autant plus que nous avons besoin d'efficacité et de précision d'une part et de pouvoir automatiser ces calculs d'autre part. Ainsi, l'outil de DA semble un outil idéal car contrairement aux différences finies, il est capable de fournir le gradient de notre fonction à un coût proportionnel à l'évaluation de la fonction donc à un coût raisonnable. Par exemple, en grande dimension, de l'ordre de 10000, alors qu'il faut plusieurs secondes pour obtenir le gradient par différences finies, la DA est capable de le calculer presque instantanément ($\leq 4ms$). Bien qu'il soit encore en constant progrès, il a déjà fait ses preuves et est largement utilisé en optimisation comme avec le langage AMPL *A Mathematical Programming Language*. Un outil de DA a été élaboré (*Sciad*) par Benoit Hamelin, et une estimation des coûts de calcul

a été faite par décompte du nombre d'opérations. Cependant, il n'est pas exploitable pour des grandes dimensions (≥ 5). Nous allons développer un environnement d'expérimenation efficace au sein de *Scilab* afin de comparer les méthodes d'optimisation grâce à la DA qui concrétise les progrès précédents en termes de temps de calcul. Cet outil nous permettra d'atteindre des dérivées supérieures. Cela ouvre la voie à de nouvelles méthodes que nous allons tester.

Dans une première partie nous introduirons le problème d'optimisation, les méthodes de descentes et leur complexité parallèlement à leur ordre de convergence. Ensuite, nous étudierons les opérations critiques ; la résolution de systèmes linéaire d'une part et le calcul des dérivées d'autre part. Par conséquent, nous présenterons le fonctionnement de la DA avec deux modes d'exécution, le mode direct et le mode inverse qui ont différentes complexités. Puis, après avoir détaillé les outils avec les résultats obtenus nous comparerons les méthodes d'ordres supérieurs en terme d'itérations et de temps de calculs.

Chapitre 1

Algorithmes pour l'optimisation sans contraintes

1.1 Introduction aux directions de descente

Commençons par définir le contexte avec un peu de vocabulaire et les hypothèses faites sur le problème. Puis nous allons analyser les concepts nécessaires pour la suite.

1.1.1 Hypothèses de travail

Dans l'ensemble du texte, nous faisons deux hypothèses : la continuité, voir les définitions A.1, et la différentiabilité. Soit le problème d'optimisation suivant :

$$\min_{x \in \mathbb{R}^n} f(x). \tag{1.1}$$

Il s'agit d'un problème sans contraintes. La fonction objectif f, à valeurs de $\mathbb{R}^n$ dans $\mathbb{R}$, est continue et différentiable. L'ordre de différentiabilité va dépendre de la méthode choisie. Cela exclut les problèmes en nombres entiers. On parle de problème d'optimisation à n variables de décision avec $0 < n$. Il existe deux types de solutions : les minima locaux, dont aucun point de leur voisinage n'est meilleur et les minima globaux, dont aucun des points du domaine n'est meilleur. Par la suite, nous ne traiterons que les minima locaux.

En notant $\nabla f(x)$ ou $F(x)^T : \mathbb{R}^n \to \mathbb{R}^n$ le gradient de la fonction objectif, une matrice ligne et $\nabla^2 f(x)$ ou $\nabla F(x) : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ son hessien, la condition nécessaire d'optimalité indique que si x^* est un minimum local et que f est différentiable dans un voisinage ouvert V de x^* alors

$$\nabla f(x^*) = 0. \tag{1.2}$$

Ces points sont nommés points stationnaires. Si, de plus, f est deux fois différentiable sur V alors

$$\nabla^2 f(x^*) \text{ est semi définie positive.} \tag{1.3}$$

La condition (1.2) s'appelle la condition nécessaire du premier ordre et la condition (1.3) correspond à la condition nécessaire du second ordre. Lorsque la matrice est définie positive, il s'agit d'une condition suffisante, [**?**].

1.1.2 Méthodes avec recherche linéaire

Soit une fonction $f : \mathbb{R}^n \to \mathbb{R}$ continûment différentiable sur $\mathbb{R}^n$. Définissons la fonction $h_{x,d}(\theta) = f(x + \theta d)$ qui permet de se placer dans le sous espace de $\mathbb{R}^n$ de dimension un. Pour le problème de minimisation (1.1), les algorithmes couramment utilisés sont généralement les algorithmes de descente car ils permettent d'obtenir une convergence plus forte que pour des problèmes d'équations non linéaires.

Donnons la définition d'une direction de descente.

Définition 1.1.1 *Soit $x \in \mathbb{R}^n$ et $d \neq 0$ un vecteur de $\mathbb{R}^n$, alors d est une direction de descente de f au point x s'il existe $0 < \theta_m$ tel que pour tout $\theta \in]0, \theta_m]$, $h_{x,d}(\theta) < f(x)$.*

Il s'agit d'algorithmes itératifs basés sur le fait que si un point x ne satisfait pas aux conditions d'optimalité, alors il est possible de construire un autre point x' qui vérifie $f(x') < f(x)$. L'ensemble du *Demi Espace de Diminution* en x, noté $DED(x)$ est l'ensemble des directions qui satisfait à la relation : $\nabla f(x)d < 0$. Ces algorithmes ont tous la même forme ; trouver une direction dans le $DED(x)$ et ensuite approcher la fonction $h_{x,d}$ pour passer du point x_k au suivant $x_{k+1} = x_k + \theta d$. Néanmoins, il faut s'assurer que la suite x_k possède bien des points d'accumulation satisfaisant aux conditions.

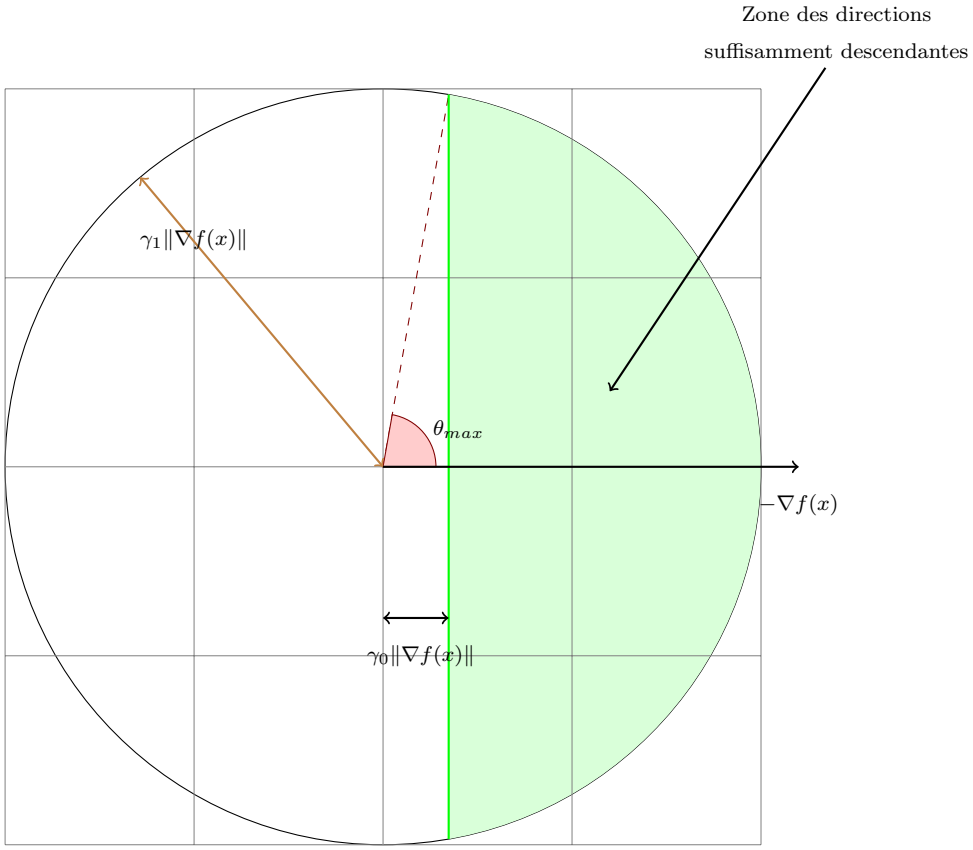
Définition 1.1.2 *Une direction d est considérée suffisamment descendante s'il existe deux constantes positives γ_0 et γ_1 indépendantes de x telles que d satisfasse aux inégalités suivantes :*

$$\nabla f(x)d \leq -\gamma_0 \|\nabla f(x)\|^2, \tag{1.4}$$

$$\|d\| \leq \gamma_1 \|\nabla f(x)\|. \tag{1.5}$$

Cette définition assure que toute direction d utilisée par un algorithme de descente est un vecteur assez long, et fait un angle assez aigu avec l'opposé de ∇f. La stratégie, appelée *linesearch*, consiste à minimiser $h_{x,d}(\theta)$ par rapport à θ. Évidemment, le minimum θ_m est approximatif, nous aurons pas besoin d'une précision aussi grande que le minimum x^*. On peut démontrer que pour une point dans le voisinage de la solution

figure 1.1 – Directions suffisamment descendantes



vérifiant les conditions suffisantes du second ordre, pour des directions étudiées dans ce mémoire, la recherche linéaire n'est plus active.

Définition 1.1.3 *Un pas θ est dit admissible pour une direction suffisamment descendante d lorsqu'il satisfait aux deux inégalités suivantes, nommées critère d'Armijo et de Wolfe respectivement :*

$$f(x+\theta d) - f(x) \leq \tau_0 \theta \nabla f(x) d, \;\; \tau_0 \in]0, \frac{1}{2}[\qquad \text{(Armijo)}$$

$$\tau_1 \nabla f(x) d \leq \nabla f(x+\theta d) d, \;\; \tau_1 \in]\tau_0, 1[. \qquad \text{(Wolfe)}$$

Famille de directions suffisamment descendantes Considérons le cas général d'une direction $\bar{d} = -H \nabla f(x)^T$, comme la direction de Newton, il s'agit d'une transformation linéaire de la direction de pente la plus forte. En supposant que H est une matrice définie positive, alors la direction $\bar{d}$ vérifie les conditions d'une direction suffisamment descendante. En effet

$$\nabla f(x) \bar{d} = -\nabla f(x) H \nabla f(x)^T.$$

En notant λ_{min} la plus petite valeur propre de H A.1 , on a

$$\nabla f(x) H \nabla f(x)^T \geq \lambda_{min} \|\nabla f(x)\|^2,$$

$$\nabla f(x) \bar{d} \leq -\lambda_{min} \|\nabla f(x)\|^2.$$

Et d'autre part

$$\|\bar{d}\| \leq \lambda_{max} \|\nabla f(x)\|.$$

L'ensemble des algorithmes de descente peut être généralisé sous la forme suivante :

Algorithm 1 Algorithme de descente

while ¬ fini **do**
  $d \leftarrow$ direction qui satisfait la définition 1.1.2
  $\theta \leftarrow$ qui satisfait la définition 1.1.3, les critères d'Armijo et Wolfe
  $x_{k+1} \leftarrow x_k + \theta d$
end while

Théorème 1.1.1 *Soit un algorithme de descente appliqué au problème :*

$$\min_{x\in\mathbb{R}^n} f(x),\ \ f \in \mathcal{C}^1(\mathbb{R}^n)$$

supposons qu'à chaque itération, la direction utilisée d_k est une direction suffisamment descendante, et pour laquellle le pas utilisé dans cette direction est un pas admissible ; alors, tous les points d'accumulation de la suite $\{x_k\}$ engendrée par l'algorithme sont des points stationnaires pour le problème $\min f(x)$.

1.2 Recherche linéaire

Pour s'assurer que les méthodes utilisées convergent correctement, une possibilité est d'utiliser une recherche linéaire. En effet, celle-ci va nous garantir que l'on ne s'éloigne pas trop du point courant. La direction de Newton par exemple peut fournir des directions de norme élevée et du même coup la convergence n'est pas systématique. Ceci se remarque d'autant que la dimension est grande. Il existe plusieurs techniques pour nous assurer que la valeur de la fonction objectif diminue bien au court des itérations si on possède une direction de descente, comme la région de confiance [**?**].

1.3 Méthode de Newton

La méthode de Newton joue un rôle central dans la résolution d'équations non linéaires et ainsi dans l'optimisation non linéaire. Elle permet de trouver les racines d'une fonction. Comme le montre la condition nécessaire du premier ordre, il faut trouver un point tel que $F(x)^T := \nabla f(x) = 0$.

L'idée est de simplifier notre équation, très souvent complexe, en une équation plus simple : une équation quadratique. Pour obtenir cette simplification nous utilisons la relation de Taylor.

Définition 1.3.1 ***(Modèle quadratique d'une fonction)***
Soit $f : \mathbb{R}^n \to \mathbb{R}$ une fonction deux fois différentiable. Le modèle quadratique de f en

$\bar{x}$ est une fonction $q_{\bar{x}} : \mathbb{R}^n \to \mathbb{R}$ définie par

$$q_{\bar{x}}(x) = f(\bar{x}) + \nabla f(\bar{x})(x - \bar{x}) + \frac{1}{2}(x - \bar{x})^T \nabla^2 f(\bar{x})(x - \bar{x})$$

où $\nabla f(\bar{x})$ est le gradient de f en $\bar{x}$ et $\nabla^2 f(\bar{x})$ est la matrice hessienne de f en $\bar{x}$. En posant $d = x - \bar{x}$, on obtient la formulation équivalente :

$$q_{\bar{x}}(d) = f(\bar{x}) + \nabla f(\bar{x})d + \frac{1}{2}d^T \nabla^2 f(\bar{x})d.$$

Si nous minimisons le modèle quadratique au lieu de la fonction :

$$\min_{d \in \mathbb{R}^n} q_{\bar{x}}(d) = f(\bar{x}) + \nabla f(\bar{x})d + \frac{1}{2}d^T \nabla^2 f(x)d.$$

La condition suffisante d'optimalité du premier ordre nous donne :

$$\nabla q_{\bar{x}}(d) = \nabla f(\bar{x}) + \nabla^2 f(x)d = 0.$$

L'équation $\nabla^2 f(\bar{x})d_N = -\nabla f(\bar{x})$ est appelée équation de Newton et d_N direction de Newton. En supposant que la matrice $\nabla^2 f(x)$ est définie positive et donc inversible, la solution revient à trouver le minimum du modèle quadratique de la fonction en x_k, d'où :

$$x_{k+1} = \arg\min_{x \in \mathbb{R}^n} q_{x_k}(x)$$

La solution peut s'écrire

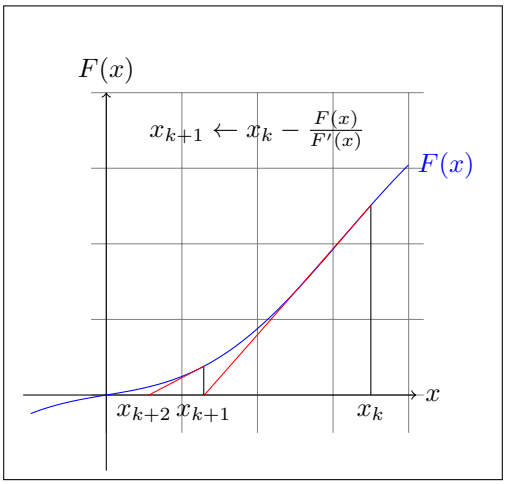
$$x_{k+1} \leftarrow x_k \underbrace{-\nabla F(x_k)^{-1} F(x_k)}_{d_N}$$

$$\text{où } d_N = -\nabla^2 f(x)^{-1} \nabla f(x)^T.$$

L'idéal est de commencer à partir d'une approximation de x^*, notre minimum local ; nommons le x_0. On calcule d'abord le modèle quadratique en x_0 pour obtenir son minimum x_1. Si les conditions d'optimalité sont satisfaites alors l'algorithme s'arrête, sinon on recalcule l'approximation quadratique en x_1.

Dans le cas unidimensionnel, avec plusieurs expérimentations, nous pouvons consta-

figure 1.2 – Deux itérations de la méthode de Newton dans $\mathbb{R}$



ter que la méthode de Newton converge très vite lorsque

- la fonction n'est pas trop non linéaire c'est-à-dire que les variations de la fonction ne sont pas trop grandes pour une petite variation de x.
- la dérivée de la fonction n'est pas trop proche de 0.
- le point initial x_0 n'est pas trop loin de la solution.

Si une de ces conditions n'est pas satisfaite, il se peut que l'algorithme diverge.

Fourier [**?**] a démontré la convergence quadratique locale dans le cas réel mais le théorème de Kantorovich nous assure la convergence sous certaines conditions dans le voisinage de x_0. De plus, il donne une borne de l'erreur pour chaque itéré.

Théorème 1.3.1 *(Kantorovich [**?**]) Soit $x_0 \in D_0$ tel que $\nabla F(x_0)^{-1}$ existe et que*

$$\|\nabla F(x_0)^{-1}\| \leq B$$

$$\|\nabla F(x_0) F(x_0)\| \leq \eta$$

$$\|\nabla F(x_0) - \nabla F(y)\| \leq K\|x - y\| \text{ pour tout } x \text{ et } y \text{ dans } D_0$$

avec $h = BK\eta \leq \frac{1}{2}$
Soit $\Omega_* = \{x \mid \|x - x_0\| \leq t^*\}$ où $t^* = \left(\frac{1-\sqrt{1-2h}}{h}\right)\eta$
Si $\Omega_* \subset D_0$ alors les itérations de Newton ; $x_{k+1} = x_k - \nabla F(x_k)^{-1} F(x_k)$ sont bien

définies, restent dans Ω_* *et convergent vers* $x_* \in \Omega_*$ *tel que* $F(x^*) = 0$. *De plus,*

$$\|x_* - x_k\| \leq \frac{\eta}{h}\left(\frac{(1-\sqrt{1-2h})^{2^k}}{2^k}\right) \quad k = 0,\ 1,\ 2,\ \cdots$$

Un théorème de convergence pour une méthode itérative est appelé un théorème de convergence locale lorsque l'on suppose l'existence d'une solution x^* et le point initial x_0 est suffisamment proche de x^*. D'autre part, un théorème de convergence tel que 1.3.1, qui ne suppose pas l'existence d'une solution mais suppose certaines conditions sur x_0 est appelé une théorème de convergence semi-locale.

Sans appliquer les conditions des définitions 1.1.2 et 1.1.3, l'algorithme de Newton a une convergence locale et semi-locale.

1.3.1 Ordre de convergence

Pour pouvoir comparer les algorithmes, nous définissons la vitesse de convergence qui est le témoin théorique de l'efficacité de la méthode.

Définition 1.3.2 *La vitesse de convergence de la suite* $\{x_k\}$ *vers le point* x_*, *telle que* $\forall k,\ x_k \neq x^*$ *s'exprime à l'aide des scalaires* p *et* γ *dans l'expression suivante :*

$$\limsup_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^p} = \gamma < \infty$$

L'ordre de convergence de la suite est donné par la plus grande valeur que p *puisse prendre pour que la limite ci-haut demeure finie. Lorsque p=1,* γ *est nommée le taux de convergence.*

Le cas où $p = 1$ est dite convergence linéaire, le cas $p = 2$, convergence quadratique, $p = 3$ cubique, plus p est élevée et plus la méthode sera efficace.

Théorème 1.3.2 *Soit* x^* *une racine isolée de la fonction* g *telle que* $g'(x^*) \neq 0$, *avec la fonction* g' *lipschitzienne (A.1), Alors, il existe un voisinage de* x^* *tel que si la méthode de Newton est initialisée dans ce voisinage, elle produit une suite convergeant vers* x^* *et la vitesse de convergence asymptotique est quadratique.*

La condition pour que x_0 soit proche de la solution se traduit par la convergence locale, c'est-à-dire que x_0 doit être choisi dans un certain voisinage de la solution. Le fait que la fonction ne soit pas trop non linéaire correspond au caractère lipschitzien de la fonction. Par exemple, l'algorithme de Newton aura beaucoup de mal à trouver le zéro de l'équation $\frac{1}{x} - C$ où C est une constante positive, si l'on part d'un point x_0 proche de zéro. Enfin, le théorème de Kantorovich suppose que $F(x_0)^{-1}$ existe. Pour un ordinateur il faudrait que $\|F(x_0)^{-1}\| \geq \epsilon > 0$ à cause des erreurs d'arrondis et d'annulation.

La direction d_N est suffisamment descendante si la matrice $\nabla^2 f(x)$ est définie positive. Dans le cas contraire, la suite $\{x_k\}_k$ peut diverger, c'est pour cela que cette matrice va être modifiée pour devenir définie positive.

1.3.2 Itération de Newton modifiée

Dans le but de satisfaire les conditions pour la méthode de descente, il faut modifier la direction d_N. Sinon, la méthode peut ne plus être globalement convergente. De la même manière, on considère l'approximation d'ordre deux pour trouver le minimum de f sauf qu'au lieu d'avoir la matrice hessienne, nous avons une modification : B_k

$$f(x_k + d) = f(x_k) + \nabla f(x_k)d + \frac{1}{2}d^T B_k d$$

où $B_k \simeq \nabla^2 f(x)$. On va chercher le problème d'optimisation par rapport à la direction d :

$$\min_d \nabla f(x_k)d + \frac{1}{2}d^T B_k d.$$

En dérivant par rapport à d, la condition nécessaire du premier ordre fournit la relation :

$$\nabla f(x_k)^T + B_k d = 0$$

$$d = -B_k^{-1}\nabla f(x_k)^T.$$

Le hessien $\nabla^2 f(x)$ va être transformé pour le rendre défini positif. Il suffit par exemple de prendre :

$$B_k = \nabla^2 f(x_k) + \max(-\lambda_{min} + \epsilon, 0)I$$

où λ_{min} est la plus petite valeur propre de $\nabla^2 f(x_k)$ et $\epsilon > 0$.

L'inconvénient de cette formule est dans le calcul de λ_{min}, il faut avoir l'ensemble des valeurs propres (voir A.1) de la matrice et ce calcul a une complexité en $\mathcal{O}(n^3)$ avec une constante implicite plutôt défavorable. Nous allons directement nous aider de la décomposition de Cholesky pour modifier la diagonale. Ainsi, on espère avoir une complexité totale inférieure.

1.4 Méthodes d'ordre supérieur à deux

Les méthodes de Halley et de Chebychev sont des techniques célèbres pour résoudre des équations non linéaires. Ces algorithmes sont très proches de la méthode de Newton et ont une convergence cubique. Le gain de convergence s'obtient par une analyse plus précise de la fonction puisqu'elles requièrent la dérivée seconde de F donc la dérivée troisième de l'objectif f. En fin de section nous verrons une méthode du même type qui a fait l'objet de recherches récentes.

1.4.1 Méthode de Halley

Cette méthode a été découverte par Edmond Halley (1656-1742), elle s'applique à une fonction $\mathcal{C}^2$. Au lieu de faire une approximation linéaire de la fonction F, on part d'une approximation quadratique :

$$F(x+d) = F(x) + \nabla F(x)d + \frac{1}{2}d^T\nabla^2 F(x)d + \mathcal{O}(\|d\|^3)$$

Cauchy a démontré sous certaines conditions la convergence semi-locale cubique. En effectuant le développement de Taylor limité $\sqrt{1-x} \simeq 1 - \frac{1}{2}x$, on obtient la méthode de Halley (1694) :

$$x_{k+1} = x_k - [\nabla F(x_k) - \frac{1}{2}\nabla^2 F(x_k)\nabla F(x_k)^{-1}F(x_k)]^{-1}F(x_k)$$

Cela revient à résoudre les différents systèmes :

$$F(x_k) + \nabla F(x_k)c_k = 0 \Leftrightarrow c_k = -\nabla F(x_k)^{-1}F(x_k)$$

$$F(x_k) + \nabla F(x_k)d_k + \frac{1}{2}\nabla^2 F(x_k)c_k d_k = 0 \Leftrightarrow d_k = -[\nabla F(x_k) - \frac{1}{2}\nabla^2 F(x_k)c_k]^{-1}F(x_k)$$
$$x_{k+1} = x_k + d_k, 0 \leq k$$

1.4.2 Méthode de Chebychev

Chebyshev proposa une méthode d'ordre deux en 1841, de convergence cubique :

$$x_{k+1} = x_k - \nabla F(x_k)^{-1}F(x_k) - \frac{1}{2}\nabla F(x_k)^{-1}\nabla^2 F(x_k)[\nabla F(x_k)^{-1}F(x_k)]^2 \qquad (1.6)$$

ce qui revient à résoudre :

$$F(x_k) + \nabla F(x_k)c_k = 0 \Leftrightarrow c_k = -\nabla F(x_k)^{-1}F(x_k)$$
$$F(x_k) + \nabla F(x_k)d_k + \frac{1}{2}\nabla^2 F(x_k)c_k^2 = 0 \Leftrightarrow d_k = -c_k - \frac{1}{2}\nabla F(x_k)^{-1}\nabla^2 F(x_k)c_k^2$$
$$x_{k+1} = x_k + d_k, 0 \leq k$$

1.4.3 Méthode d'extrapolation d'ordre trois

Les méthodes présentées peuvent être résumées comme suit. Soit F une fonction de $\mathbb{R}^n$ dans $\mathbb{R}^n$. Chaque méthode est vue comme une direction de déplacement avec laquelle une suite d'itérés est construit

$$x_{k+1} = x_k + d_k$$

La condition nécessaire du premier ordre doit être satisfaite, on cherche x^* tel que

$$F(x^*) = 0$$

La méthode de Newton revient à résoudre le système :

$$F(x) + \nabla F(x)d_N = 0.$$

Celle de Halley revient à faire :

$$F(x) + \nabla F(x)d_H + \frac{1}{2}\nabla^2 F(x)d_N d_H = 0.$$

Et celle de Chebychev :

$$F(x) + \nabla F(x)d_C + \frac{1}{2}\nabla^2 F(x)d_N d_N = 0.$$

À partir de ces directions de Halley, Newton et Chebyshev, d'après [**?**], on peut développer de nouvelles méthodes sur le même plan mais à un ordre supérieur :

$$F(x) + \nabla F(x)d + \frac{1}{2}\nabla^2 F(x)d_1 d_2 + \frac{1}{6}\nabla^3 F(x)d_3 d_4 d_5 = 0 \tag{1.7}$$

où la direction recherchée est d et les directions d_i sont des directions connues.

Les méthodes qui viennent d'être présentées ne sont pas universelles et souffrent des mêmes défauts que la méthode de Newton à savoir que la convergence est seulement locale et les fonctions doivent être lipschitziennes. Comme la plupart des algorithmes en optimisation, il n'existe pas de méthode meilleure que toutes. Dans certains cas de figure, les méthodes qui sont a priori moins efficaces ; c'est-à-dire de moins bonne convergence, peuvent résoudre certains programmes non linéaires en moins d'itérations.

1.5 Ordre de la complexité

Une borne théorique de la complexité bien connue est celle de Griewank [**?**], qui énonce que le coût d'évaluation du gradient nécessite jamais plus de cinq fois le coût de l'évaluation de la fonction en mode inverse et n fois le coût de l'évaluation en mode direct. Par ailleurs, Mihael Ulbrich et Stephan Ulbrich [**?**] donnent des bornes plus précises sur les deux modes tangent et direct de la DA, étudiés plus loin au chapitre 3.2. En notant $\#(f)$, le coût d'évaluation de f, les bornes de complexité pour le mode direct sont :

$$(n+1)\#(f) \leq \#(f, \nabla f) \leq (3n+1)\#(f)$$

$$\frac{n^2+3n+2}{2}\#(f) \leq \#(f, \nabla f, \nabla^2 f) \leq \frac{7n^2+11n+2}{2}\#(f)$$

On remarque qu'en mode direct, le coût du gradient est de l'ordre de la dimension de l'espace de définition par le coût de la fonction. Cela va vite devenir lourd si nous voulons obtenir des ordres trois, voire quatre. En revanche, avec le mode inverse la borne de complexité, dûe à W. Baur et V. Strassen[**?**], ne dépend plus de n et nous avons :

$$\#(f, \nabla f) \leq 4\#(f)$$

$$\#(f, \nabla^2 f) \leq 16n\#(f)$$

En réalité, dans beaucoup de cas, nous avons besoin du calcul du gradient multiplié par un vecteur ou de la hessienne multipliée par un vecteur, ce qui va simplifier la complexité avec la DA. Bien qu'il y ait une remarque sur l'opération $\nabla^2 f.d$, M. et S. Ulbrich ne donnent pas de borne précise pour cette opération. Quand on applique le mode direct dans une certaine direction, le coût est proportionnel au coût de f. En généralisant aux dérivations supérieures, les opérations $\nabla^3 f \cdot u \cdot v$ et $\nabla^4 f \cdot u \cdot v \cdot w$ ne dépendent pas de n. Ce qui est le plus surprenant dans le tableau c'est qu'il est possible d'obtenir le gradient qui est de dimension n à un coût proportionnel à la fonction. Intuitivement, on pourrait croire qu'il est possible d'obtenir de la même manière les hessiens et ordres plus élevés avec le même coût. En réalité, nous verrons que ce résutlat ne peux pas s'obtenir avec une certaine contrainte de stockage.

Opération	coût
Gradient : $\nabla f(x)$	$\leq 4\#(f)$
Hessien : $\nabla^2 f(x)$	$\leq 16n\#(f)$
Hessien$\times$ vecteur : $\nabla^2 f(x) \cdot v$	$\mathcal{O}(\#(f))$
$\nabla^3 f\times$ vecteur $\times$ vecteur : $\nabla^3 f(x) \cdot v_1 \cdot v_2$	$\mathcal{O}(\#(f))$
$\nabla^3 f\times$ vecteur : $\nabla^3 f(x) \cdot v_1$	$\mathcal{O}(n\#(f))$
$\nabla^4 f\times$ vecteur $\times$ vecteur$\times$ vecteur : $\nabla^4 f(x) \cdot v_1 \cdot v_2 \cdot v_3$	$\mathcal{O}(\#(f))$

Bien sûr, il s'agit de bornes approximatives, nous verrons ce qu'il en est en pratique. Voyons maintenant en détail l'ordre de complexité de chaque méthode.

1.5.1 Newton modifié

Pour calculer la direction de Newton modifiée, le calcul du gradient et du hessien sont nécessaires. En effet, la direction est obtenue en résolvant le système

$$\nabla^2 f(x) d_N = -\nabla f(x)^T \leftrightarrow d_N = -\left[\nabla^2 f(x)\right]^{-1} \nabla f(x)^T$$

la matrice $\nabla^2 f(x)$ est carrée et symétrique puisque

$$\begin{aligned}\left[\nabla^2 f(x)\right]_{ij} &= \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \\ &= \frac{\partial^2 f(x)}{\partial x_j \partial x_i} \\ &= \left[\nabla^2 f(x)\right]_{ji}\end{aligned}$$

Algorithm 2 Direction de Newton modifiée

Préalables :
- $\nabla f(x) \in \mathbb{R}^n$ le gradient en x
- $\nabla^2 f(x) \in \mathbb{R}^{n \times n}$ le hessien en x

Variable en sortie : r

$g \leftarrow \nabla f(x)^T$
$H \leftarrow \nabla^2 f(x)$
$A, P \leftarrow$ décomposition de H {P étant la matrice de permutation}
$A' \leftarrow$ modification de la diagonale de A
$d_N \leftarrow$ résolution(A', P, g) {Résolution du système $A''x = g$ où A'' est la matrice A' avec les bonnes permutations}
$r \leftarrow (d_N, A', P)$ {On retourne la direction mais aussi la décomposition et la permutation que l'on pourra réutiliser par la suite}

L'algorithme 2 résume les étapes du calcul de la direction de Newton modifiée. La matrice $\nabla^2 f(x)$ est factorisée pour pouvoir réutiliser la décomposition. P est un vecteur qui contient les informations sur les permutations à faire et la matrice A est modifiée pour obtenir une direction de descente lorsque la matrice n'est pas définie positive (A.1). Pour chaque calcul de la direction en x les opérations sont

— Calcul du hessien $\nabla^2 f(x)$
— Factorisation $LDL^T = PAP^T$

— Changement de la diagonale $D' = D + \tilde{D}$

— Résolution de systèmes triangulaires $Lx = b$

Ce qui a un comportement en $\frac{n^3}{3} + n^2$ et la convergence est quadratique.

1.5.2 Chebychev

Pour calculer la direction de Chebychev d_C, on va réutiliser le calcul de la direction de Newton qui apparaît plusieurs fois dans l'équation (1.6).

Algorithm 3 Direction de Chebychev

Préalables :

- $\nabla f(x) \in \mathbb{R}^n$ le gradient en x
- $\nabla^2 f(x) \in \mathbb{R}^{n \times n}$ le hessien en x
- La fonction $g \in \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n : u,\ v \mapsto \nabla^3 f(x) \cdot u \cdot v$ qui fait le calcul en fonction de deux vecteurs quelconques u et v
- $(d_N, A', P) \leftarrow$ Direction de Newton

Variable en sortie : d_C

$w \leftarrow \nabla^3 f(x) \cdot d_N \cdot d_N = g(d_N, d_N)$

$d \leftarrow$ résolution$(A',\ P,\ w)$

$d_C \leftarrow d_N - \frac{1}{2} d$

$w = \nabla^2(\nabla f(x) \cdot d_n) \cdot d_n$ contient un tenseur $\times$ vecteur$\times$ vecteur donc un vecteur.

Pour chaque calcul de la direction de Chebychev :

— Calcul de la direction de Newton : l'algorithme réutilise les valeurs calculées dans l'algorithme de Newton.

— $\nabla^3 f(x) \cdot d \cdot d$

— Résolution de systèmes triangulaires $Lx = b$

La complexité de la direction de Chebychev est de l'ordre de $\frac{n^3}{3} + (2 + C)n^2$ et la méthode a une convergence cubique.

1.5.3 Halley

Pour calculer la direction de Halley d_H, on va réutiliser le calcul de la direction de Chebychev et de Newton.

Algorithm 4 Direction de Halley

Préalables :
- $\nabla f(x) \in \mathbb{R}^n$ le gradient en x
- $\nabla^2 f(x) \in \mathbb{R}^{n \times n}$ le hessien en x
- La fonction $g \in \mathbb{R}^n \to \mathbb{R}^{n \times n} : u \mapsto \nabla^3 f(x) \cdot u$ qui effectue le produit scalaire du tenseur $\nabla^3 f(x)$ et d'un vecteur
- $(d_N, A', P) \leftarrow$ Direction de Newton
- $d_C \leftarrow$ Direction de Chebychev

Variable en sortie : d_C

$A \leftarrow \nabla^3 f(x) \cdot d_N$

$M \leftarrow \frac{1}{2} A + \nabla^2 f(x)$

$d_H \leftarrow$ résolution$(M d_H = \nabla f(x)^T)$

— Calcul de la direction de Chebychev et donc celle de Newton

— $\nabla^3 f(x) \cdot v$

— Décomposition de Cholesky modifiée

— Changement de la diagonale

— Résolution de systèmes triangulaires $Lx = b$

Soit un coût de $\frac{2}{3}n^3 + (2 + C + D)n^2$ où C et D sont des constantes dépendantes de l'efficacité du logiciel de DA. Pour rappel, cette méthode a une convergence cubique. Pour bien comprendre la difficulté, pour effectuer cette méthode, il faut être capable de calculer $\nabla^3 f(x) \cdot d$ pour une certaine direction d.

1.5.4 Extrapolation d'ordre trois

Les opérations requises sont :

— Calcul de la direction de Chebychev et donc celle de Newton

— $\nabla^3 f(x) \cdot d \cdot d$

— $\nabla^4 f(x) \cdot d \cdot d \cdot d$

— Deux résolutions de systèmes triangulaires $Lx = b$

Le coût total est de l'ordre de $\frac{1}{3}n^3 + (3 + C + E)n^2$ où C et E vont dépendre de l'outil de DA et avec une convergence quartique.

Algorithm 5 Direction d'extrapolation d'odre trois

Préalables :

- $(d_N, A', P) \leftarrow$ Direction de Newton
- $d_C \leftarrow$ Direction de Cheychev
- La fonction $g : u,\ v \mapsto \nabla^3 f(x) \cdot u \cdot v$ qui fait le calcul en fonction de deux vecteurs quelconque u et v
- La fonction $h : u,\ v,\ w \mapsto \nabla^4 f(x) \cdot u \cdot v \cdot w$

Variable en sortie : d_3

$d \leftarrow 2d_C - d_N$
$v \leftarrow \nabla^3 f(x).d.d_N$
$w \leftarrow$ résolution$(A',\ P,\ v)$
$z \leftarrow \nabla^4 f(x).d_N^3 = h(d_N, d_N, d_N)$
$t \leftarrow$ résolution$(A',\ P,\ z)$
$d_3 \leftarrow d_N - \frac{1}{2}w - \frac{1}{6}t$

1.5.5 Résumé

Résumons l'ordre de convergence et la complexité des calculs sous forme d'un tableau. Les résolutions des systèmes linéaires et la modification de la diagonale sont négligés par rapport aux autres calculs.

Méthode	Ordre du coût par rapport à $\#(f)$	Convergence
Newton modifié	$\frac{1}{3}n^3 + n^2$	quadratique
Chebychev	$\frac{1}{3}n^3 + (2 + C)n^2$	cubique
Halley	$\frac{2}{3}n^3 + (2 + C + D)n^2$	cubique
Extrapolation d'ordre 3	$\frac{1}{3}n^3 + (2 + C + E)n^2$	quartique

Ainsi, on peut observer que pour le même ordre de complexité, la convergence de l'algorithme de Chebychev est meilleure que celle de Newton. Si nous arrivons à atteindre les bornes théoriques des calculs, la méthode de Chebychev devrait prendre moins de temps d'exécution surtout pour des dimensions plus élevées pour atténuer la constante C. De plus, il va être intéressant de comparer les méthodes classiques avec l'extrapolation d'ordre trois puisqu'elle a une meilleure convergence.

1.6 Tests d'algorithmes d'optimisation

Comme beaucoup de tests en optimisation furent insuffisants et pas toujours révélateurs, Moré, Garbow et Hillstrom, voir référence [**?**], ont créé une banque de fonctions, voir le tableau 1.1, dans le but de tester des algorithmes d'optimisation sans contraintes. Nous savons que le point initial a une importance primordiale dans l'algorithme, pour cette raison, ils ont choisi de ne pas toujours le placer dans un voisinage de la solution. Cette librairie va aussi me servir de référence pour pouvoir évaluer si les gradients et hessiens calculés par l'outil développé à partir de *Tapenade* 4.2 sont exacts. En effet, pour chaque fonction, la libraire a été traduite dans le langage scilab et il est possible de calculer les gradients et hessiens des fonctions.

1.6.1 Les routines en Fortran ; propriétés des fonctions

Pour $f_i : \mathbb{R}^n \to \mathbb{R}$ pour $i = 1, \cdots, n$, on cherche à résoudre

$$\min \left\{ \sum_{i=1}^{m} f_i^2(x) : x \in \mathbb{R}^n \right\}$$

Chaque routine fournit le vecteur $f(x)$, le scalaire $f(x)^T f(x)$, la jacobienne de f, et le gradient qui est calculé par l'opération $\nabla f(x) = f(x)^T \nabla f(x)$.

L'entête des fonctions est toujours le même, ce qui va permettre d'automatiser la différentiation.

```
subroutine getfun( x, n, f, m, ftf, fj, lfj, g, mode)
```

`n` : dimension de x,
`x(n)` : vecteur de variables
`f(m)` : vecteur résultat
`ftf` : valeur de la fonction objectif qui vaut la somme des carrés de `f`
`m` : dimension de f,
`fj(m,n)` : matrice jacobienne de f
`g(n)` : contient le produit de la matrice transposée `fj` et du vecteur `f` évalué en x, `g` est la moitié du gradient de la somme des carrés de `f`

`mode` : permet d'initialiser ou de choisir les quantités à calculer

On veut obtenir la dérivée de `ftf` par rapport à `x`. Dans toutes les fonctions, on aura le même cas de figure.

Voir le tableau 1.1 pour la liste des fonctions. Les 19 premiers problèmes ont des variables de taille fixe. Les nombres entre parenthèses sont les variables de dimension modifiable.

1.7 Précision des objectifs

Maintenant que nous avons présenté les algorithmes, nous pouvons mieux préciser les objectifs du présent travail. Le but est d'adapter une libraire sous scilab et d'être capable d'une part de fournir les dérivées des fonctions qui vérifient les complexités exposées en 1.5 et d'autre part résoudre les systèmes linéaires. La difficulté des méthodes réside dans l'ensemble des opérations critiques :

- $\nabla f(x)$
- $\nabla f(x) \cdot u$
- $\nabla^2 f(x)$
- $\nabla^2 f(x) \cdot u$
- $\nabla^3 f(x) \cdot u$
- $\nabla^3 f(x) \cdot u \cdot v$
- $\nabla^4 f(x) \cdot u \cdot v \cdot w$
- décomposition de Cholesky $LDL^T = P(A + E)P^T$
- résolution du système $LDL^T x = b$

Nous voulons exploiter au mieux l'utilisation de la DA afin d'obtenir des temps de calcul raisonnable pour les dérivées. Nous allons ainsi présenter la résolution des systèmes linéaire requis par les équations et par conséquent la décomposition de Cholesky modifiée et la résolution des systèmes triangulaires. Bien entendu, il nous faudra une efficacité assuré pour que ces calculs ne soient pas un frein à ceux de la DA. Puis nous analyserons les processus de la différentiation automatique dans l'obtention des dérivées. Nous verrons ainsi qu'il existe deux modes bien distincts pour les calculs et différentes techniques d'implémentation. Puis, nous détaillerons les choix au ni-

veau des outils et des librairies pour les opérations critiques et leurs tests. Enfin, nous observerons les résultats obtenus pour les temps d'exécution et l'efficacité des méthodes.

tableau 1.1 – Liste des fonctions de la librairie MGH, les variables entre parenthèses sont modifiables.

No	n	m	Nom
1.	2	2	Rosenbrock
2.	2	2	Freudenstein and Roth
3.	2	2	Powell Badly Scaled
4.	2	3	Brown Badly Scaled
5.	2	3	Beale
6.	2	10	Jennrich and Sampson
7.	3	3	Helical Valley
8.	3	15	Bard
9.	3	15	Gaussian
10.	3	16	Meyer
11.	3	10	Gulf Research and Development
12.	3	10	Box 3-Dimensional
13.	4	4	Powell Singular
14.	4	6	Wood
15.	4	11	Kowalik and Osborne
16.	4	20	Brown and Dennis
17.	5	33	Osborne 1
18.	6	13	Biggs EXP6
19.	11	65	Osborne 2
20.	(20)	31	Watson
21.	(10)	(10)	Extended Rosenbrock
22.	(10)	(10)	Extended Powell Singular
23.	(4)	(5)	Penalty I
24.	(4)	(8)	Penalty II
25.	(10)	(12)	Variably Dimensioned
26.	(10)	(10)	Trigonometric
27.	(10)	(10)	Brown Almost Linear
28.	(10)	(10)	Discrete Boundary Value
29.	(10)	(10)	Discrete Integral Equation
30.	(10)	(10)	Broyden Tridiagonal
31.	(10)	(10)	Broyden Banded
32.	(10)	(20)	Linear — Full Rank
33.	(10)	(20)	Linear — Rank 1
34.	(10)	(20)	Linear — Rank 1 with Zero Columns and Rows
35.	(10)	(10)	Chebyquad

Chapitre 2

Calculs d'algèbre linéaire : inversion du hessien

2.1 Résolution de système linéaire

Comme nous l'avons vu, le calcul des itérés passe par la résolution de systèmes linéaires. Dans l'exemple de Newton, il faut résoudre $\nabla^2 f(x) d_N = \nabla f(x)^T$ où $\nabla^2 f(x^*) \in \mathbb{R}^{n\times n}$ est une matrice symétrique et $\nabla f(x)^T \in \mathbb{R}^n$. Il s'agit donc d'un système linéaire de la forme $Ax = b$ de grande taille. Il n'est pas envisageable d'adopter une résolution du type Cramer, pour que ce soit efficace, nous devons modifier la matrice A, il existe plusieurs décompositions :

2.1.1 Survol des décompositions classiques pour la résolution

Élimination de Gauss-Jordan

Aussi appelé pivot de Gauss, elle s'applique sur une matrice $A \in \mathbb{R}^{n\times n}$ non singulière. La stratégie est de réduire grâce aux opérations élémentaires sur les colonnes de A pour obtenir une matrice triangulaire supérieure. Il y a $n-1$ étapes, premièrement, $A^{(1)} \leftarrow A$ et $b^{(1)} \leftarrow b$ sont initialisés. Au bout de la $k$ième étape, nous avons $A^{(k)}x = b^{(k)}$

$$A^{(k)} = \begin{bmatrix} A_{11}^{(k)} & A_{12}^{(k)} \\ 0 & A_{22}^{(k)} \end{bmatrix}$$

où $A_{11}^{(k)} \in \mathbb{R}^{(k-1)\times(k-1)}$ est une matrice triangulaire supérieure.

L'élimination de Gauss-Jordan a un coût de $\frac{2}{3}n^3$.

Décomposition LU

Pour une matrice $A \in \mathbb{R}^{n\times n}$, cette décomposition fournit deux matrices LU où L est une matrice triangulaire inférieure et U une matrice triangulaire supérieure. Il existe une unique décomposition si et seulement si $A_k = A(1:k, 1:k)$ est non singulière pour $k = 1 : n-1$, sinon elle existe mais n'est pas unique.

La décomposition LU a un coût de l'ordre de $\frac{2}{3}n^3$.

Décomposition de Cholesky

Cette décomposition, dûe au français André-Louis Cholesky (1875-1918 alors qu'il était commandant en chef) permet de résoudre de manière efficace des

systèmes d'équation linéaire de la forme $Ax = b$ lorsque A est une matrice définie positive.

Théorème 2.1.1 *Si A est une matrice réelle symétrique, définie positive, alors il existe une unique matrice L triangulaire inférieure et inversible, telle que*

$$A = LL^T$$

Algorithm 6 Factorisation de Cholesky

Préalables :
- Soit $A \in \mathbb{R}^{n\times n}$ une matrice symétrique définie positive

En sortie :
- calcule R où $A = R^T R$ et $R = (r_{ij})_{1\leq i,j\leq n}$

for $j = 1 : n$ **do**
  for $i = 1 : n$ **do**
    $r_{ij} \leftarrow (a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj})/r_{ii}$
  end for
  $r_{jj} = (a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2)^{1/2}$
end for

Pour résoudre le système, $Ax = LL^T x = b$, on commence par résoudre $Ly = b$ puis $L^T x = y$. Le nombre d'opérations requis pour cette décomposition est de l'ordre de $\frac{1}{3}n^3$. Il s'agit de la méthode la plus efficace donc celle que l'on devrait utiliser, cependant lorsque la méthode de Newton est appliquée, la matrice hessienne n'est a priori pas définie positive. C'est pour cette raison que l'algorithme de Cholesky a été modifié. La matrice va être corrigée pour obtenir une décomposition définie positive et plutôt bien conditionnée.

2.1.2 Décomposition de Cholesky Modifiée

Soit une matrice A, symétrique mais pas nécessairement définie positive. L'algorithme de Cholesky modifiée calcule la décomposition $P(A+E)P^T = LDL^T$ où P est une matrice de permutation, E est une perturbation pour rendre la matrice $A + E$ définie positive, D est une matrice diagonale et L une matrice triangulaire inférieure. La norme de E devrait être petite et $A + E$ bien conditionnée. Cette technique est

largement utilisée en optimisation comme dans notre cas ou bien pour calculer des pré-conditionneurs définis positifs. Comme le soulignent Cheng et Higham dans [**?**], les objectifs de l'algorithme de Cholesky modifié peuvent être déclarés comme suit :

O1 Si A est "suffisamment définie positive", alors E devrait être égale à zéro.

O2 Si A est indéfinie, $\|E\|$ ne devrait pas être plus grand que

$$\min\{\|\Delta A\| : A + \Delta A \text{ est définie positive } \}$$

pour une norme appropriée.

O3 La matrice $A + E$ devrait être raisonnablement bien conditionnée.

O4 Le coût de l'algorithme devrait être le même que le coût de la décomposition standard de Cholesky pour l'ordre le plus élevé.

2.2 Utilisation de la décomposition de Cholesky modifiée

Il existe plusieurs versions de la décomposition de Cholesky modifiée en Scilab mais pas de version efficace. Cela est lié intrinsèquement à *Scilab* qui est un langage de haut niveau et n'est pas performant pour effectuer du code impératif sur des grandes dimensions comparativement au C ou Fortran. Par conséquent, nous avons choisi deux routines appartenant à Lapack[1], une libraire sur les systèmes linéaires écrite en fortran. La première permet la décomposition de Cholesky modifiée et la deuxième la résolution du système avec cette décomposition, nommée `dsytrf` et `dsytrs` respectivement. Cette décomposition de Cholesky modifiée utilise la méthode de pivotement de Bunch-Kaufman (BK) [**?**].

Soit une matrice $A \in \mathbb{R}^{n\times n}$ non nulle, la factorisation fournit

$$P(A + E)P^T = L(D + F)L^T.$$

F est choisie pour que $D + F$ et ainsi $A + E$ soient définies positives. Cette technique a été proposée par Moré et Sorensen [**?**]. L'idée consiste à trouver une permutation

1. http://www.netlib.org/lapack/

Π et un entier $s = 1$ ou 2 tel que

$$\Pi A \Pi^T = \begin{bmatrix} E & C^T \\ C & B \end{bmatrix}$$

avec $E \in \mathbb{R}^{s \times s}$ non singulière et $B \in \mathbb{R}^{(n-s)\times(n-s)}$. En choisissant correctement Π, nous avons la factorisation :

$$\Pi A \Pi^T = \begin{bmatrix} I_s & 0 \\ CE^{-1} & I_{n-s} \end{bmatrix} \begin{bmatrix} E & 0 \\ 0 & B - CE^{-1}C^T \end{bmatrix} \begin{bmatrix} I_s & E^{-1}C^T \\ 0 & I_{n-s} \end{bmatrix}$$

Le procédé est répété récursivement sur la matrice $S = B - CE^{-1}C^T$ de taille $(n-s) \times (n-s)$. On remarque ainsi qu'au lieu d'utiliser un pivot de taille 1×1, on peut utiliser une matrice 2×2.

Selon Cheng et Higham [**?**], l'algorithme de BK, que celui de Schnabel et Heskow [**?**], a un coût identique à la décomposition de Cholesky standard relativement aux termes d'ordre les plus élevés, cependant les objectifs O1 et O3 de la partie 2.1.2 sont difficilement satisfaits. Il se peut que la matrice $A + E$ soit mal conditionnée car $\|L\|_\infty$ n'est pas bornée et par conséquent les valeurs propres de D peuvent largement différer de A. Les auteurs proposent ainsi une autre version de l'algorithme de BK, permettant de satisfaire les conditions mais nous considérerons que les routines de lapack permettrons d'obtenir des directions satisfaisantes d'autant plus que nous utiliserons une recherche linéaire ce qui injectera un niveau de contrôle en plus.

Afin d'obtenir une matrice définie positive, nous profitons de cette décomposition pour changer les éléments diagonaux. L'avantage, c'est que l'on a plus besoin de faire ces changements sur une matrice n par n mais seulement 2×2 ou 1×1.

2.3 Résolution des systèmes linéaires

Pour commencer, il est à noter que la résolution des systèmes triangulaires dans *Scilab* n'est généralement pas efficace. Il s'avère que la détection du système triangulaire n'est pas faite systématiquement. La figure 2.1 révèle que sur les quatre tests de résolution $Lx = e$, $L'x = e$, $U'x = e$, $Ux = e$ où L et U proviennent de la décomposition LU réalisé par *Scilab* et $e = (1)_{1 \le i \le n}$ un vecteur colonne, une seule est

Algorithm 7 Changement de la diagonale

Préalables :

- $\epsilon > 0$
- $\tilde{D}$ la matrice diagonale par bloc

Sortie

- D la matrice diagonale par bloc avec pour valeur propre minimale $\lambda_{min} \geq \epsilon$

Pour chaque bloc de la diagonale $\tilde{D_k}$,
if $\tilde{D_k}$ est de dimension 1×1 **then**
  $D_k \leftarrow \max(\tilde{D_k}, \epsilon)$
else
  $[Z, W] = spec(\tilde{D_k})$ {Il s'agit de la diagonalisation de $\tilde{D_k}$: $ZWZ^T = \tilde{D_k}$ avec W matrice diagonale}
  $D_k \leftarrow Z * diag(max(diag(W), \epsilon)) * Z^T$
end if

performante.

C'est pour cette raison que nous avons préféré choisir une décomposition écrite en fortran pour nos calculs.

La résolution des systèmes $Ax = b$ s'effectue en trois temps. Tout d'abord, A qui est par exemple la hessienne de notre fonction, est décomposée en

$$L\tilde{D}L^T = P^T(A + E)P$$

où D est une matrice diagonale par bloc soit de taille 1×1, soit 2×2. La factorisation a un coût de $\frac{1}{3}n^3$. Ensuite, on modifie chacun de ces blocs pour le rendre défini positif : $D \leftarrow \tilde{D} + \Delta\tilde{D}$. Cette opération est de l'ordre de n mais elle est effectuée avec *Scilab* Enfin, on résout le système $LDL^T x = b$ qui est une opération en n^2.

2.3.1 Temps de calcul de l'ensemble de la résolution

Comme le montre la figure 2.3, la factorisation modifiée est l'opération qui prend le plus de temps, elle a un comportement en n^3. La résolution du système linéaire est plus rapide que le changement de la diagonale parce qu'elle est exécutée en fortran.

figure 2.1 – Résolution d'un système triangulaire par *Scilab* avec une décomposition LU, sur les quatre versions, qu'une seule n'est efficace.

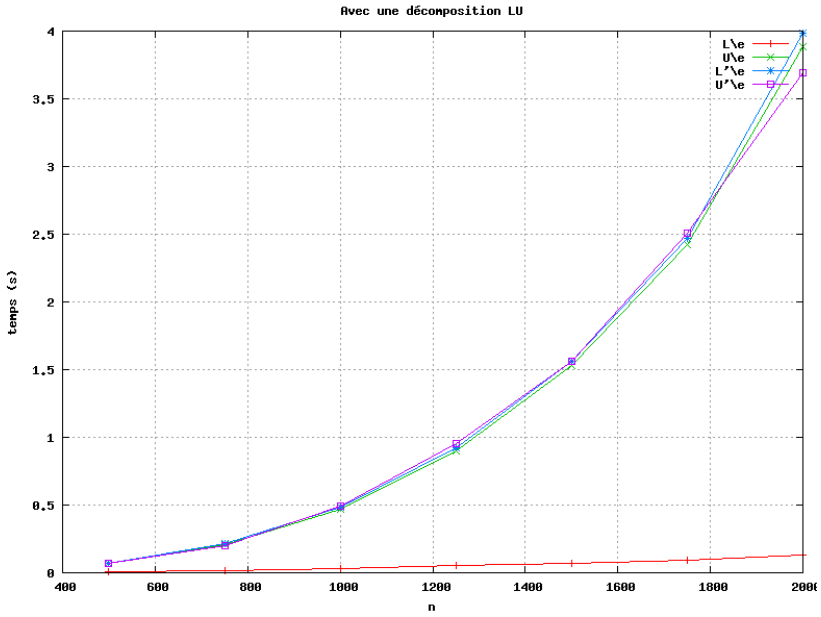


figure 2.2 – Résolution d'un système triangulaire par *Scilab* avec une décomposition de Cholesky

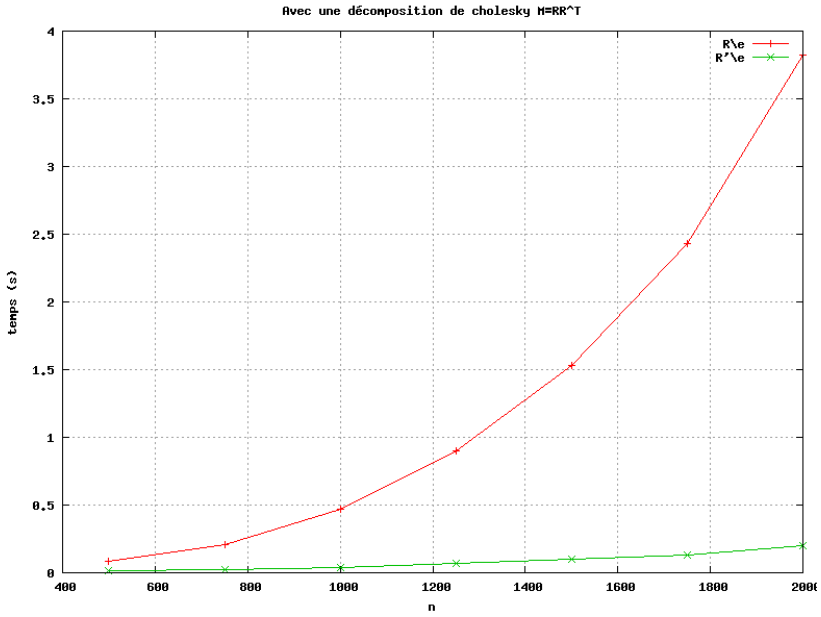
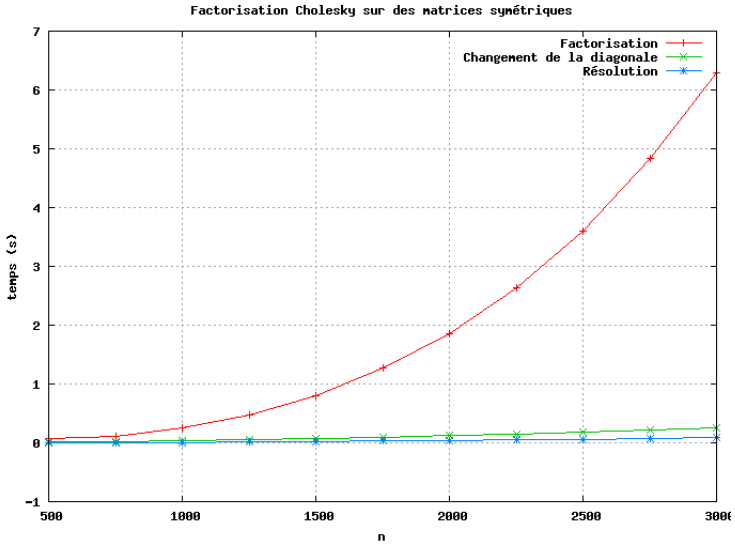


figure 2.3 – Résolution du système $Ax = b$ avec la factorisation de Cholesky modifiée



2.3.2 Conclusion

Nous venons d'obtenir une interface reliant *Scilab* à *Fortran* nous permettant d'obtenir une résolution de systèmes linéaire de manière efficace et ce même si la matrice n'est pas définie positive. À présent, nous allons nous interesser à la différentition automatique, les problèmatiques qu'elle soulève et différents modes qui existent. Nous verrons plus tard si les outils sont suffisamment performants pour valider les bornes de complexité définies dans le chapitre 1.

n	Décomposition $A \leftarrow LDL^T$	Arrangement $D \leftarrow \tilde{D} + \Delta\tilde{D}$	Résolution $Lu = b$, $\tilde{D}v = u$, $L^T x = v$	$A\backslash b$ avec Scilab
500	0.037000	0.011000	0.002000	0.090000
750	0.122000	0.026000	0.005000	0.220000
1000	0.254000	0.038000	0.010000	0.486000
1250	0.475000	0.055000	0.015000	0.944000
1500	0.809000	0.075000	0.022000	1.607000
1750	1.278000	0.097000	0.030000	2.543000
2000	1.863000	0.123000	0.038000	3.732000
2250	2.636000	0.149000	0.047000	5.413000
2500	3.602000	0.179000	0.057000	7.402000
2750	4.830000	0.215000	0.070000	10.116000
3000	6.320000	0.251000	0.084000	13.327000

tableau 2.1 – Temps de calcul en seconde pour chaque étape de la résolution du système $Ax = b$, bien que la modification de la diagonale soit $\mathcal{O}(n)$, elle est moins efficace que la résolution car elle est codée en *Scilab*.

Chapitre 3

Obtention des dérivées : Différentiation automatique

3.1 Introduction

Comme le souligne Corliss [**?**], la DA est une technique introduite vers 1962 avec le mode direct mais n'a pas réussi à s'imposer. Ce n'est que plus tard, en 1982, grâce à l'amélioration des techniques de programmations et l'introduction du mode inverse que la DA a connu plus de succès. Griewank est à l'origine de grands progrès et il s'en ait suivi une forte augmentation du nombre d'outils, de techniques et d'applications de la différentiation automatique. L'implantation de la DA se divise en deux formes : la surcharge des opérateurs et la transformation du code. La surcharge des opérateurs consitent à étandre la sémantique des opérations, c'est-à-dire que chaque variable est surchargée par sa dérivée et les opérations s'opèrent ces les deux quantités. La tranformation de code, ne fait que retourner le code de la dérivées. Le code est analysé puis transformer, en fait les lignes correspondant au calcul de la différentiée sont rajoutées. Ce code est parfois écrit à la main mais la DA a suffisamment fait de progrès pour générer un code en quelques minutes (pour les gros programmes) et d'une qualité comparable d'après l'article [**?**].

Nous verrons ainsi quelles sont les points forts et faibles de ces deux procédés.

Ainsi, comme indiqué dans [**?**], le but de la DA est de calculer la dérivée d'une fonction spécifiée par un programme, un algorithme. Cette méthode de calcul s'oppose à deux autres bien connues : la différentiation symbolique et la différentiation par différences finies. La première, que l'on peut retrouver dans *Maple*, utilise l'expression de la fonction pour déterminer sa dérivée. Cette technique est très vite limitée d'une part lorsque l'on a des expressions un peu complexes et d'autre part parce qu'il faut l'expression de la fonction.
Par exemple sur *Maple*, si on veut obtenir :

$$\frac{\partial^3((x^2+y^2)*(\ln(x)-\ln(y)))}{\partial y \partial^2 x}$$

```
diff((x^2+y^2)*(ln(x)-ln(y)), y,x$2);
```
$\rightarrow -\frac{2y}{x^2} - \frac{2}{y}$

Griewank illustre un exemple de différentiation symbolique dans [**?**], avec le logiciel

Macsyma. La fonction qui permet de calculer l'energie d'Helmholtz est fournie au logiciel. Le résultat correspond à la figure 3.1. En plus du fait que le code généré est difficilement comprehensible pour un être humain, le code a du être modifié à cause du maximum de 19 lignes consécutives en Fortran. Ainsi, il est extrêment difficile de pouvoir maintenir une telle structure de code.

La plupart du temps, il faut différentier un code constitué de boucles et de conditions qui est difficilement exprimable par une expression mathématique. La différentiation numérique ou par différences finies s'appuie sur l'expression théorique de la dérivée :

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Dans le cas à plusieurs dimensions :

$$\lim_{\varepsilon \to 0} \frac{P(X + \varepsilon \cdot dX) - P(X)}{\varepsilon} = \nabla P(X) \cdot dX$$

Cependant, il s'agit d'un problème hasardeux à cause de la discrétisation imposée par ordinateur. h doit être choisi dans l'ordre de grandeur de la racine de la précision machine : si h est trop proche de 0 la différence va être mal approchée ; l'écart entre $f(x+h)$ et $f(x)$ étant trop faible et si h est trop grand, on s'éloigne de la véritable valeur de la dérivée. Ainsi, nous allons voir que la différentiation automatique pallie à ces deux inconvénients majeurs.

3.2 Principes de la différentiation automatique

Contrairement à la différentiation symbolique, l'objectif de la différentiation automatique n'est pas de concevoir l'expression de la dérivée mais uniquement le programme qui la calcule. La DA calcule la dérivée de manière analytique, c'est-à-dire qu'elle obtient le calcul exact de la dérivée. Ainsi, il n'y a pas d'erreurs d'approximations. À chaque fois qu'apparaît une variable dans le programme source, le programme différentié va calculer une variable additionnelle de la même forme : sa différentiée. Il est à noter que la DA ne vise pas à fournir l'expression mathématique de la dérivée, puisqu'elle ne fournit que du code permettant son évaluation. Il existe deux manières

figure 3.1 – Code produit par différentiation symbolique à partir du logiciel *Macsyma*

```
RUTU=DSQRT(2.D0)
    F=0.0013564*(-(x(5)+x(4)+x(3)+x(2)+x(1))*DLOG(-b(5)*x(5)-b(4)*x(
   1  4)-b(3)*x(3)-b(2)*x(2)-b(1)*x(1)+1)+x(5)*DLOG(x(5))+x(4)*DLOG
   2  (x(4))+x(3)*DLOG(x(3))+x(2)*DLOG(x(2))+x(1)*DLOG(x(1)))-(x(5
   3  )*(x(5)*a(5,5)+x(4)*a(5,4)+x(3)*a(5,3)+x(2)*a(5,2)+x(1)*a(
   4  5,1))+x(4)*(a(4,5)*x(5)+x(4)*a(4,4)+x(3)*a(4,3)+x(2)*a(4,2
   5  )+x(1)*a(4,1))+x(3)*(a(3,5)*x(5)+a(3,4)*x(4)+x(3)*a(3,3)+x
   6  (2)*a(3,2)+x(1)*a(3,1))+x(2)*(a(2,5)*x(5)+a(2,4)*x(4)+a(2,
   7  3)*x(3)+x(2)*a(2,2)+x(1)*a(2,1))+x(1)*(a(1,5)*x(5)+a(1,4)*
   8  x(4)+a(1,3)*x(3)+a(1,2)*x(2)+x(1)*a(1,1)))*DLOG(((RUTU+1
   9  )*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1)/(
   :  (1-RUTU)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*
   ;  x(1))+1))/(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x( 1))
    g(1)=b(1)*(x(5)*(x(5)*a(5,5)+x(4)*a(5,4)+x(3)*a(5,3)+x(2)*a(5,2)
   1  +x1(1)*a(5,1))+x(4)*(a(4,5)*x(5)+x(4)*a(4,4)+x(3)*a(4,3)+x(2)
   2  *a(4,2)+x(1)*a(4,1))+x(3)*(a(3,5)*x(5)+a(3,4)*x(4)+x(3)
   3  *a(3,3)+x(2)*a(3,2)+x(1)*a(3,1))+x(2)*(a(2,5)*x(5)+a(2,4)*x(4
   4  )+a(2,3)*x(3)+x(2)*a(2,2)+x(1)*a(2,1))+x(1)*(a(1,5)*x(5)+a
   5  (1,4)*x(4)+a(1,3)*x(3)+a(1,2)*x(2)+x(1)*a(1,1)))*DLOG(((RUTU
   6  +1)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1)
   7  )+1)/((1-RUTU)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)
   8  +b(1)*x(1))+1))/(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b
   9  (1)*x(1))**2-(x(5)*a(5,1)+a(1,5)*x(5)+x(4)*a(4,1)+a(1,4)*x
   7  (4)+x(3)*a(3,1)+a(1,3)*x(3)+x(2)*a(2,1)+a(1,2)*x(2)+2*x(1)
   7  *a(1,1))*DLOG(((RUTU+1)*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b
   7  (2)*x(2)+b(1)*x(1))+1)/((1-RUTU)*(b(5)*x(5)+b(4)*x(4)+b
   1  (3)*x(3)+b(2)*x(2)+b(1)*x(1))+1))/(b(5)*x(5)+b(4)*x(4)+b(3
   6  )*x(3)+b(2)*x(2)+b(1)*x(1))
    g(1)=g(1)+0.0013625*(-DLOG(-b(5)*x(5)-b(4
   7  )*x(4)-b(3)*x(3)-b(2)*x(2)-b(1)*x(1)+1)+DLOG(x(1))+b(1)*(x(
   7  5)+x(4)+x(3)+x(2)+x(1))/(-b(5)*x(5)-b(4)*x(4)-b(3)*x(3)-b(
   7  2)*x(2)-b(1)*x(1)+1)+1)-((1-RUTU)*(b(5)*x(5)+b(4)*x(4)+
   7  b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1)*((RUTU+1)*b(1)/((1-RUTU
   7  )*(b(5)*x(5)+b(4)*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))
   7  +1)-(1-RUTU)*b(1)*((RUTU+1)*(b(5)*x(5)+b(4)*x(4)+b(3
   7  )*x(3)+b(2)*x(2)+b(1)*x(1))+1)/((1-RUTU)*(b(5)*x(5)+b(4
   7  )*x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1)**2)*(x(5)*(x(5)*a
   7  (5,5)+x(4)*a(5,4)+x(3)*a(5,3)+x(2)*a(5,2)+x(1)*a(5,1))+x(4
   7  )*(a(4,5)*x(5)+x(4)*a(4,4)+x(3)*a(4,3)+x(2)*a(4,2)+x(1)*a(
   7  4,1))+x(3)*(a(3,5)*x(5)+a(3,4)*x(4)+x(3)*a(3,3)+x(2)*a(3,2
   7  )+x(1)*a(3,1))+x(2)*(a(2,5)*x(5)+a(2,4)*x(4)+a(2,3)*x(3)+x
   7  (2)*a(2,2)+x(1)*a(2,1))+x(1)*(a(1,5)*x(5)+a(1,4)*x(4)+a(1,
   7  3)*x(3)+a(1,2)*x(2)+x(1)*a(1,1)))/((b(5)*x(5)+b(4)*x(4)+b(
   7  3)*x(3)+b(2)*x(2)+b(1)*x(1))*((RUTU+1)*(b(5)*x(5)+b(4)*
   7  x(4)+b(3)*x(3)+b(2)*x(2)+b(1)*x(1))+1))
```

d'utiliser la DA, soit le code est transformé pour obtenir un nouveau programme qui calculera directement la différentiée, soit par surcharge des opérateurs. Nous allons décrire en détail ces différentes approches. Pour la deuxième approche, il s'agit d'ajouter aux fonctions de base (l'addition, cos, log) les opérations de dérivations.

Par exemple, en prenant x, y, z comme variables et V comme vecteur, lors de l'instruction :

$$\mathtt{x} = \mathtt{y} * \mathtt{V}(10) + \mathtt{z}$$

le programme différentié va calculer :

$$\dot{\mathtt{x}} = \dot{\mathtt{y}} * \mathtt{V}(10) + \mathtt{y} * \dot{\mathtt{V}}(10) + \dot{\mathtt{z}}$$

en utilisant les règles de dérivations usuelles sur les fonctions. Il n'y a plus d'approximations, c'est un calcul exact. Le principe est de considérer que chaque programme peut s'écrire comme une séquence d'instructions.

$$I_1; I_2; ...; I_{p-1}; I_p$$

Cette suite peut être identifiée comme une composition de fonctions

$$f = f_p \circ f_{p-1} \circ \cdots \circ f_1$$

par la règle de dérivation sur la composition (A) on obtient :

$$\begin{aligned} f'(X) =& (f'_p \circ f_{p-1} \circ f_{p-2} \circ ... \circ f_1(X)) \\ & .(f'_{p-1} \circ f_{p-2} \circ ... \circ f_1(X)) \\ & \cdots \\ & .f'_1(X) \\ =& f'_p(W_{p-1}).f'_{p-1}(W_{p-2}).\cdots .f'_1(W_0). \end{aligned}$$

En notant $W_0 = X$ et $W_k = f_k(W_{k-1})$. Comme plusieurs données sont traitées, tous les f'_k sont des matrices Jacobiennes de taille relativement grande dans un cas général. Calculer la différentiée revient à calculer les multiplications de ces matrices. Cependant, il n'est pas possible de calculer ce produit avec un coût raisonnable. Par exemple, avec dix variables, si on effectue une quinzaine d'instructions cela revient à faire de l'ordre de 10^4 opérations. La complexité est exponentielle. Dans la plupart des cas, l'application qui utilise $\nabla f(X)$ n'a en réalité que besoin d'une direction de la jacobienne : $\nabla f(X).\dot{X}$ pour une certain vecteur $\dot{X}$. Nous allons voir les deux modes de différentiation, le mode tangent et le mode inverse. Dans le premier mode, les calculs de la fonctions se propagent parallèlement aux dérivées tandis que dans le mode inverse, le calcul s'effectue à rebours en partant de la fin du code.

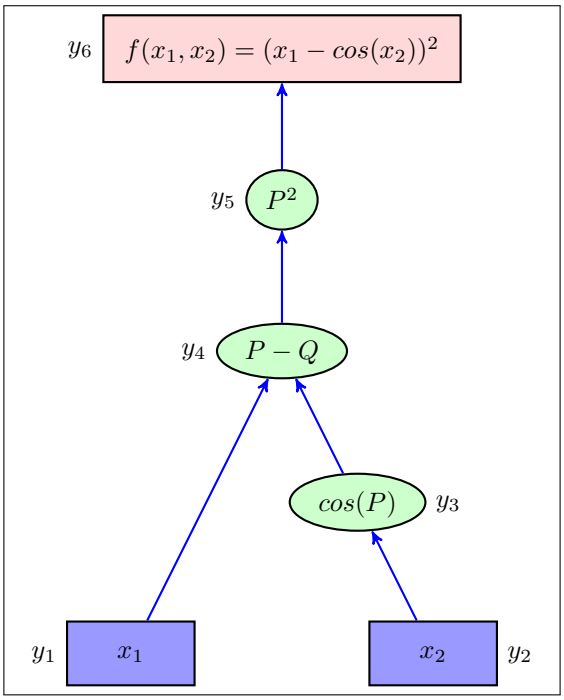
3.2.1 Mode tangent ou mode direct

Dans notre cas, comme par exemple pour la direction de Chebychev, nous avons besoin de calculer $\nabla^3 f(x) \cdot u \cdot v$ et non $\nabla^3 f(x)$. En prenant cela en compte, le calcul va être largement simplifié. À l'ordre un : $\dot{Y} = f'(X).\dot{X}$

$$\dot{Y} = f'_p(W_{p-1}).f'_{p-1}(W_{p-2}).\cdots .f'_1(W_0).\dot{X}.$$

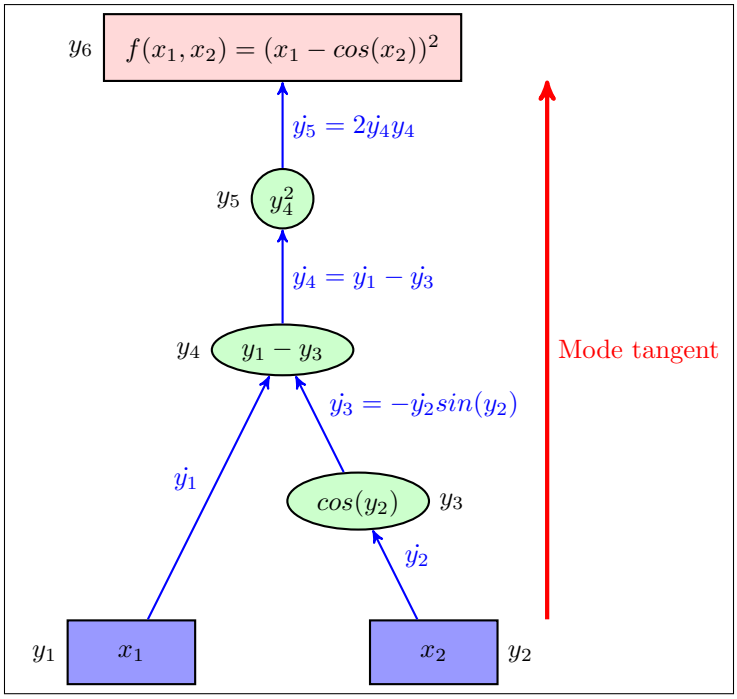
Pour profiter de la multiplication avec le vecteur, le calcul va se faire de droite à gauche afin d'éviter d'avoir des multiplications de Matrice×Matrice (correspond au mode

figure 3.2 – GAO : $f(x_1, x_2) = (x_1 - cos(x_2))^2$ pour évaluer la fonction, le parcours se fait à partir des feuilles de l'arbre jusqu'à la racine.



multi-directionnel pour Tapenande) mais plutôt Matrice×Vecteur. De plus, de cette manière, les appels aux W_i vont se faire dans l'ordre, donc en même temps qu'ils seront calculés. Cette méthode donne une combinaison linéaire des colonnes de la matrice Jacobienne. Voici un exemple illustré pour la fonction $f(x_1, x_2) = (x_1 - cos(x_2))^2$. Dans le Graphe Acyclique Orienté 3.2, le gradient de chaque quantité en partant des feuilles va être propagé.

figure 3.3 – GAO : mode tangent, il suit le même parcours que celui de l'évaluation



y	Valeurs de y	$\dot{y}$	Valeurs de $\dot{y}$	Valeurs vectorielles
y_1	x_1	$\dot{y_1}$	$\dot{x_1}$	$[1 \quad 0]$
y_2	x_2	$\dot{y_2}$	$\dot{x_2}$	$[0 \quad 1]$
y_3	$cos(y_2)$	$\dot{y_3}$	$-\dot{y_2}sin(y_2)$	-$[0 \quad sin(x_2)]$
y_4	$y_1 - y_3$	$\dot{y_4}$	$\dot{y_1} - \dot{y_3}$	$[1 \quad sin(x_2)]$
y_5	y_4^2	$\dot{y_5}$	$2\dot{y_4}y_4$	$2[x_1 - cos(x_2) \quad (x_1 - cos(x_2))sin(x_2)]$

Le programme généré par la DA évalue simultanément la fonction et le gradient. Le nombre de lignes obtenu est environ deux fois celui du programme d'origine puisque chaque affectation est accompagnée du calcul du gradient. En général, comme c'est expliqué dans [**?**], le mode tangent multiplie le nombre d'opérations arithmétiques de n. Chaque quantité x_i est précédée du calcul de ∇x_i de taille n. Dans l'exemple, on peut observer que les quantités propagées ont une dimension équivalente au nombre de composante de l'argument. Ainsi, le coût dû au calcul du gradient est de l'ordre de n fois le coût de l'évaluation de la fonction.

3.2.2 Mode inverse

Comme nous venons de le voir, le mode tangent propage un vecteur de la taille du gradient dans le graphe, par conséquent, le coût est proportionnel à $n\#(f)$. En revanche, le mode inverse ne va que propager un scalaire dans le graphe mais le parcours va se faire dans le sens inverse à celui de l'évaluation. C'est parce que les calculs ne se font que sur un scalaire que le coût du gradient est proportionnel au coût de l'évaluation de la fonction. Ainsi, il va être préférable de choisir le mode inverse au mode tangent.

Le mode inverse va nous permettre d'obtenir une ligne de la Jacobienne c'est-à-dire un gradient par rapport à une composante k.

$$\overline{X} = f'^T(X).\overline{Y}$$

$$\overline{X} = f_1'^T(W_0).f_2'^T(W_1).\cdots.f_p'^T(W_{p-1}).\overline{Y} \tag{3.1}$$

L'idée sous-jacente est l'utilisation des quantités adjointes :

$$y_i^* = \frac{\partial f}{\partial y_i}$$

$$\bar{y_j} = \sum_{i \in I_j} \frac{\partial f}{\partial y_j} \bar{y_i}$$

où tous les $\bar{y_i}$ sont des quantités scalaires

$$I_j = \{i | y_j \text{ intervenant dans } y_i\}.$$

Le parcours du GAO se fait en profondeur, de la racine jusqu'aux feuilles. Contrairement au mode tangent, comme les quantités propagées sont des scalaires, qu'une seule équation n'est impliquée à chaque nœud, au lieu d'en avoir n, la dimension. Comme l'indique la figure 3.4, les quantités $\bar{y_i}$ se propagent à rebours, dans le sens inverse des quantités y_i. C'est le fait que le calcul se propage sur l'ensemble des feuilles qui va permettre de reconstruire le gradient de dimension n, chaque feuille correspondant à une composante. Commençons par observer ce mode sur notre exemple. Cette fois-ci, le parcours n'est plus le même que l'évaluation de la fonction.

y	Valeurs de y
y_1	x_1
y_2	x_2
y_3	$cos(y_2)$
y_4	$y_1 - y_3$
y_5	y_4^2

$\bar{y}$	Valeurs de $\bar{y}$	Valeurs
$\bar{y_5}$	1	1
$\bar{y_4}$	$2y_4$	$2(x_1 - cos(x_2))$
$\bar{y_3}$	$-\bar{y_4}$	$2(cos(x_2) - x_1)$
$\bar{y_2}$	$-\bar{y_3} sin(x_2)$	$2(x_1 - cos(x_2))sin(x_2)$
$\bar{y_1}$	$\bar{y_4}$	$2(x_1 - cos(x_2))$

Dans l'équation 3.1, l'opération doit se faire encore de droite à gauche pour que le calcul soit efficace. Malheureusement, cette fois-ci, nous n'avons pas les appels aux W_i dans le même ordre qu'ils sont calculés ; cela vient du fait que le parcours n'est plus dans le même sens. Dans l'exemple, à la deuxième étape, la quantité y_4 est nécessaire, elle fait intervenir y_1 et y_2 alors que ces états n'ont pas encore été parcourus. Ainsi, il existe deux stratégies pour obtenir les W_i. Soit on recalcule toutes les quantités, soit on les mémorise toutes.

figure 3.4 – GAO : mode inverse, cette fois-ci, l'arbre est parcouru depuis la racine.

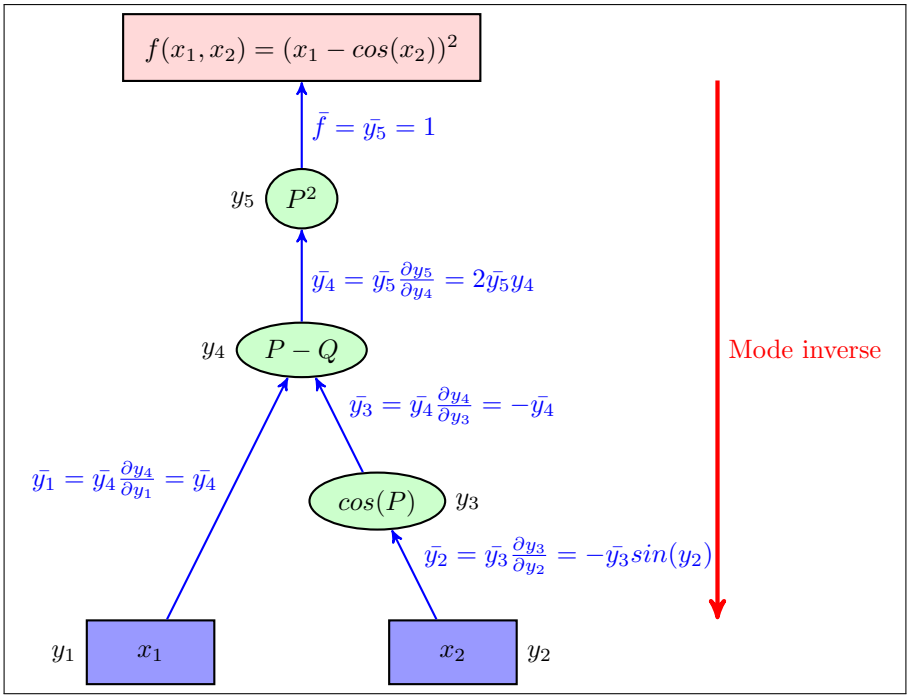


figure 3.5 – Stratégie RA : pour chaque quantité à calculer, on reparcours le graphe pour faire un pas dans l'algorithme inverse. Prend moins de place mais plus de temps.

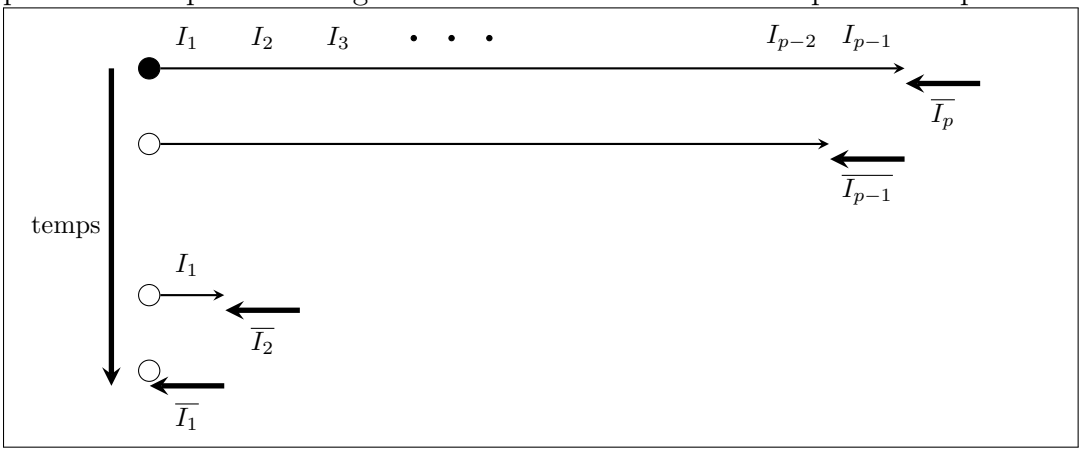
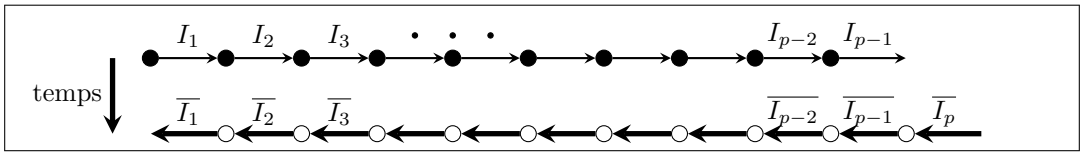


figure 3.6 – Stratégie SA : le graphe des évaluations est parcouru une seule fois pour toutes les mémoriser, l'algorithme inverse n'aura plus qu'à dépiler. Prend moins de temps mais plus de capacité de stockage.



3.2.3 Stratégies de la DA pour le mode inverse

Recompute-All Pour chaque terme $W_p = f_k(W_{p-1})$, on recalcule l'ensemble de la suite W_i à chaque fois. L'opération $W_1 = f'(X)$ va être effectuée p fois. Cette méthode demande plus de temps d'exécution puisque les termes ne sont pas mémorisés, les mêmes calculs sont effectués plusieurs fois. Sur les figures commençant à 3.5 jusqu'à 3.8, les points noirs représentent le stockage de W_k sur la pile d'exécution et les points blancs représentent un dépilement.

Store-All Cette fois-ci, tous les termes vont être calculés et enregistrés une seule fois. Il s'agit d'une méthode qui nécessite plus de mémoire. Le coût en mémoire est linéaire par rapport à p.

figure 3.7 – Checkpoint RA - on effectue des sauvegardes à certains nœuds du GAO et entre chacun de ces nœuds on adopte une stratégie de tout recalculer.

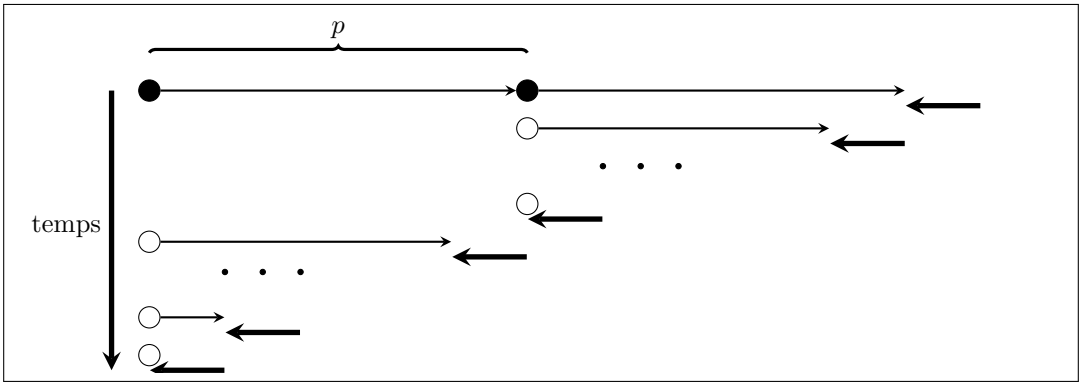
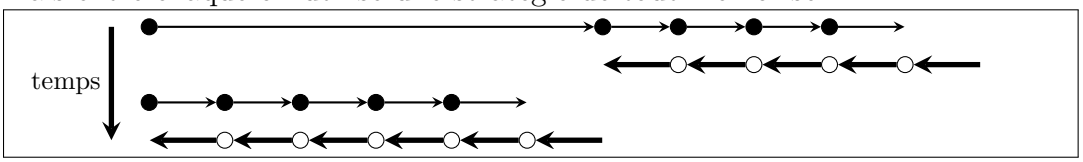


figure 3.8 – Checkpoint SA - là aussi, on sauvegarde les données à certains nœuds mais entre chaque on utilise une stratégie de tout mémoriser.



Dans les deux cas, si le problème a une dimension trop grande, ni la stratégie RA, ni la SA ne pourra être efficace. Une méthode alternative apparaît comme un bon compromis : le *Checkpointing*. L'idée est de décomposer le programme en plusieurs parties, si possible imbriquées et d'effectuer une sauvegarde, un *snapshot*, des quantités entre chaque. Encore peu de travail a été effectué sur la comparaison de l'emplacement de ces *Checkpoints* et cela reste une problème ouvert. Il n'y a pour l'instant pas d'emplacement optimal connu pour un algorithme quelconque. Néanmoins, ils seront évidemment placés à l'extérieur des sous-routines ou des boucles. À partir des ces sauvegardes on peut soit appliquer la méthode RA sur la sous partie du code comme l'illustre la figure 3.7, soit la méthode SA 3.8. C'est la deuxième qui a été retenue par *Tapenade* car la taille de la pile est dans ce cas raisonnable et les exécutions d'empilement et de dépilement sont rapides.

3.3 Implantation de la DA

Deux possibilités s'offrent, soit la surcharge des opérateurs, plus flexible, soit la transformation du code. Dans le premier cas, les opérateurs vont être transformés pour ajouter les opérations de dérivation alors que dans la transformation du code, on ne fait qu'analyser et modifier le code texte qui permet les calculs de la fonction.

3.3.1 La surcharge des opérateurs

Comme l'a fait Karczmarczuk dans l'article [**?**], il est possible de programmer la différentiation automatique par surcharge des opérateurs avec un langage fonctionnel. Voir l'annexe C. Malheureusement, les structures de données sont très rapidement lourdes à gérer, surtout qu'il s'agit de listes paresseuses infinies. Bien qu'il s'avère être un outil simple à manipuler, il n'est pas possible d'obtenir une efficacité acceptable. D'autre part, un outil de surcharge des opérateurs à été effectué pour *Scilab* nommé sciad par Benoit Hamelin. Cependant, il souffre d'un grand overhead et les temps d'exécution sont long lorsque la dimension est plus grande que 5. D'autre part, cette approche consiste à faire l'acquisition du graphe en surchargeant les opérateurs usuels et les fonctions élémentaires, cependant il n'est pas possible de surcharger les structures de contrôle comme les comparaisons et les boucles. Par conséquent, si une condition vient à changer, il faut procéder à une nouvelle aquisition du graphe alors que dans le cas par transformation de code, l'intégralité du programme est transformé, donc cette difficulté n'apparaît pas.

3.3.2 La transformation du code

Ce procédé n'utilise que le code de la fonction pour générer celui de la dérivée. Au lieu de surcharger les opérateurs, le code est analysé pour détecter les variables dépendantes. Ensuite, par un procédé analytique, il applique la dérivation sur les opérations usuelles ; `sin(x)` est transformée en `cos(x)*xd` où `xd`=$\dot{x}$ et il rajoute cette ligne juste avant.

Si on reprend le mode tangent sur notre exemple : comme variable de sortie, on a `f` et comme variable d'entrée `x`. En Fortran, l'exposant se note `**`.

Code original SUBROUTINE F(x, f)	Mode tangent SUBROUTINE F_d(x, xd, f, fd)
f=x(1)-COS(x(2)) f=f**2	fd = xd(1) + xd(2)*SIN(x(2)) f = x(1) - COS(x(2)) fd = 2*f*fd f = f**2

Ainsi, dans le mode tangent, la valeur `fd` renvoie $\nabla F(x).xd$, en *Scilab* `derivative(F,x)*xd`. Ce code, nommé code adjoint, reste très proche du code d'origine et est impératif. Il bénéficiera ainsi d'une exécution efficace.

Le tableau ci-dessous donne à gauche le code du programme de notre fonction et à droite il s'agit de résultat retourné par l'outil que nous utiliserons plus tard. L'outil génère des noms d'incrément évitant des conflits d'où `ii1`.

Code original SUBROUTINE F(x, f)	Mode reverse SUBROUTINE F_b(x, xb, f, fb)
f=x(1)-COS(x(2)) f=f**2	f = x(1) - COS(x(2)) fb = 2*f*fb DO ii1=1,2 xb(ii1) = 0.D0 ENDDO xb(1) = fb xb(2) = SIN(x(2))*fb fb = 0.D0 END

`xb` renvoie la valeur du gradient, en *Scilab*, cela correspond à la ligne de commande `--> xb=derivative(F,x)*fb`.

3.3.3 Discussion

La surcharge des opérateurs est plus souple et plus simple à utiliser. Il suffit en général d'enrichir le type de données pour effectuer la différentiation sur ce nouveau type. La transformation de code source se fait en amont et utilise des concepts issus de la compilation en arbre de syntaxe. Maintenant que nous venons de voir les principes de fonctionnement, il a fallu choisir un outil de différentiation automatique permettant d'implanter efficacement les opérations : ∇f, $\nabla^2 f \cdot v$, $\nabla^2 f$, $\nabla^3 f \cdot u \cdot v$, $\nabla^3 f \cdot u$, $\nabla^4 f \cdot u \cdot v \cdot w$. Malgré le fait qu'il existe actuellement plusieurs outils de DA, le choix n'est pas évident car pour une implémentation efficace, il est préférable d'utiliser un outil par transformation de code et le fait d'obtenir des dérivées supérieures est en général un point fort de la surcharge des opérateurs. De plus, nous avons dû choisir une banque de tests adéquate à notre outil et qui représente suffisamment de cas de figure afin de tester correctement les algorithmes. Nous allons ainsi présenter les temps d'exécution pour l'ensemble des opérations ; gradient, hessien, etc...

Chapitre 4

Les outils utilisés

Un langage très connu de modélisation algébrique en optimisation est AMPL, *A Mathematical Programming Language*, voir le livre [**?**]. Développé par Fourer, Gay et Kernighan, il a été conçu pour résoudre des problèmes complexes de grande dimension. On peut retrouver une liste de solveurs sur le site http://en.wikipedia.org/wiki/Optimization_%28mathematics%29. Comme exemple les plus connus de solveurs externes, on peut citer MINOS, IPOPT, SNOPT, KNITRO. Une des particularités du langage AMPL est qu'il a une syntaxe très proche des expressions mathématiques en optimisation. AMPL peut fournir ∇f et $\nabla^2 f$ mais pas les ordres supérieurs. Il n'est donc pour l'instant pas possible de l'utiliser pour les directions plus complexes.

D'autre part, la librairie CUTEr, *A Constrained and Unconstrained Testing Environment, revisited*, fait partie des librairies les plus réputées pour tester des algorithmes d'optimisation. Elle fournit une collection de problèmes et fonctionne sur un grand nombre de plate-formes. Les problèmes tests sont écrits en SIF *Standard Input Format*. Malheureusement, même si ce code peut-être transformé en Fortran, il est difficilement exploitable par les outils de différentiation automatique qui ne gèrent pas le code SIF. En revanche, il existe une librairie qui est un sous-ensemble de CUTEr, écrite en Fortan : celle de Moré, Garbow et Hillstrom (MGH) [**?**] et qui est exploitable par les outils de DA. Dans le choix de l'outil de différentiation automatique, *Tapenade* nous est apparu comme le plus adéquat car il marche par transformation de code en mode inverse et direct. De plus, il traite et retourne un code en Fortran que l'on peut de nouveau différentier. Théoriquement, il est possible d'obtenir n'importe quel ordre de dérivation mais nous verrons les limitations de cet outil. Néanmoins, cet outil a déjà fait ses preuves dans certains milieux, par exemple pour modéliser la circulation océanique, [**?**].

Une fois que l'implémentation des dérivées sera faite, le but est de vérifier la convergence des algorithmes et de comparer les coûts de calculs des méthodes. La libraire de MGH [1], permettra de traiter un large éventail de cas possibles et évaluera la fiabilité et la robustesse des algorithmes.

1. disponible sur http://www.netlib.org/uncon/data/

4.1 Les outils de différentiation automatique

Il existe plusieurs outils de différentiation automatique, d'après le site spécialisé très connu en DA : autodiff[2], consulter le tableau 4.1. Les outils qui atteignent un ordre supérieur à deux de dérivation utilisent généralement le mode direct. On observe que la plupart des langages traités sont C/C++, Fortran et Matlab.

Logiciel	Langage	Transformation de code (t) surcharge (s)	Mode direct	Mode inverse	Ordre
ADC	C/C++	s	1	0	2
ADF	Fortran77, 95	s	1	0	2
ADG	Fortran77	s	0	1	>2
ADIC	C/C++	t	1	0	2
ADIFOR	Fortran77	t	1	0	2
ADiMat	MATLAB	t	1	0	2
ADMAT / ADMIT	MATLAB	s	1	1	2
ADOL-C	C/C++	s	1	1	>2
ADOL-F	Fortran95	s	1	1	>2
AUTO_DERIV	Fortran77, 95	s	1	0	2
COSY INFINITY	Fortran77, 95,C/C++	s	1	0	>2
CppAD	C/C++	s	1	1	>2
FAD	Haskell	s	1	0	>2
FADBAD/TADIFF	C/C++	s	1	1	>2
FFADLib	C/C++	s	1	0	2
GRESS	Fortran77	t	1	1	1
HSL_AD02	Fortran95	s	1	1	>2
INTLAB	MATLAB	s	1	0	2
NAGWare Fortran 95	Fortran77, 95	t	1	0	>2
OpenAD	C/C++,Fortran77, 95	t	1	1	2
PCOMP	Fortran77	t	1	1	2
pyadolc	python	s	1	1	2
pycppad	Interpreted,python	s	1	1	>2
Rapsodia	C/C++,Fortran95	s	1	0	>2
Sacado	C/C++	s	1	1	1
TAF	Fortran77, 95	t	1	1	2
TAMC	Fortran77	t	1	0	1
TAPENADE	C/C++,Fortran77, 95	t	1	1	>2
TOMLAB /TomSym	MATLAB	t/s	1	1	2

tableau 4.1 – Plusieurs outils de DA

2. http://www.autodiff.org/

4.2 Un outil de DA : *Tapenade*

Tapenade[3] est un outil de différentiation automatique qui a commencé à être développé en 1999 par une équipe du projet Tropics à l'INRIA. Il utilise la transformation de code. L'avantage, c'est que l'on va pouvoir différentier plusieurs fois puisque le code différentié est vu comme une routine classique.

4.2.1 Comment utiliser la DA pour les dérivées d'un point de vue théorique

Ce que l'on cherche, c'est obtenir les dérivées successives du code Fortran de manière efficace pour une dimension assez grande $n = 1000$.

$$F = \nabla f \in \mathbb{R}^n \to \mathbb{R}^n$$

Nous voulons l'expression $\nabla^2 f \cdot v$. Pour calculer par différentiation automatique, nous allons utiliser l'astuce suivante :

$$\nabla_x(\nabla_x f(x) \cdot d)) = \nabla^2_{xx} f(x) \cdot d$$

Où d représente la direction obtenue. Au lieu de calculer la hessienne, nous allons appliquer le mode direct sur le calcul du gradient par un vecteur. Regardons sur un exemple :

$$f(x) = x_1^3 x_2^2 - 10 x_1 x_2 - x_2^3$$

$$\nabla_x f(x) = F(x) = \begin{pmatrix} 3x_1^2 x_3^2 - 10x_2 & 2x_1^3 x_2 - 10x_1 - 3x_2^2 \end{pmatrix}$$

$$\nabla^2_{xx} f(x) = \begin{pmatrix} 6x_1 x_2^2 & 6x_1^2 x_2 - 10 \\ 6x_1^2 x_2 - 10 & 2x_1^3 - 6x_2 \end{pmatrix}$$

$$d = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$\nabla_x f(x).d = 3x_1^2 x_2^2 - 10x_2 + 4x_1^3 x_2 - 20x_1 - 6x_2^2$$

3. disponible sur http://www-sop.inria.fr/tropics/

$$\nabla_x(\nabla_x f(x)\cdot d)) = \begin{pmatrix} 6x_1x_2^2 + 12x_1^2x_2 - 20 & 6x_1^2x_2 - 10 + 4x_1^3 - 12x_2 \end{pmatrix}$$

$$\nabla^2_{xx} f(x).d = \begin{pmatrix} 6x_1x_2^2 + 12x_1^2x_2 - 20 \\ 6x_1^2x_2 - 10 + 4x_1^3 - 12x_2 \end{pmatrix}$$

Puisque la matrice hessienne est symétrique, on remarque bien que

$$\left(\nabla^2_{xx} f(x).d\right)^T = \nabla_x(\nabla_x f(x).d))$$

En mode inverse :

$$\psi(t) = F(x + t\cdot d) = F(g(t))$$

$g(t) = x + t\cdot d$ où d représente la direction

$$\psi'(t) = \nabla F(x + t\cdot d)d$$

$$\psi'(0) = \nabla F(x)d$$

Ainsi, pour obtenir $\nabla^2 f(x)\cdot v$, nous appliquons le mode direct après le mode inverse. Le fait d'appliquer ces deux modes, l'un après l'autre nous donnent le bon résultat uniquement parce qu'à la base nous utilisons une fonction scalaire et que la matrice hessienne est symétrique. Dans le cas contraire, si la $i$ème ligne de $\nabla^2 f$ ne correspondrait pas à sa $i$ème colonne et nous ne pourrions pas utiliser ce procédé. À partir de là, nous pouvons réappliquer plusieurs fois le mode direct pour atteindre $\nabla(\nabla f\cdot d)\cdot d$.

4.2.2 Utilisation de *Tapenade*

L'outil peut s'utiliser soit en local, soit en ligne grâce à un serveur. Son utilisation s'effectue en plusieurs étapes. D'abord, il faut fournir le code en Fortran qui contient le code à différentier sous la forme d'un fichier. Ensuite, il faut définir la routine que l'on souhaite différentier et les variables d'entrées par rapport à laquelle la différentiation doit être faite et les variables de sortie dépendantes. Enfin, il faut choisir le mode : tangent, inverse ou multi-directionnel. Supposons que la variable de sortie est $Y \in \mathbb{R}^n$, dépendante de $X \in \mathbb{R}^m$. En fait, nous avons $Y = f(X)$ avec $f : \mathbb{R}^m \to \mathbb{R}^n$. Notons $J := \nabla f \in \mathbb{R}^n \times \mathbb{R}^m$ la matrice jacobienne de f. La routine contient donc les arguments

Y ; une variable résultat et X la variable d'entrée.

Mode direct

Comme expliqué dans le paragraphe 3.2.1, en plus de spécifier la variable d'entrée X, nous allons aussi spécifier une variation dX sur laquelle la jacobienne va s'appliquer.

$$\begin{pmatrix} \dot{y_1} \\ \dot{y_2} \\ \vdots \\ \dot{y_m} \end{pmatrix} = dY = J(X) \times dX = \underbrace{\begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} & \cdots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \frac{\partial y_n}{\partial x_3} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix}}_{\text{Jacobienne en } X} \times \begin{pmatrix} \tilde{x_1} \\ \tilde{x_2} \\ \vdots \\ \tilde{x_m} \end{pmatrix}$$

Par exemple, si dX est initialisée à e_1, nous aurons la première colonne de la Jacobienne et si Y est un scalaire nous aurons la première composante de son gradient. Deux nouvelles variables vont être ajoutées. De la routine

```
SUBROUTINE FONCTION(X,Y)
```

le logiciel génère :

```
SUBROUTINE FONCTION_D(X, Xd, Y, Yd)
```

où `Y`, `Yd` sont les variables de sortie et `X`, `Xd` les variables d'entrée. On a donc `Y=`F`(X)` et `Yd=`∇F`(X).Xd`

Le mode multidirectionnel revient à appliquer à plusieurs vecteurs `Xd`, en fait `Xd` est une matrice.

Mode inverse

De même, un vecteur doit être spécifié en mode inverse mais comme le calcul se fait à rebours, l'opération est inversée.

$$\begin{pmatrix} \bar{x_1} \\ \bar{x_2} \\ \vdots \\ \bar{x_m} \end{pmatrix} = dX = \underbrace{J^*(X)}_{J(X)^T} \times dY = \underbrace{\begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \frac{\partial y_3}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_3}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \frac{\partial y_3}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix}}_{\text{Transposée de la jacobienne en } X} \times \begin{pmatrix} \bar{y_1} \\ \bar{y_2} \\ \vdots \\ \bar{y_m} \end{pmatrix}$$

En reprenant le même exemple de `SUBROUTINE FONCTION(X,Y)`

Tapenade génère :

```
SUBROUTINE FONCTION_B(X, Xb, Y, Yb)
```

cette fois-ci `Y` et `Xb`:= dX sont les variables de sortie et `X`, `Yb`:= dY les variables d'entrée. Si nous appliquons sur $dY = e_i$ nous obtiendrons la $i$ème ligne de la jacobienne. Le calcul correspond à `Xb`=$\nabla F(\texttt{X})^T$`Yb`

Dérivées supérieures Pour les ordres supérieurs, nous allons réappliquer le même procédé sur les fonctions obtenues. Par exemple appliquer le mode direct sur le mode inverse, c'est-à-dire sur la routine `SUBROUTINE FONCTION_B(X, Xb, Y, Yb)`
En spécifiant de différentier `Xb` par rapport à `x` :
`SUBROUTINE FONCTION_B_D(X, Xd, Xb, Xbd, Y, Yb)`
variables d'entrée `X Xd Xb Yb`
variables de sortie `Xbd, Y`
`Xbd`=$\nabla(\nabla F($`X`$)\cdot$`Yb`$)^T)\cdot$`Xd`

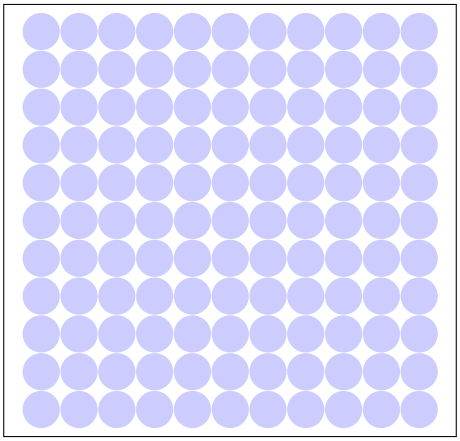
Malheureusement, le logiciel *Tapenade* n'a pas été conçu pour obtenir des dérivées supérieures à deux avec le mode inverse. D'autre part le mode inverse sur inverse n'existe pas (il serait trop compliqué à gérer). À l'aide d'un Makefile, j'ai généré l'ensemble des dérivées dont j'avais besoin. Pour chaque fonction, plusieurs fichiers sont générés : un pour chaque opération que l'on souhaite. Pour atteindre les dérivées d'ordre supérieur, les fichiers générés sont redonnés à l'outil pour être de nouveau différentiés.

4.2.3 Tests sur la librairie de Moré, Garbow, Hillstrom

Étant donné que nous voulons faire des tests de calcul sur des fonctions types, il est pertinent d'étudier le comportement des fonctions et notamment si la matrice hessienne est creuse. La figure 4.1 indique par des points les éléments non nuls de la matrice hessienne pour la fonction trigonométrique. On n'exploite pas le fait que la matrice soit creuse mais les coûts de calculs seront probablement diminués quand même car les opérations seront faites sur des zéros.

Nous allons présenter trois exemples de fonction appartenant à la librairie ; la fonction bien connue en optimisation Rosenbrock, généralisée à dimension variable, une fonction trigonométrique et une fonction utilisant les polynômes de Chebychev.

figure 4.1 – Matrice hessienne de taille 10 par 10 de la fonction trigonométrique, les points bleus représentent les éléments non nuls de la matrice.



Fonction trigonométrique

$$n \text{ variable, } m = n$$

$$f_i(x) = n - \sum_{j=1}^{n} cos(x_j) + i(1 - cos(x_i)) - sin(x_i)$$

$$x_0 = [1/n, \cdots, 1/n]$$

$$min_x f(x) = 0$$

La matrice hessienne de la fonction trigonométrique est une matrice pleine 4.1, cette fonction pourra nous servir de référence car les calculs sont relativement simples.

La fonction Rosenbrock étendue

$$n \text{ variable mais pair } m = 2$$

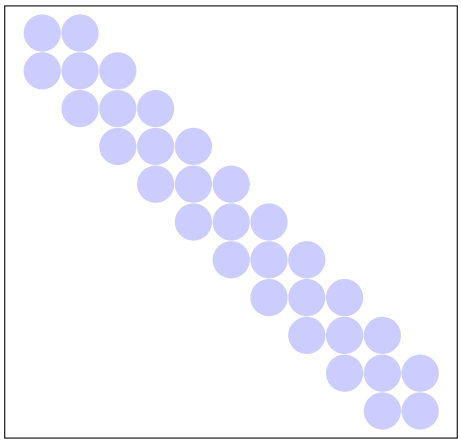
$$f_{2i-1}(x) = 10(x_{2i} - x_{i-1}^2)$$

$$f_{2i}(x) = 1 - x_{2i-1}$$

$$x_0 = (\xi_i) \text{ où } \xi_{2i-1} = -1.2 \text{ et } \xi_{2i} = 1$$

$$min_x f(x) = 0 \text{ en } [1, \cdots, 1]$$

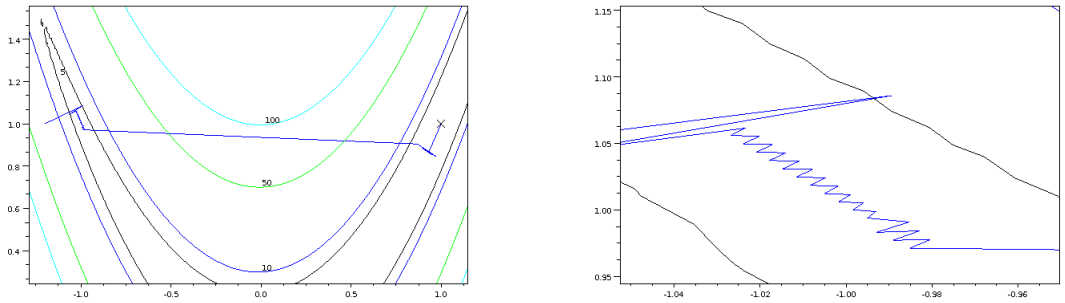
figure 4.2 – Matrice hessienne de la fonction de Rosenbrock étendue



Chaque composante de la fonction n'est dépendante que de deux variables, c'est pourquoi la matrice hessienne est creuse.

La fonction a été proposée par Rosenbrock en 1960 afin de comparer des algorithmes de descente. Dans le cas avec $n = 2$, la fonction forme un sillon étroit, ce qui oblige les méthodes de descente à suivre une courbe. L'agorithme du gradient par exemple est très médiocre car il "rebondit" sur chaque parois sans avancer vraiment 4.3. La figure de gauche montre les lignes de niveau de la fonction et les droites en bleues correnpondent aux itérés de l'algorithme du gradient. L'itéré initial est à gauche et la solution à droite. Au milieu, le "saut" a été trouvé par une recherche linéaire avancée. La figure de droite est un zoom de l'algorithme, nous pouvons observer la difficulté à trouver une direction qui permet d'avancer plus efficacement. En ce qui concerne la méthode de Newton, augmenter la dimension de la fonction n'influencera pas le nombre d'itérations car le problème pourra être vu comme $\frac{n}{2}$ problèmes de Rosenbrock qui s'effectuent parallèlement.

figure 4.3 – Alogrithme du gradient sur Rosenbrock : 19436 itérations



La fonction Chebyquad

$$n \text{ variable, } m = n$$

$$f_i(x) = \frac{1}{n}\sum_{j=1}^{n} T_i(x_j) - \int_0^1 T_i(x)dx$$

Où T_i est le ième polynôme de Chebychev réduit à l'intervale $[0,\ 1]$

$$\int_0^1 T_i(x)dx = 0 \text{ pour i paire}$$

$$\int_0^1 T_i(x)dx = \frac{-1}{i^2-1} \text{ pour i impaire}$$

$$x_0 = (\xi_j) \text{ où } \xi_j = j/(n+1)$$

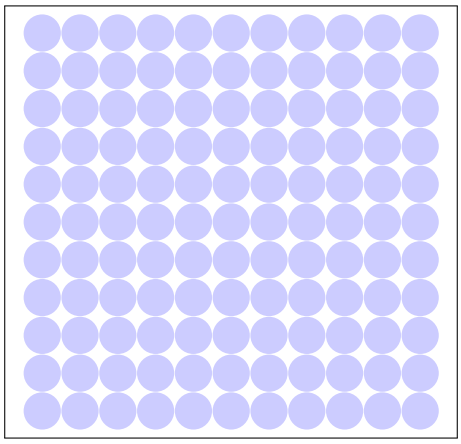
$$f = 0 \text{ pour } m = n,\ 1 \leq n \leq 7 \text{ et } n = 9$$

$$f = 3.51687...10^{-3} \text{ pour } n = m = 8$$

$$f = 6.50395...10^{-3} \text{ pour } n = m = 10$$

Ces trois fonctions ont des dimensions variables ; elle reflètent l'ensemble des résultats qui sont équivalents pour les calculs des dérivées. La fonction Chebyquad est plus particulière dans le sens où le temps d'exécution de la fonction n'est pas proportionnel à la dimension car les polynômes sont de plus en plus complexes à calculer en fonction de n.

figure 4.4 – Matrice hessienne de la fonction de Chebyquad



Temps de calcul du gradient fourni par la routine et du gradient recodé Pour la fonction trigonométrique, comme le montre 4.5, le calcul du gradient donné par la routine de netlib n'est pas efficace. Ceci s'explique par le fait que pour l'obtenir, on utilise la relation

$$\nabla f(x) = F(x)^T \nabla F(x)$$

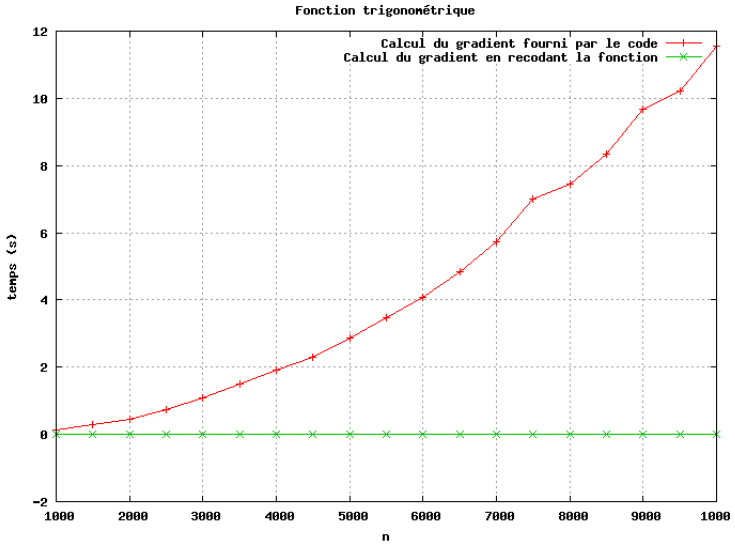
en notant que

$$f(x) = \frac{1}{2} \sum_{i=1}^{m} F_i(x)^2$$

Le calcul de la Jacobienne $\nabla F(x)$ de taille n par m impose beaucoup de calculs qui peuvent être évités. En recodant le gradient de la fonction, les multiplications sur les lignes sont factorisables, et en évitant le calcul de la jacobienne, on améliore nettement l'exécution. Pour une dimension relativement grande, le temps de calcul du gradient fourni prend plusieurs seconde alors que celui de la fonction recodé de manière optimisé est instantané.

Comparaison entre l'évaluation de la fonction et le mode inverse Afin de pouvoir comparer les temps d'exécution de la fonction et du mode inverse, les appels sont faits plusieurs fois dans une boucle de mille itérations. En effet, même pour une dimension de $n = 10000$, le temps pour évaluer la fonction est inférieur au pas

figure 4.5 – Temps d'évaluation du gradient en modifiant la taille de x et celui du gradient recodé



d'horloge unitaire de 4ms. On remarque sur la figure 4.6 que l'obtention du gradient par le mode inverse est clairement proportionnel au coût de la fonction avec un facteur de proportionnalité d'environ deux ; pour rappel, la borne théorique est de quatre. Le mode inverse est nettement plus performant qu'une implantation naïve.

Le mode multi-directionnel Le mode multi-directionnel correspond au mode direct appliqué à chacune des composantes ; il équivaut à effectuer $\nabla f(x).(e_i)$ pour chaque $1 \leq i \leq n$. Dans la figure 4.7, nous pouvons voir que le temps en mode multi-directionnel est environ le même que pour calculer le gradient par différences finies. Le mode multi-directionnel n'est pas avantageux puisqu'il effectue le mode tangent n fois. La figure 4.8 permet de comparer le mode multi-directionnel et le mode inverse ; pour une dimension assez grande, le temps de calcul pour le mode multi-directionnel est insatisfaisant et il sera préférable de l'éviter tant que c'est possible.

Hessien×vecteur Comme pour le mode inverse, le coût du hessien multiplié par un certain vecteur calculé par mode direct sur mode inverse est proportionnel au coût de la foncion 4.9 environ $4\times\#(f)$. En effet, les trois courbes sont très proches

figure 4.6 – Temps de calcul de la fonction et du gradient en mode direct par *Tapenade* dans une boucle de mille itérations, fonction trigonométrique

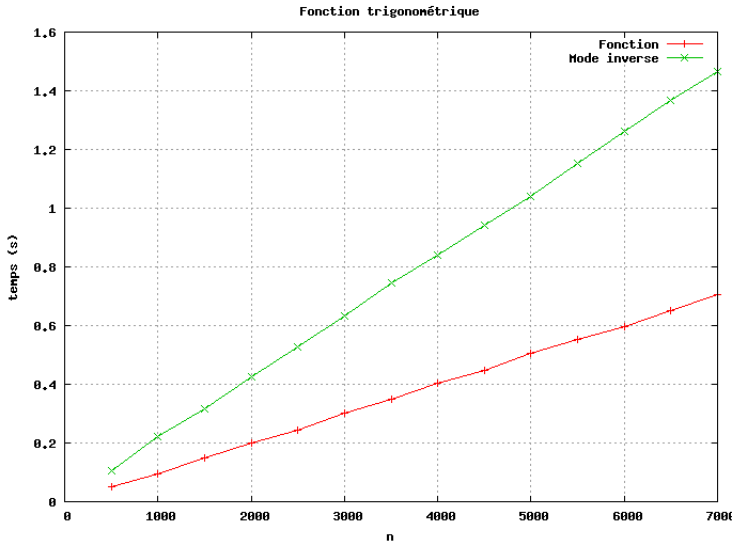


figure 4.7 – Temps de calcul - fonction trigonométrique

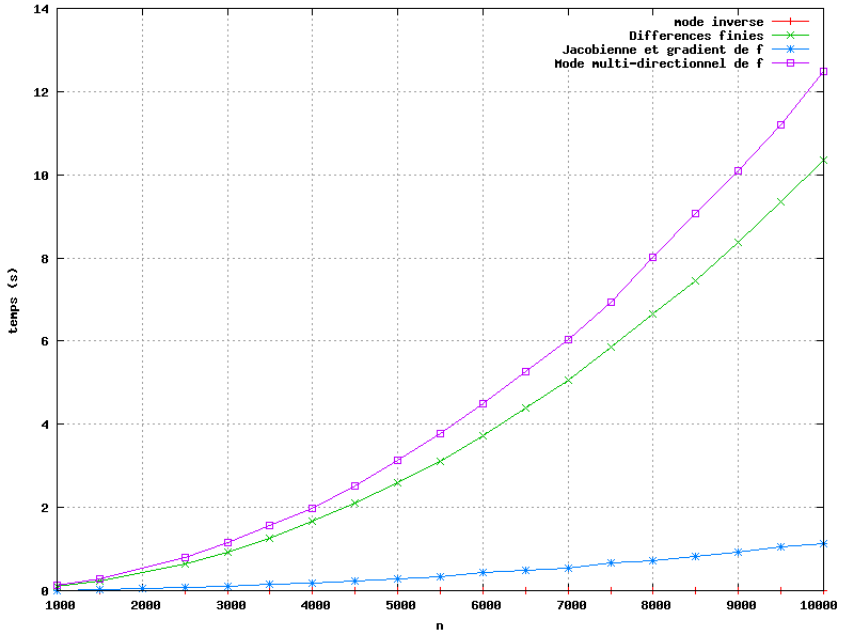
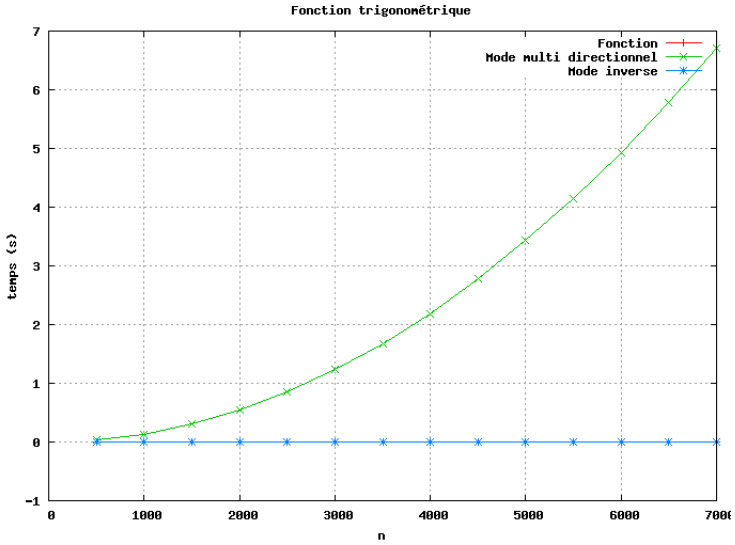


figure 4.8 – Mode multi-directionnel : $\nabla f(x)$



de droites. Ainsi, nous validons bien que la borne de complexité de $\nabla^2 f(x) \cdot v$ est $\mathcal{O}(\#(f))$.

Hessien Pour obtenir la matrice hessienne 4.10, nous avons pas le choix d'appliquer le mode multi-directionnel. Nous observons cette fois ci que le coût est proportionnel à n fois le coût de l'évaluation de la fonction.

Ordres supérieurs Le coût des opérations $\nabla^3 f(x) \cdot u \cdot v$ et $\nabla^4 f(x) \cdot u \cdot v \cdot w$ ne dépendent pas de n comme le montre la figure 4.11. Toutes les courbes sont affines.

4.2.4 Avantages et inconvénients de *Tapenade*

La transformation du code est la meilleure approche pour la DA ; elle donne la capacité de calculer les gradients à un faible coût puisqu'elle exécute de manière impérative comme le programme original. De plus, il n'y a pas de restriction sur le style ou la taille de l'application à différentier. Au lieu de coder à la main une fonction possédant plusieurs millions de lignes, ce qui est source d'erreurs et demande un effort considérable, il est plus pratique de le faire automatiquement. C'est un outil qui

figure 4.9 – Mode tangent sur inverse (vert $\times$) sur une boucle de mille itérations, ce qui correspond au calcul de $\nabla^2 f(x).v$ pour un certain vecteur, le résultat est donc aussi un vecteur

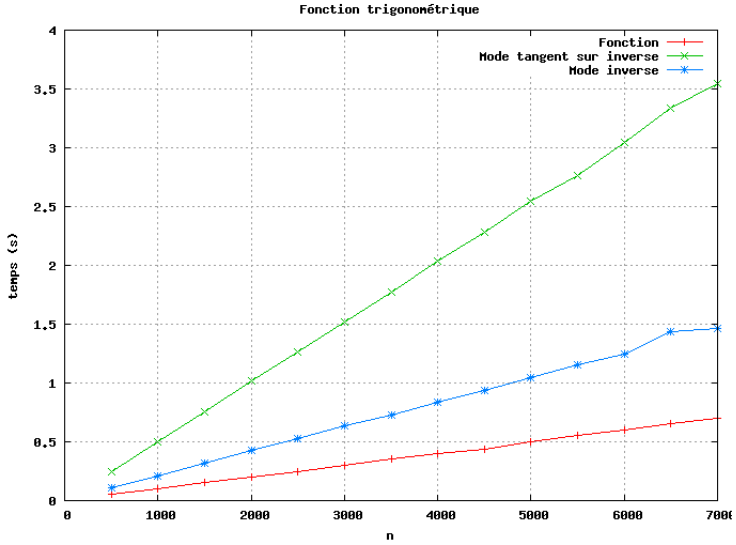


figure 4.10 – Mode multi-directionnel sur inverse (vert $\times$) ce qui donne la hessienne $\nabla^2 f(x)$ pour un certain vecteur, le résultat est donc aussi un vecteur

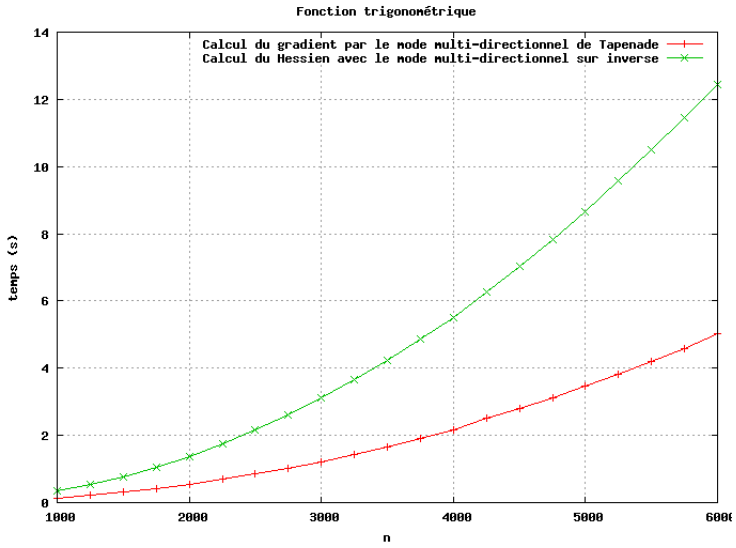
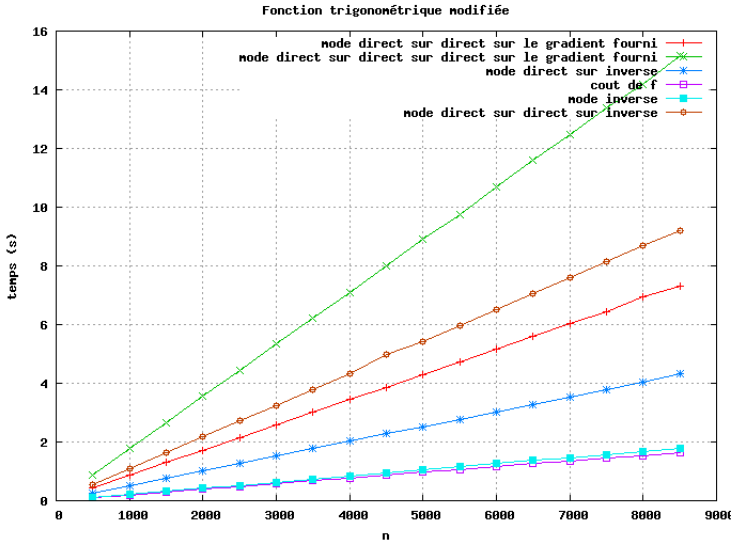


figure 4.11 – Temps des opérations $\nabla^4 f(x) \cdot u \cdot v \cdot w$ en vert et marron, $\nabla^3 f(x) \cdot u \cdot v$ en rouge : elles ne dépendent pas de n et sont proportionnelles au coût de la fonction.



progresse encore ; les performances du calcul de l'adjoint sont en cours d'amélioration. Cependant, il ne traite que du code Fortran et C, même si théoriquement, tous les langages peuvent être traités. Le schéma optimal de checkpoints imbriqués pour un programme quelconque reste un problème de recherche. De plus, les boîtes noires ne peuvent évidemment pas être gérées, ce qui n'est pas le cas pour la différentiation par différences finies. Pour finir, la plus grosse difficulté vient du fait que le logiciel ne gère pas l'ordre trois avec le mode inverse. Il a donc fallu adapter la librairie mais malgré tout, certains cas ne sont pas fiables, et donne par exemple une hessienne non symétrique.

4.2.5 Difficultés pour les dérivées supérieures

La complexité du stockage des variables lors de la différentiation automatique est expliquée dans le livre de Griewank [**?**]. Il analyse entre autres le stockage des variables avec l'ordre des opérations atomiques et le mode inverse répété. Bien qu'efficace, le mode inverse comporte des complications avec *Tapenade*, lorsque l'on veut différentier à un ordre supérieur. Contrairement au mode tangent qui ne fait que rajouter des

opérations élémentaires dans le code, le mode inverse va faire apparaître des appels à des routines : `PUSH` et `POP`. Celles-ci vont permettre de gérer une pile qui alternativement stockera et restituera les variables tel que décrit à la section 3.2.3. Elles sont codées en C et appartiennent à une libraire extérieure mais ne sont pas fournies à l'outil de différentiation. Leur propre code adjoint a été codé manuellement mais seulement à l'ordre un par *Tapenade*. Pour obtenir des dérivées d'ordre trois, il a fallu coder leurs dérivées. Lorsque l'on effectue le mode tangent sur tangent sur inverse, certaines routines correspondant aux PUSH et POP ne sont pas correct et il a fallu les modifier de manière automatique. Malgré cela, il existe de rares cas où la matrice hessienne n'est pas symétrique mais anti-symétrique.

4.3 Conclusion

Ce chapitre valide les bornes de complexité des calculs pour les dérivées définies dans le premier chapitre, à savoir que $\nabla f(x), \nabla^2 f(x) \cdot v$, $\nabla^3 f(x) \cdot v_1 \cdot v_2$ et $\nabla^4 f(x) \cdot v_1 \cdot v_2 \cdot v_3$ ont des coûts proportionnels au coût de l'évaluation de la fonction. Néanmoins, nous avons vu certaines limites de la DA ; la difficultés de traiter un code faiblement typé qui accepte certaines astuces et l'obtention des dérivées d'ordre trois et plus avec le mode inverse. À présent, comme nous venons d'obtenir ces bornes et que la résolution des systèmes linéaires est efficace nous allons pouvoir comparer les différentes méthodes de Newton d'ordre supérieur.

Chapitre 5

Comparaison des méthodes de type Newton et d'ordres supérieurs

5.1 Introduction

Sur l'ensemble des 35 problèmes, après avoir modifié à la main les codes générés, seuls deux problèmes ne donnent pas une valeur de Hessien correcte par *Tapenade*. Nous allons voir dans ce chapitre le temps d'exécution des dérivées et allons vérifier que les bornes théoriques de complexité sont respectées. Ensuite, nous étudierons les variantes de méthodes de descente sur la librairie MGH.

5.2 Méthode de descente avec recherche linéaire

Les méthodes d'ordres supérieurs requièrent souvent moins d'itérations. Nous présentons quelques expériences illustrant que notre implantation traduit cette réduction du nombre d'itérations dans une réduction du temps de calcul. Les directions ont été testées avec et sans recherche linéaire. Il s'avère que sans recherche linéaire, les algorithmes n'aboutissent pas dans beaucoup de cas car x_0 est éloigné de la solution et la direction peut être trop grande. Pour les méthodes de Newton et Chebychev, j'ai d'abord conservé le point initial donné dans les fonctions de la librairie. Ce point est généralement assez proche de la solution. Dans le tableau, nous notons x_{N_f} et x_{C_f} les points finals des méthodes de Newton et Chebychev respectivement. Cette norme confirme ou infirme le fait que les deux algorithmes convergent bien vers le même point.

On remarque sur la figure 5.1 que la méthode de Chebychev comme celle d'extrapolation d'ordre trois, n'aboutit pas dans plusieurs cas, le maximum d'itérations étant fixé à 999. Afin d'améliorer ceci, nous allons choisir la direction de Chebychev uniquement lorsqu'il s'agit bien d'une direction de descente. Dans le cas contraire nous reprendrons la direction de Newton.

L'utilisation de la direction de Newton lorsque celle de Chebychev n'est pas descendante améliore significativement l'algorithme. En revanche pour Halley, il n'y a presque pas de changement.

Ensuite, j'ai effectué les mêmes tests mais avec des dimensions plus grandes. Comme point de départ, j'ai d'abord choisi un nombre donné par la fonction random, à valeurs comprises entre zéro et cent. Sachant que les algorithmes ne sont pas

figure 5.1 – Profil des performances sur les fonctions de la librairie MGH, le point initial et les dimensions sont ceux par défaut. Les méthodes de Chebychev et Halley n'arrivent pas à la solution dans beaucoup de cas.

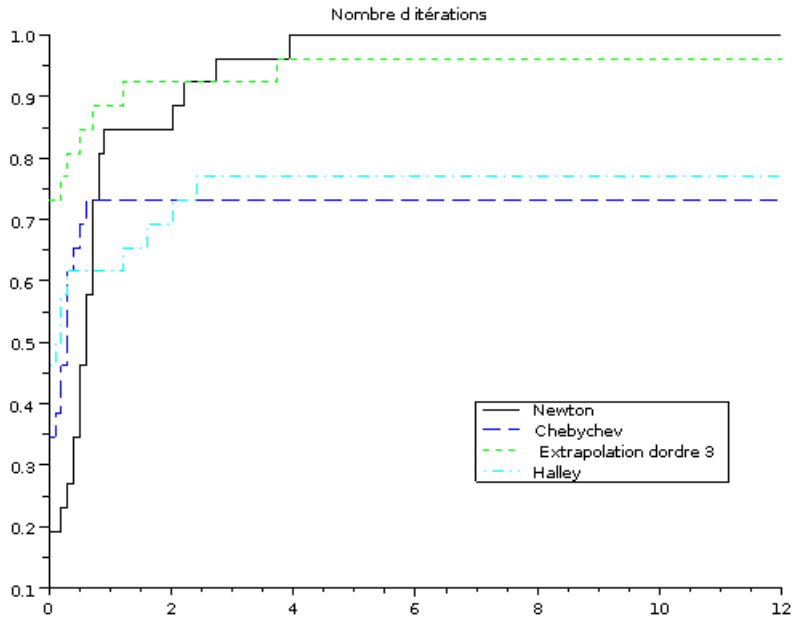
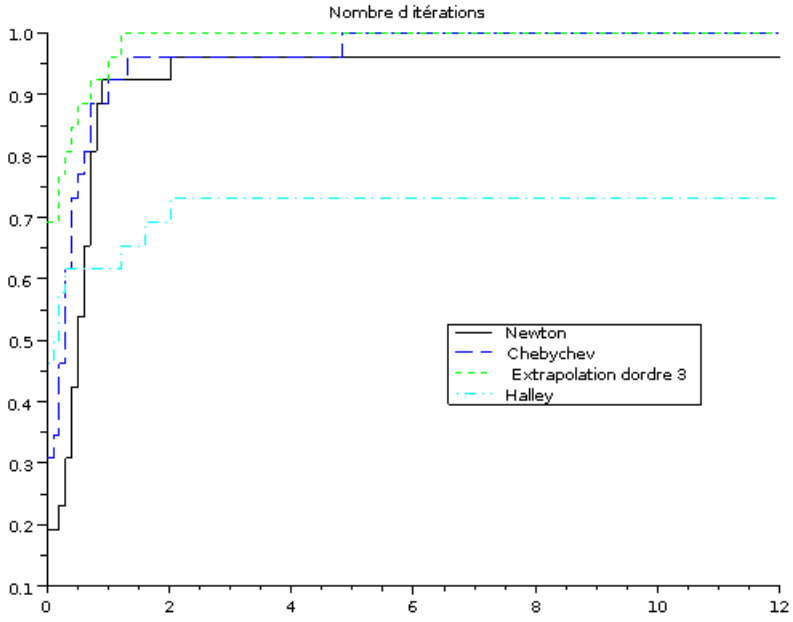


figure 5.2 – Profil des performances : les directions ne sont gardées uniquement s'il s'agit de direction de descente, sinon on reprend celle de Newton. Cette fois-ci l'extrapolation d'ordre trois réussit pour tous les problèmes.



globalement convergents, le tableau B.1 montre ce fait et en général, ils ne convergent pas vers le même point. Pour le tableau 5.1, j'ai choisi 1, 10 ou 100 fois la valeur du point initial de la librairie MGH. Dans les cas où les méthodes convergent vers le même point, on peut noter une amélioration dans le nombre d'itérations. Pour la première fonction du tableau, la différence des normes n'est pas égale mais les points finals obtenus ont un gradient vérifiant la condition $\nabla f(x) \leq \epsilon$. L'écart entre la valeur de l'objectif entre ces points est très faible. En ce qui concerne le temps d'exécution, il est meilleur pour Chebychev mais pour l'extrapolation d'ordre trois les temps sont moins bon car j'ai du partir du code fourni par le code des fonctions car il n'est pas possible d'atteindre l'ordre 4 avec Tapenade, même en essayant de modifier les routines PUSH et POP.

5.2.1 Figures qui illustrent les parcours

Rosenbrock Sur les figures 5.3 et 5.4, j'ai tracé le chemin qu'emprunte l'algorithme de Newton et de Chebychev pour la fonction de Rosenbrock. On observe bien que sans recherche linéaire, l'algorithme est plus rapide, cependant la valeur de l'objectif peut augmenter. À l'itération 2, la valeur de l'objectif atteint 1411.8, et si l'algorithme revient vers la solution, c'est ≪par chance≫ car il n'y a pas de convergence globale. Lorsque l'on applique une recherche linéaire, le parcours est mieux contrôlé et suit une ≪vallée≫ où la valeur de la fonction objectif reste faible.

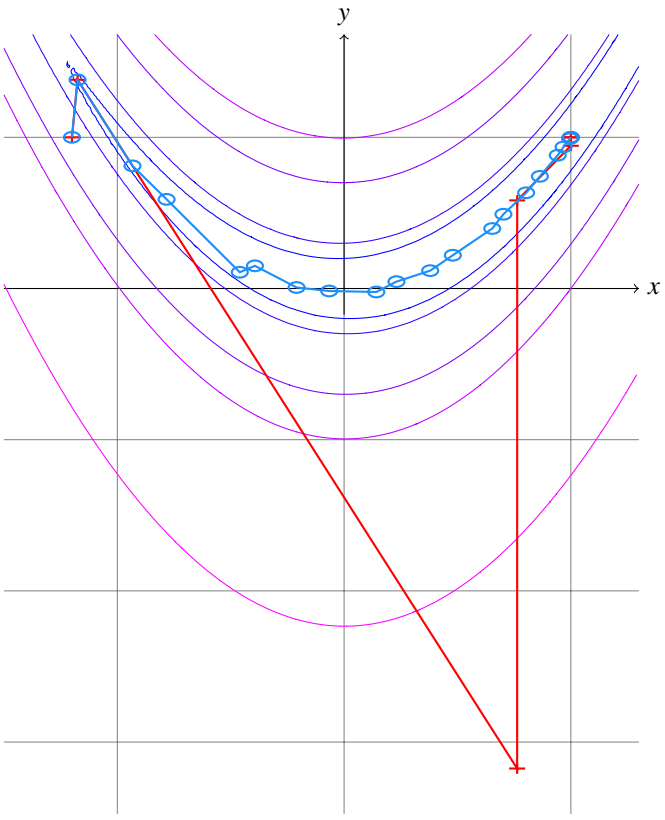
5.3 Conclusion

Par l'interprétation de ces résultats, nous pouvons conclure qu'il est possible d'obtenir des méthodes convergant plus rapidement que la méthode de Newton dans un cas général. De plus, l'efficacité est meilleure ; les temps de calcul sont réduits. Malheureusement, la DA est encore immature pour fournir l'ordre 4 de manière efficace. On peut cependant espérer que les outils de DA seront bientôt capable d'atteindre ces ordres de dérivations de manière systématique. En ce sens, nous pourrons gagner du temps grâce au méthodes d'ordre supérieur et ce, d'autant que la dimension du problème est grande.

fonction	$k * x_0$	n	IN	IC	I3	TN	TC	T3	$\|x_{N_f} - x_{C_f}\|$	$\|x_{N_f} - x_{3_f}\|$
bv	1	300	2	2	2	0.043	0.043	0.074	0.000042	0.000042
bv	10	300	4	**3**	4	0.082	0.065	0.149	0.000228	0.000308
bv	100	300	13	**10**	11	0.266	0.218	0.413	0.000029	0.000267
bv	1	500	7	6	**4**	0.407	0.375	0.433	0.000725	0.000473
bv	10	500	13	13	15	0.747	0.815	1.626	0.001876	0.002417
bv	100	500	23	**19**	21	1.323	1.193	2.289	0.001084	0.000554
bv	1	800	6	5	**3**	1.062	0.942	0.920	0.016409	0.141987
bv	10	800	11	9	**7**	1.938	1.717	2.175	0.706669	0.924390
bv	100	800	21	**20**	17	3.720	3.841	5.325	0.798127	0.800345
ie	1	300	4	3	3	1.834	1.398	1.468	0.000	0.000000
ie	10	300	7	5	5	3.205	2.330	2.474	0.000	0.000000
ie	1	500	4	3	3	8.044	6.105	6.348	0.000	0.000000
ie	10	500	7	5	5	14.054	10.155	10.556	0.000	0.000000
ie	1	800	4	3	3	32.521	24.463	25.169	0.000	0.000000
ie	10	800	7	5	5	56.313	40.456	41.555	0.000	0.000000
trid	1	300	7	6	6	0.152	0.129	0.204	0.000	0.000000
trid	10	300	12	10	10	0.246	0.215	0.344	0.000	0.000000
trid	100	300	18	15	14	0.369	0.325	0.482	0.000	0.000000
trid	1	500	7	6	6	0.397	0.366	0.579	0.000	0.000000
trid	10	500	12	10	10	0.673	0.613	0.967	0.000	0.000000
trid	100	500	18	15	**14**	1.015	0.906	1.333	0.000	0.000000
trid	1	800	7	6	6	1.177	1.080	1.656	0.000	0.000000
trid	10	800	12	10	10	2.043	1.842	2.757	0.000	0.000000
trid	100	800	18	15	14	3.067	2.760	3.854	0.000	0.000000
band	1	300	9	8	**7**	0.330	0.321	0.416	0.000	0.000000
band	10	300	19	15	15	0.695	0.608	0.894	0.000	0.000000
band	100	300	29	23	23	1.061	0.933	1.372	0.000	0.000000
band	1	500	9	8	**7**	0.904	0.895	1.149	0.000	0.000000
band	10	500	19	15	15	1.894	1.633	2.398	0.000	0.000000
band	100	500	29	23	23	2.892	2.508	3.680	0.000	0.000000
band	1	800	9	8	**7**	2.538	2.408	3.018	0.000	0.000000
band	10	800	19	15	15	5.391	4.570	6.468	0.000	0.000000
band	100	800	29	23	23	8.231	7.040	9.933	0.000	0.000000

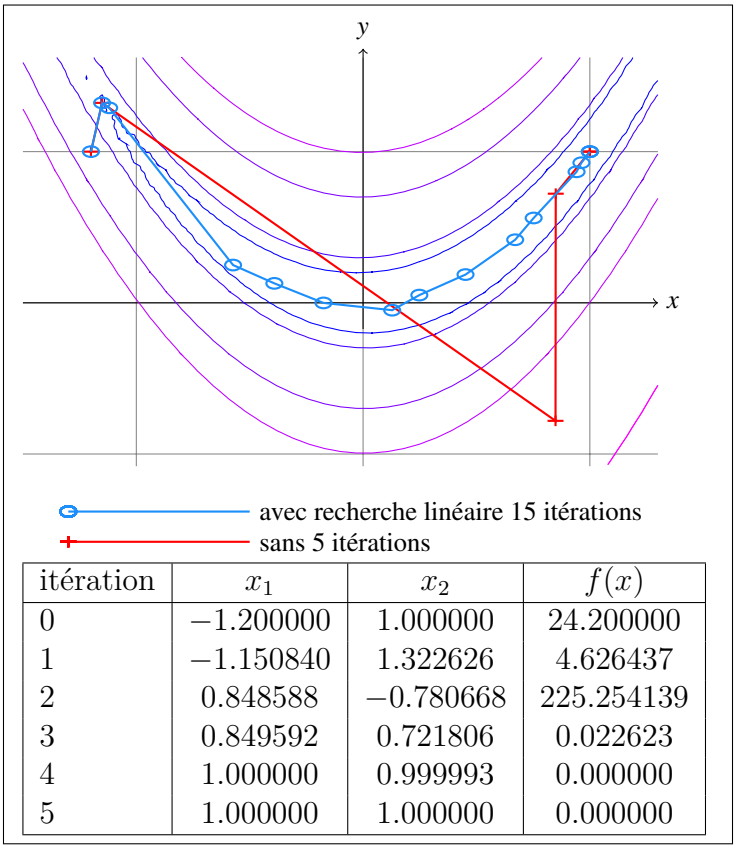
tableau 5.1 – En testant avec des dimensions plus grandes. Comme point initial : 1, 10 ou 100 fois x_0. Le temps pour l'extrapolation d'odre 3 est plus grand car les dérivées sont calculées à partir du gradient fourni et non du mode inverse. Les points finals de la première fonction vérifient les conditions d'un gradient suffisamment petit.

figure 5.3 – Newton - La recherche linéaire restreint à fournir des itérés dont la valeur de l'objectif est toujours décroissante tandis que sans recherche, on s'éloigne pour converger plus vite.



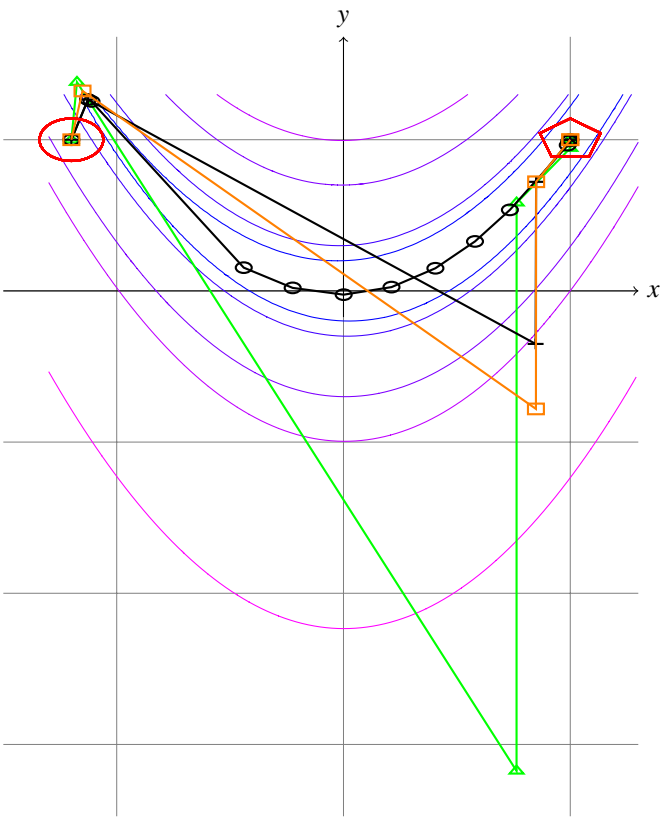
itération	x_1	x_2	$f(x)$
0	-1.200000	1.000000	24.200000
1	-1.175281	1.380674	4.731884
2	0.763115	-3.175034	1411.845179
3	0.763430	0.582825	0.055966
4	0.999995	0.944027	0.313189
5	0.999996	0.999991	0.000000
6	1.000000	1.000000	0.000000

figure 5.4 – Chebychev - La direction de Chebychev est meilleure sur l'exemple, cependant qu'une seule itération n'est gagnée



itération	x_1	x_2	$f(x)$
0	-1.200000	1.000000	24.200000
1	-1.150840	1.322626	4.626437
2	0.848588	-0.780668	225.254139
3	0.849592	0.721806	0.022623
4	1.000000	0.999993	0.000000
5	1.000000	1.000000	0.000000

figure 5.5 – Ordre supérieur : la direction s'éloigne encore moins de la vallée que les procédés de Newton ou Chebychev.



Algorithme d'extrapolation d'ordre 3 sans recherche liénaire
Algorithme d'extrapolation d'ordre 3 avec recherche liénaire
Algorithme de Chebychev
Algorithme de Newton

itération	x_1	x_2	$f(x)$
0	-1.200000	1.000000	24.200000
1	-1.126673	1.265834	4.524003
2	0.847210	-0.350714	114.187943
3	0.849335	0.721366	0.022700
4	1.000000	1.000000	0.000000
5	1.000000	1.000000	0.000000

Conclusion

Nous avons réussi à développer un environnement au sein de *Scilab*, grâce à une interface avec fortran, nous permettant d'obtenir les dérivées d'ordres supérieurs de la librairie de Moré, Garbow, Hilstrom. Le logiciel Tapenade a été adapté pour fournir ces dérivées de manière efficace, y compris l'ordre trois alors qu'il n'est pas conçu pour le faire en mode inverse. D'autre part, la généricité des fonctions de la librairie de MGH a permis une automatisation de la génération des codes différentiés. Étant donné que ces fonctions ont toutes les mêmes arguments, la différentiation se fait toujours par rapport aux mêmes variables. Ainsi, pour automatiser le traitement et la génération des codes différentiés, il suffit d'avoir des fonctions ayant les mêmes paramètres. Enfin, grâce à cet environnement nous avons implémenté des méthodes efficaces notamment avec une décomposition de Cholesky modifiée provenant de la librairie *Fortran* LAPACK pour la résolution des systèmes linéaires. Ainsi, plusieurs méthodes ont pu être testées, parfois soldées par des directions non pertinentes, parfois par des directions meilleures que les classiques de Newton et Chebychev. Cependant, la génération du code demande un pré-traitement du code source lorsqu'il n'est pas écrit de manière rigoureuse ou trop astucieuse. De plus, comme Tapenade n'a pas été conçu pour utiliser le mode inverse à un ordre supérieur à deux, des incohérences apparaissent à l'ordre trois avec ce mode et il faut modifier le code que Tapenade produit. Le résultat n'est pas fiable à cent pourcent. Par conséquent, ces parties de code à modifier ne peuvent pas être automatisées. Voir l'annexe C pour plus de détails. Même si le gain en temps n'est pas énorme, on peut espérer que les versions futures de Tapenade pour les ordres trois et quatre permettront cette avancée.

Il est certain que les progrès de la DA vont ouvrir la voie à des méthodes d'optimisation plus élaborées. Comme nous l'avons vu, la complexité est d'utiliser la tran-

formation de code pour l'efficacité. En effet, la surcharge des opérateurs est beaucoup plus flexible et facile à utiliser mais plus lente. La transformation de code requiert des calculs explicites et sans astuces. Une fois cette étape franchie, nous pourrions voir apparaître des méthodes plus performantes remplaçant la méthode de Newton. En réalité, ce genre de méthodes ne seraient qu'applicables sur des problèmes où la méthode de Newton marche, sinon il y a de fortes chances pour que ces méthodes échouent aussi. Néanmoins, dans ce cas, pour des dimensions relativement grandes, elles pourraient permettre un gain de temps.

On peut imaginer qu'à l'avenir, certains compilateurs pourraient intégrer des outils de DA permettant d'obtenir des ordres supérieurs sans l'intervention humaine, c'est-à-dire que le procédé serait entièrement automatisé. En effet, en analysant le code source du programme principal, il serait en mesure de savoir quel ordre à atteindre et quels paramètres utiliser.

Annexe A

Première annexe

A.1 Définitions

Définition A.1.1 ***(Fonction continue)***
Soient $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$ et $x_0 \in X$. La fonction f est continue en x_0 si et seulement si

$$\lim_{x \to x_0} f(x) = f(x_0)$$

c'est-à-dire que si, pour tout $\epsilon \in \mathbb{R}$, $\epsilon > 0$, il existe $\eta > 0$ tel que

$$\|x - x_0\| < \eta \text{ et } x \in X \Rightarrow \|f(x) - f(x_0)\| < \epsilon$$

Définition A.1.2 ***(Valeurs et vecteurs propres)***
Soit une matrice carrée $M \in \mathbb{R}^{n \times n}$. Les valeurs propres de A sont les racines de son polynôme caractéristique

$$p_M(z) = \det(zI - M)$$

où I est la matrice identité de dimension n. Si λ est une valeur propre de A, les vecteurs $x \in \mathbb{R}^n$ non nuls tels que

$$Mx = \lambda x$$

sont appelés vecteurs propres.

Définition A.1.3 ***(Matrice définie positive)***
Une matrice carrée $A \in \mathbb{R}^{n\times n}$ est dite définie lorsque

$$x^T Ax > 0,\ \forall x \in \mathbb{R}^n,\ x \neq 0$$

si de plus A est symétrique, toutes ses valeurs propres sont strictement positives.

Définition A.1.4 ***(Matrice bande)***
Une matrice carrée $A \in \mathbb{R}^{n\times n}$ est dite matrice bande si tout ses éléments en dehors de la bande diagonale bornée par deux constantes k_1 et k_2 sont nuls :

$$a_{ij} = 0 \text{ si } j < i - k_1 \text{ ou } j > i + k_2,\ k_1, k_2 \geq 0$$

Définition A.1.5 ***(Fonction lipschitzienne)***
Soit I un intervalle de $\mathbb{R}$ non vide et non réduit à un point et $f : I \to \mathbb{R}$ une application alors on dit que f est k-lipschitzienne s'il existe un k réel strictement positif tel que

$$\forall (x, y) \in I^2, |f(x) - f(y)| \leq k|x - y| \tag{A.1}$$

La constante k est dite constante de Lipschitz.

Annexe B

Les difficultés rencontrées

B.1 Librairie Moré, Garbow et Hillstrom

Sur le site de netlib [1], on peut retrouver plusieurs librairies écrites en fortran. Celle qui nous intéresse se trouve plus précisément dans les problèmes sans contraintes [2].

Pour faciliter l'utilisation de tapenade, les fichiers sources ont dû être modifiés. Tout d'abord, j'ai nommé les fonctions par leur nom (à la place de `getfun`) pour pouvoir avoir un appel unique à la routine. J'ai remplacé tous les appels à la fonction `ddot` par les opérations qu'effectue cette routine (produit scalaire). Il arrive qu'au lieu de passer un vecteur en argument, on donne un scalaire : `g(j) = ddot( m, fj( 1, j), 1, f, 1)` l.155 dans rose.f par exemple, ce qui a pour effet de multiplier la $j$ème colonne de `fj` par `f`. Cependant, Tapenade ne peut analyser ce code pour le différentier parce qu'il s'agit d'une astuce sur l'incrémentation de `fj(1,j)` qui donne `fj(2,j)` ce qui est propre au fonctionnement interne de Fortran.

De plus, certains paramètres ne sont initialisés que pour le mode -1 lors de l'appel à la fonction or ces paramètres sont utilisés pour les autres modes. J'ai donc rajouté cette initialisation au début de la fonction. Par exemple `nqrtr = 1` au début sing.f.

— Changement du nom des routines getfun par le nom de la fonction (celui du fichier)

— Remplacement des ddot par la somme des carrés

1. http://www.netlib.org
2. http://www.netlib.org/uncon/data/

— Remplacement des dcopy par copie éléments par éléments
— Rajout des initialisations des paramètres qui le sont uniquement dans le mode -1
— parameter (one = 1.d0) n'est pas défini dans badscp.f!!
— Rajout de `ftf=0d0` pour éviter d'additionner les valeurs lorsque l'on fait plusieurs appels

L'ensemble de ces modifications peut être retrouvé dans le script `make`.

B.2 Méthodes d'ordres supérieurs

Comme le montre le tableau suivant, le point initial a une grande importance dans l'exécution des méthodes. En choisissant un point aléatoire, il y a moins de chance pour que les algorithmes convergent vers le même point. D'autre part, la convergence n'est pas garantie.

fonction	n	Iter Newton	Iter Cheb	temps Newton	temps Cheby	Norme diff
rose	2	77	5	0.075	0.004	0.000000
froth	2	17	12	0.015	0.011	0.000000
badscp	2	19*	999	0.104	5.140	73081.313391
badscb	2	18	999	0.024	5.814	999915.027828
beale	2	14*	999	0.074	4.536	28470.397921
jensam	2	1	1	0.001	0.002	Nan
helix	3	14	10	0.013	0.014	0.000000
bard	3	9	2	0.009	0.002	Nan
gauss	3	1	1	0.001	0.001	0.000000
meyer	3	271*	999	0.302	6.159	6168.587560
gulf	3	1	1	0.001	0.001	0.000000
box	3	21	6	0.021	0.016	Nan
sing	4	26	19	0.024	0.017	0.000136
wood	4	27	21	0.025	0.024	0.000000
kowosb	4	230	2	0.221	0.003	Nan
bd	4	13	12	0.013	0.011	0.000000
rosex	10	203	5	0.207	0.005	0.000000
singx	12	30	24	0.030	0.023	0.000553
pen1	4	42	30	0.038	0.027	0.000000
vardim	10	26	19	0.025	0.018	0.000000
trig	10	20	2	0.023	0.027	Nan
almost	10	25	28	0.078	0.297	Nan
bv	10	22	999	0.022	5.438	185.331743
ie	10	21	17	0.025	0.029	0.000000
band	10	31	25	0.033	0.034	0.000000
cheb	10	167	999	0.208	5.010	1.674059
ie	100	89	360	2.251	11.208	0.000000
ie	200	177	234	27.493	39.950	0.000000
trig	100	46	45	0.340	0.343	0.069866
trig	250	61	58	2.104	2.114	14217.789748
trig	350	61	65	4.270	4.700	0.022663
bv	500	20	16	1.172	1.016	0.0066708
bv	1	19	15	5.404	4.585	0.1489379
bv	1500	17	14	10.86	9.613	12.905217
bv	2	17	13	19.097	15.863	17.806732

tableau B.1 – Nombre d'itération des méthodes de Newton et Chebychev mais sur un point initial loin de la solution. Les algorithmes convergent rarement au même point.

Annexe B. Les difficultés rencontrées

Annexe C

Troisième annexe

C.1 La surcharge des opérateurs

Programmation paresseuse Pour commencer à me familiariser avec la différentiation automatique, j'ai d'abord essayé de concevoir un programme qui dérive en programmation fonctionnelle. D'après l'article de Karczmarczuk [**?**], il est possible de calculer la différentiation automatique avec un langage fonctionnel de manière paresseuse. La sémantique du programme original va être étendue par surcharge des opérateurs en utilisant les règles usuelles de dérivation. Par exemple, la règle de Leibniz $(fg)' = f'g + fg'$ où la règle d'enchaînement : $(f(g(x))' = f'(g(x))g(x)$. Pour toutes opérations élémentaires, nous allons surcharger par les opérations de dérivation. Pour cela, on considère une paire (e, e') qui représente la valeur orginale et sa dérivée. De cette manière, les constantes seront représentées par $(c, 0)$ et la variable $(x, 1)$. Toutes les opérations vont être surchargées pour ce type : $(f, f') + (g, g') = (f + g, f' + g')$, $(f, f') \cdot (g, g') = (f \cdot g, f' \cdot g + f \cdot g')$, $cos(f, f') = (cos(f), cos(f) \cdot f')$ et ainsi de suite.

Le langage fonctionnel que j'ai choisi est caml. On commence par définir un type expression qui traduit les opérations élémentaires. Comme il s'agit d'un exemple, la liste est non exhaustive. Le type expression est d'abord introduit, il va nous permettre d'analyser le type d'opération. En caml, il est impossible de faire un "match" sur une fonction par exemple :

```
| cos -> sin
```

c'est pour cette raison que l'on définit un constructeur de type : expression.

```
type expression=
    Const of float
  | Var of string
  | Opp of expression
  | Plus of expression*expression
  | Moins of expression*expression
  | Mult of expression*expression
  | Quot of expression*expression
  | Puiss of expression*float
  | Cos of expression
  | Sin of expression
  | Exp of expression
  | Log of expression
;;
```

Pour différentier nos constantes de nos variables, il faut introduire lors de l'évaluation un environnement qui fournira la valeur de chaque variable. Par exemple si $x = 3.5$ et $y = -2.1$, $env = [(\text{"}x\text{"}, 3.5); (\text{"}y\text{"}, -2.1)]$ (les parenthèses ne sont pas nécessaires mais permettent de bien comprendre qu'il s'agit d'une liste de couples).

Ainsi, pour évaluer une expression, nous n'aurons plus qu'à faire :

```
let rec evaluer env expr = match expr with
    Const c-> c
  | Var v->(try List.assoc v env with Not_found ->
 raise(Unbound_variable v))
  | Opp f-> -.evaluer env f
  | Plus(f,g) -> evaluer env f +. evaluer env g
  | Moins(f,g) -> evaluer env f -. evaluer env g
  | Mult(f,g) -> evaluer env f *. evaluer env g
  | Quot(f,g) -> evaluer env f /. evaluer env g
  | Puiss(f,g) ->(evaluer env f)**g
  | Cos(f) -> cos(evaluer env f)
  | Sin(f) -> sin(evaluer env f)
```

```
  | Log(f) -> log (evaluer env f)
  | Exp(f) ->  exp (evaluer env f)
 ;;
```

`List.assoc v env` permet de renvoyer l'élément correspondant à `v` dans la liste de couple `env`. Avec notre exemple, `List.assoc "x" env` retourne `3.5`. Cette évaluation est la traduction du GAO. Le point après l'opérateur signifie que les composantes sont des *float*. (Les opérateurs ne sont pas surchargés et le + est pour les entiers).

Pour évaluer la dérivée :

```
let rec derive expr dv =
    match expr with
      Const c -> Const 0.0
    | Var v -> if v = dv then Const 1.0 else Const 0.0
    | Opp f -> Opp(derive f dv)
    | Plus(f, g) -> Plus(derive f dv, derive g dv)
    | Moins(f, g) -> Moins(derive f dv, derive g dv)
    | Mult(f, g) -> Plus(Mult(f, derive g dv), Mult(derive f dv, g))
    | Quot(f, g) -> Quot(Moins(Mult(derive f dv, g), Mult(f, derive g dv)),
                         Mult(g, g))
    | Puiss(f,g) -> Mult(Const g,Mult(derive f dv,f))
    | Cos(f) -> Opp(Mult(Sin(f),derive f dv))
    | Sin(f) -> Mult(Cos(f),derive f dv)
    | Exp(f) -> Mult(derive f dv,Exp(f))
    | Log(f) -> Quot(derive f dv,f)
 ;;
```

Karczmarczuk a proposé une manière d'obtenir les dérivées d'ordres supérieurs de manière paresseuse avec Haskell, un langage fonctionnel. La définition précédente est reprise et étendue mais sur une liste infinie $f :: f' :: f'' :: f^{(3)} \cdots$ représentant l'expression avec l'ensemble de ses dérivées. De la même manière, les constantes seront représentées par $c :: 0 :: 0 \cdots$ et la variable $x :: 1 :: 0 :: 0 \cdots$. En notant $f = (f_0 :: \bar{f})$ et $g = (g_0 :: \bar{g})$ où f_0, g_0 sont les éléments en tête de liste et $\bar{f}$, $\bar{g}$ sont les listes queues,

les opérations seront définies :

$$f + g = (f_0 + g_0 :: \bar{f} + \bar{g})$$

$$f \cdot g = (f_0 \cdot g_0 :: f \cdot \bar{g} + \bar{f} \cdot g)$$

$$f/g = w \text{ où } (f_0/g_0 :: (\bar{f} \cdot g + f \cdot \bar{g}) \cdot w^2)$$

On observe que la définition est auto-récursive. Évidemment, nous ne devrons pas évaluer toute la liste mais seulement les dérivées qui nous intéressent. Si on essaye d'obtenir `w`, on boucle à l'infini ! Étant donné que *Caml*[1] n'est pas un langage paresseux, contrairement à Haskell, il a fallu construire un nouveau type que l'on évaluera uniquement quand nous en aurons besoin.

```
type 'a glacon =
| Inconnu of (unit -> 'a)
| Connu of ' a;;
```

Le type `(unit -> 'a)` représente une fonction sans argument. Le résutlat n'est que potentiellement présent ; uniquement lorsque l'on évaluera cette fonction.

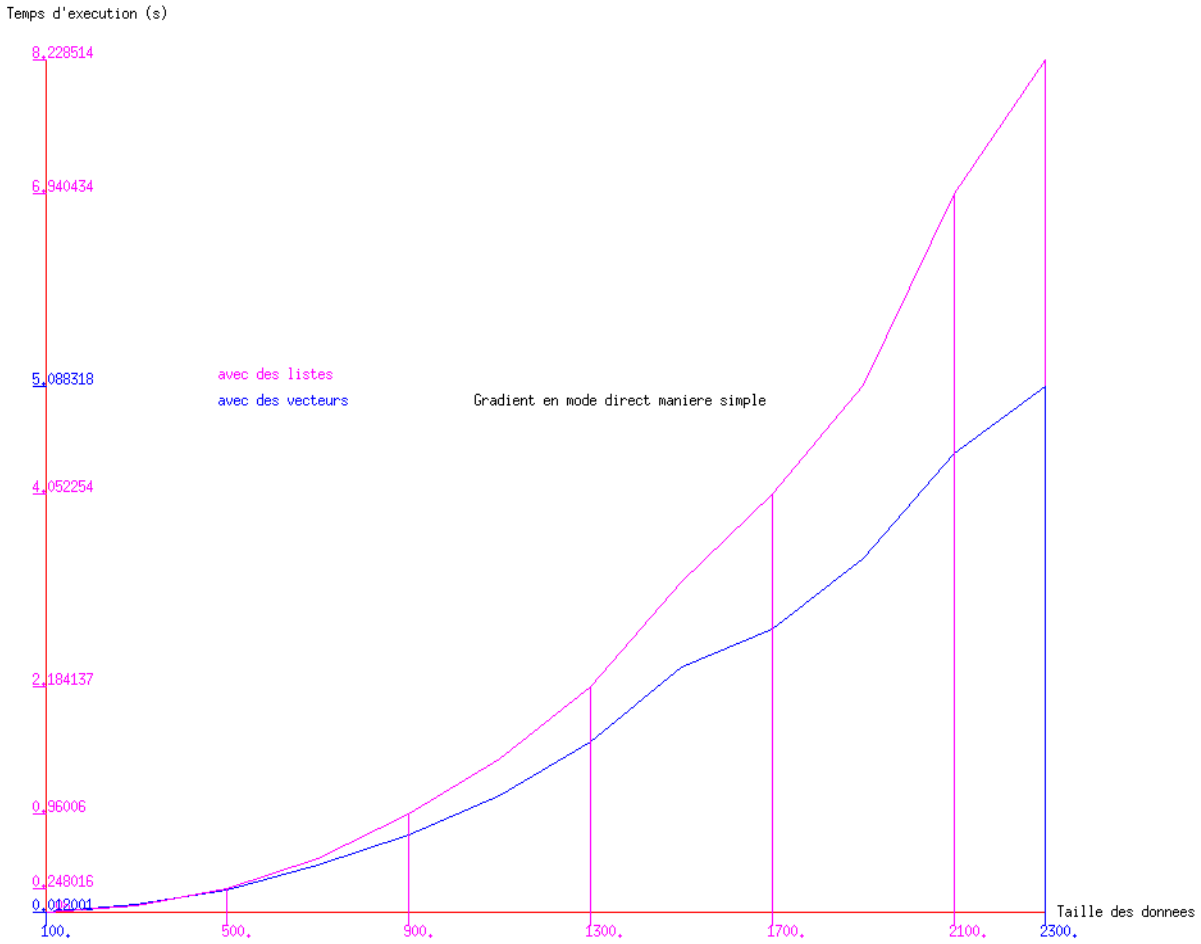
```
type 'a liste_paresseuse =
| Nil
| Cons of 'a cellule
and 'a cellule = { hd : 'a; mutable tl : 'a liste_paresseuse glacon};;
```

```
let force cellule =
  let glacon = cellule.tl in
  match glacon with
  | Connu valeur -> valeur
  | Inconnu g ->
     let valeur = g () in
     cellule.tl <- Connu valeur;
     valeur;;
```

Forcer la cellule revient à évaluer la fonction `g`. La figure C.1 illustre le fait que ces

1. http ://caml.inria.fr/

figure C.1 – Temps d'évaluation du gradient en mode direct par surcharge des opérateurs sur des listes et vecteurs avec caml



opérations impliquent des structures de plus en plus complexes à gérer.

On peut généraliser les listes infinies à plusieurs dimensions avec des dérivées partielles : $f = (f_0, [\bar{f}_1, \cdots, \bar{f}_n]$ où $\bar{f}_k = (\partial f/\partial f_k, [\cdots$ dérivées de $f'_k \cdots])$. L'inconvénient d'une telle méthode vient de la structure qui est rapidement lourde à gérer ; les listes sont très dures à manipuler lorsqu'elles sont de grandes tailles ; $\mathcal{O}(n)$ dans le pire des cas et les vecteurs ne sont pas dynamiques.