Applications de la différentiation automatique à la programmation non linéaire

par

Benoit Hamelin

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, mai 2004

Le 9 Juillet 04,
Date

le jury a accepté le mémoire de M. Benoit Hamelin dans sa version finale.

Membres du jury

M. Jean-Pierre Dussault
Directeur
Département d'informatique

M. François Dubeau
Membre
Département de mathématiques

M. Marc Frappier
Président-rapporteur
Département d'informatique

Your file Votre référence
ISBN: 0-494-00265-4
Our file Notre référence
ISBN: 0-494-00265-4

SOMMAIRE

Dans ce mémoire, nous introduisons d'abord les concepts et techniques de la différentiation automatique, une méthode de calcul des dérivées d'une fonction $\mathbb{R}^n \to \mathbb{R}^m$ précise et efficace. Nous présentons le design et l'implantation dans l'environnement Scilab d'un outil logiciel modulaire basé sur ces techniques.

Nous présentons ensuite une difficulté numérique affectant les méthodes de descente en optimisation sans contrainte de fonctions différentiables. Nous formulons cette difficulté en termes d'erreurs d'annulation survenant dans le calcul de différences finies de la fonction objectif. Enfin, nous proposons une solution basée sur les techniques de différentiation automatique, qui consiste en un calculateur de *différences finies automatiques.* La performance de cette solution est illustrée à l'aide d'expériences numériques sur une banque de problèmes de laboratoire, expériences réalisées avec des codes d'optimisation publics.

Nous complétons l'étude des différences finies automatiques en comparant empiriquement leur usage pour le calcul de dérivées directionnelles aux techniques traditionnelles de dérivation par différentiation automatique.

REMERCIEMENTS

J'aimerais tout d'abord adresser ma plus sincère reconnaissance à Jean-Pierre Dussault, directeur des travaux ayant mené à ce mémoire. Quiconque travaille à son contact ne peut être que stimulé par sa curiosité et enrichi par ses connaissances; hormis cela, je le compte parmi mes plus chers amis. J'aimerais aussi saluer la «bande du café», Marie, Francis, Jessica, François et Daniel : complices des bons moments, supports des moins bons.

Merci mille fois à ma fiancée, Mélodie Maurice, qui m'a appuyé et encouragé tout au long de ces travaux : j'aurai fort à faire pour lui rendre convenablement toutes les soirées et fins de semaine passées devant la machine. Merci aussi à ma mère et ma soeur, à qui je dois beaucoup.

TABLE DES MATIÈRES

Benoit Hamelin et Jean-Pierre Dussault

Scilab-2004 International Conference

LISTE DES TABLEAUX

LISTE DES FIGURES

INTRODUCTION

L'optimisation sans contrainte des fonctions différentiables constitue à ce jour une discipline exhaustivement étudiée, étayée et maîtrisée de la programmation non linéaire. On se sert notamment de ses techniques comme outils de base à la résolution de divers problèmes plus complexes restreints, relaxés ou transformés. À la base, pour une fonction $f \in \mathcal{C}^p(\mathbb{R}^n; \mathbb{R})$ (avec $p \geq 2$), on considère qu'un minimum local x^* de f doit être tel que $\nabla f(x^*) = 0$ et que $\nabla^2 f(x^*)$ soit une matrice semi-définie positive; par ailleurs, si un point $\overline{x}$ satisfait que $\nabla f(\overline{x})$ soit nul et que $\nabla^2 f(\overline{x})$ soit une matrice définie positive, on accepte que $\overline{x}$ constitue un minimum local de f. Divers algorithmes, tels la méthode de Newton ou celle des régions de confiance [DS83], permettent de converger rapidement, à partir de quelque point $x_0 \in \mathbb{R}^n$, vers un point x^* satisfaisant aux conditions nécessaires d'optimalité.

Un aspect capital de ces algorithmes consiste en le calcul du gradient et du hessien directionnel de $f(x)$. D'une part, cette tâche peut être déléguée au modeleur de la fonction objectif. Bien qu'elle permette de réaliser les calculs différentiels en un temps optimal, cette approche est pénible à mettre en oeuvre et sujette à l'erreur humaine. D'autre part, la dérivée directionnelle de $f(x)$ peut être estimée à partir du seul programme d'évaluation de la fonction $\mathcal{P}_f$ par les techniques de différences finies. Cependant, cette solution est onéreuse en termes de temps de calcul et, sous certaines circonstances, souffre d'erreurs numériques d'annulation significatives. Les techniques de *différentiation automatique* visent à réunir les avantages et à contrer les inconvénients de ces approches. En effet,

les travaux dans ce domaine ont montré qu'il s'agit d'une méthode efficace et précise de calcul qui n'exige en entrée que le programme $\mathcal{P}_f$ [Gri89].

Conventions sur la notation

Tout au long de cet ouvrage, une fonction vectorielle $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ associe à chaque vecteur colonne de $\mathbb{R}^n$ un vecteur colonne de $\mathbb{R}^m$. De ce fait, le jacobien d'une telle fonction, noté $\nabla f(x)$, consiste en une matrice de dimensions $m \times n$. Ainsi, si les composantes de l'image de $f(x)$ sont séparables[1], chaque ligne de $\nabla f(x)$ constitue le gradient de cette composante. D'une manière cohérente, on définit le gradient d'une fonction réelle $g : \mathbb{R}^n \rightarrow \mathbb{R}$, aussi noté $\nabla g(x)$, comme une fonction vectorielle $\mathbb{R}^n \rightarrow \mathbb{R}^n$ qui associe à chaque élément du domaine de $g(x)$ un *vecteur ligne* de $\mathbb{R}^n$.

Organisation du mémoire

Cet ouvrage regroupe d'abord deux articles relatant nos expériences sur le sujet, ainsi que leurs résultats. Dans le premier article, nous décrivons l'implantation d'un outil de différentiation automatique dans l'environnement de calcul numérique Scilab [Sci03]. Dans le second, nous expliquons comment une extension des outils traditionnels de différentiation automatique permet d'améliorer la robustesse numérique des algorithmes d'optimisation différentiables sans contrainte. Ces articles sont suivis d'un ultime chapitre où nous faisons état de plus amples expériences menées sur cette extension. Quelques perspectives sur les futures applications des outils développés seront esquissées pour conclure.

[1] Les composantes d'un vecteur de l'image de $f(x)$ sont séparables si elles peuvent être définies comme m fonctions de $\mathbb{R}^n$ à $\mathbb{R}$ indépendantes.

CHAPITRE 1

IMPLANTATION D'UN OUTIL DE DIFFÉRENTIATION AUTOMATIQUE

1.1 Préambule

Le design et l'implantation d'un outil logiciel de différentiation automatique constituent des problèmes riches et complexes. La solution de ces problèmes est d'ailleurs très sensible au choix de la plate-forme cible de cette implantation, d'où d'intéressants défis. Cet article rapporte nos efforts de réalisation de cette tâche sur la plate-forme de traitement numérique Scilab [Sci03]. Cet environnement se présente comme un langage de programmation interprété fonctionnel supportant la définition de types abstraits de données et la surcharge des opérateurs.

D'une part, j'ai été le principal architecte et unique programmeur des outils de différentiation automatique mis en oeuvre dans le cadre de ce projet, regroupés en un logiciel nommé SCIAD. Au moment d'écrire ces lignes, je suis toujours responsable du développement et de l'entretien de SCIAD. D'autre part, je suis l'auteur principal de cet article, l'ayant complètement structuré et rédigé. Le coauteur a contribué à la recherche littéraire et à la critique de ses itérations.

Comme mentionné dans l'article, des librairies de différentiation automatiques existent déjà pour de nombreux langages : les plus connues sont ADOL-C [GJM+99], pour le langage C++ et ADIFOR [BCH+98], pour le langage FORTRAN. En fait, SCIAD n'est même pas le premier outil de différentiation automatique pour l'environnement Scilab : une librairie nommée `diffcode` [JS02] calcule la dérivée directionnelle d'une fonction $\mathbb{R}^n \rightarrow \mathbb{R}^m$ à l'aide de la différentiation automatique en postordre (*forward*). Cependant, cette librairie agit sur un graphe d'évaluation implicite et ne permet pas d'obtenir des dérivées d'ordre supérieur. Notre effort dans l'implantation de SCIAD consiste à offrir une librairie de différentiation automatique plus complète et plus flexible en termes de manipulation du graphe d'évaluation d'une fonction, qui permet l'expérimentation sur des outils auxiliaires et qui simplifie le développement d'extensions.

Soumis pour publication à *Scilab-2004 International Conference.*

Object-oriented implementation of an automatic differentiation toolkit in a high-level numerical processing functional language

Rapport de recherche 1
Département d'informatique
Faculté des sciences
Université de Sherbrooke

Benoit Hamelin[*]
Jean-Pierre Dussault[†]

Abstract

We present the SCIAD toolkit, an automatic differentiation package for the Scilab [21] numerical processing environment. We propose an object-oriented design for the representation of computation graphs and information propagation and discuss the challenges one faces when implementing such a design in a functional language.

Keywords. Automatic differentiation, object-oriented design, design patterns, non-linear programming, functional languages, Scilab.

[*]e-mail: benoit@benoithamelin.com
[†]e-mail: jean-pierre.dussault@usherbrooke.ca

Introduction

Automatic differentiation (AD) is a versatile and efficient technique to compute the derivative $\nabla f(x)$ of any function $f : \mathbb{R}^n \to \mathbb{R}^n$ using a representation of the program that computes the function. This technique has been developed since the 1960's ([11], [22]) and its tools are now implemented in many popular software packages that deal with numerical optimization, resolution of differential equations, sensitivity analysis... Among other, we mention ADOL-C [16] (C++) and ADIFOR [6] (FORTRAN) as seminal implementations of the technique.

We propose yet another implementation of automatic differentiation software, the SCIAD package, devised for the Scilab [21] numerical processing environment. Being aimed at experimentation with optimization procedures, this toolkit acquires a representation of a function evaluation at runtime and gives tools to transform this representation in the derivative of the function, as well as other related results.

This paper is organized as follows. We first recall the basic methods of automatic differentiation. We then illustrate how these techniques suit the definition of a software problem well solved using an appropriate object-oriented design pattern. We push this design to describe the high-level AD tools we're interested in. Finally, we discuss issues that regard implementation of these tools using a functional programming language and formulate our solutions to these issues in the Scilab environment.

1 Principles of automatic differentiation

In context of differentiable optimization, consider the problem of evaluating the gradient $\nabla f(x)$ and Hessian $\nabla^2 f(x)$ of a multi-variate real function $f : \mathbb{R}^n \to \mathbb{R}$. This problem is often deferred to the user of the optimization package, being obliged to give programs that evaluate $\nabla f(x)$ and $d^t \nabla^2 f(x)$ (second-order derivative in direction d) along that which evaluates $f(x)$. This solution is often unacceptable, since for functions of even mild complexity, the expression of derivatives can become hardly manageable. Even using symbolic differentiation programs (like Maple), the user becomes owner and maintainer of the derivative code, a situation that is prone to errors and time-consuming. In addition, experimentation with optimization techniques that involve the evaluation of higher-order derivatives is impractical, especially if the derivation order is adaptive.

An automatic calculation of directional derivatives can be obtained using finite differences. For instance,

$$\nabla f(x)d = \lim_{\epsilon \to 0} \frac{f(x + \epsilon d) - f(x)}{\epsilon} \approx \frac{f(x + \mathcal{E} d) - f(x)}{\mathcal{E}}$$

for $\mathcal{E}$ being a suitably small number. However, as explained in [10], close to a local minimum of $f(x)$, cancellation errors occur to such an extent as to render the finite differences inaccurate. Moreover, whereas the code to acquire the gradient of some function can be made to cost no more than evaluating the function itself, gathering the whole gradient vector implies $n + 1$ evaluations of the function. Thus, using finite differences to evaluate the gradient has a runtime cost that grows linearly in the problem dimension n as compared to the runtime cost of evaluating the function.

1.1 Forward mode

Hence the need for an accurate and efficient procedure to calculate derivatives. We may observe that complex functions we work with can be reduced to an ordered calculus of primitive operators, the so-called *standard functions*, applied in succession to previous intermediate results, starting with independent variable x. Examples are the arithmetic operators ($+$, $-$, $\times$, $\div$, exponentiation), as well as often-seen operators such as the exponential, the logarithm, circular trigonometry's, etc. When one determines the derivative of a function defined with such building blocks using paper and a pencil, one applies the rules bound to each operator and binds the results using the *chain rule* of derivation:

$$\frac{d}{dx}f(g(x)) = \frac{df(g(x))}{dg(x)} \times \frac{dg(x)}{dx}$$

This process is directly translated to software. Indeed, a computer program that computes $f(x)$ for a given x is a composition of standard functions and operators. This composition may be represented using a *computation graph* (figure 1.1), a generalized abstraction of syntax trees used by compilers and interpreters [17]. Such a representation shows which primitive operations are involved, as well as a partial ordering of the calculations implied by the flow of intermediate results, starting at the predecessor-less vertexes (independent variables and constants). Name each vertex of a graph z_1 to z_r and associate to each an operator $f_k(z_i, z_j)$ from the defined closed set of standard functions. We then can derive a chain rule fit to propagating intermediate derivative results $\frac{dz_k}{dx}$ along the edges of the graph:

$$\frac{dz_k}{dx} = \frac{\partial f_k}{\partial z_i} \times \frac{dz_i}{dx} + \frac{\partial f_k}{\partial z_j} \times \frac{dz_j}{dx} \tag{1.1}$$

This rule is the basis of the *forward mode of automatic differentiation*; its intermediate quantities are named *tangent derivatives*. For a function $f : \mathbb{R}^n \to \mathbb{R}^n$ and a matrix M, the process yields $\nabla f(x)M$. One initiates it by "seeding" the predecessor-less vertexes. With high-level primitives, these vertexes are either bound to sub-vectors x_S[1] of the

[1] S is a sequence of indexes in the range from 1 to n that denote elements of independent variable vector x. For instance, with $S = 3$, we have $x_S = x_3$; with $S = [3, 4, 1]$, we have $x_S = [x_3, x_4, x_1]^t$.

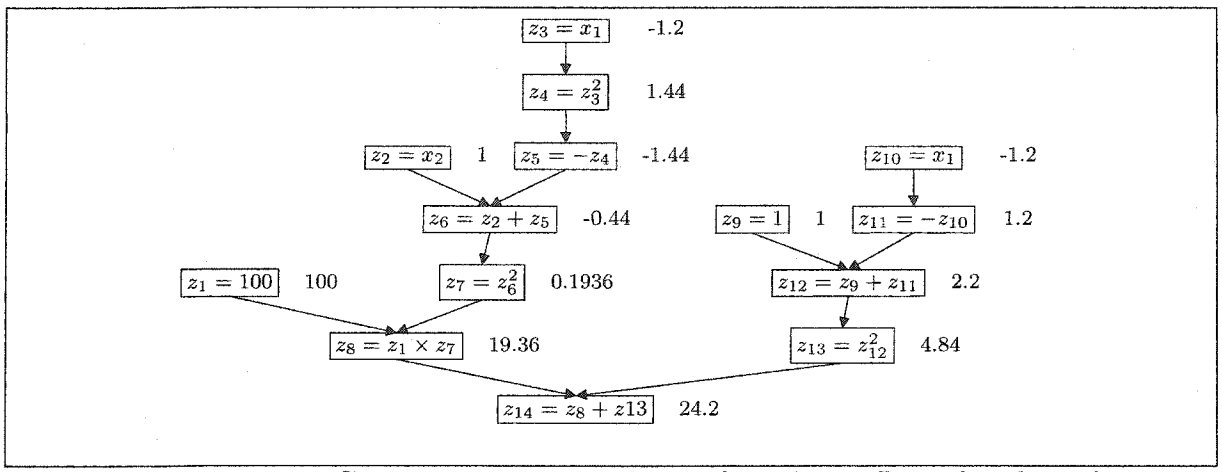


Figure 1.1: Computation graph of the Rosenbrock function, $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$. The numbers besides the vertexes denote how the graph can be used to evaluate f at $x = (-1.2, 1)^t$ by propagating intermediate results.

independent variable $x \in \mathbb{R}^n$ or to constant matrices.[2] The latter needs no seeding; for the former, we know the Jacobian of the identity function $\mathcal{F}(x) = x$ is I_n. Hence, $\nabla\mathcal{F}(x)M = I_nM = M$. Let us now consider a vertex bound to the sub-vector x_S of the independent variable. For S being a sequence of q indexes, the Jacobian of $\mathcal{F}(x_S) = x_S$ is defined as the $q \times n$ matrix, for which row i is the canonical vector e_{S_i}.[3] The tangent derivatives for the independent variables then are $\frac{dx_S}{dx} = \nabla\mathcal{F}_S(x)M$. From these quantities, rule 1.1 is used to propagate the seeds along the edges through intermediate vertexes, onward to the unique successor-less vertex. The tangent derivative computed for this vertex is exactly $\nabla f(x)M$. An example with the Rosenbrock function can be examined at figure 1.2.

1.2 Backward mode

Since the derivative is computed using exact formulas, the accuracy problem is solved. However, figure 1.2 shows that the intermediate results of the forward mode of automatic differentiation are $\mathbb{R}^n$ vectors. Since evaluating $f(x)$ costs a traversal of the computation graph, which runtime cost we'll call $C[f(x)]$, the cost of evaluating $\nabla f(x)$ still is $O(nC[f(x)])$. Let us then consider a variant of the chain rule of derivation that propagates intermediate values from the final result of the computation of $f(x)$ to the independent variable vertexes: this is the *backward mode of automatic differentiation.* The chain rule

[2]Which, by isomorphism, can be scalars and vectors.

[3]Our notation expresses this clearly: tangent derivatives at independent variable nodes are symbolized as $\frac{dx_S}{dx}$.

At z_1: vertexes bound to constants have no tangent.

At z_2: our independent variable is $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$; we obtain $x_2 = \begin{bmatrix} 0 & 1 \end{bmatrix} x$. Then, having direction matrix $M = I_2$, we seed $\dot{z}_2 = \begin{bmatrix} 0 & 1 \end{bmatrix} M = \begin{bmatrix} 0 & 1 \end{bmatrix} I_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$.

At z_3: $\dot{z}_3 = \begin{bmatrix} 1 & 0 \end{bmatrix} I_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

At z_4:
$$\begin{aligned} \dot{z}_4 &= \frac{\partial(z_3^2)}{\partial z_3} \times \dot{z}_3 \\ &= 2z_3\dot{z}_3 \\ &= 2(-1.2)\begin{bmatrix} 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -2.4 & 0 \end{bmatrix} \end{aligned}$$

At z_5:
$$\begin{aligned} \dot{z}_5 &= \frac{\partial(-z_4)}{\partial z_4} \times \dot{z}_4 \\ &= -\dot{z}_4 \\ &= \begin{bmatrix} 2.4 & 0 \end{bmatrix} \end{aligned}$$

At z_6:
$$\begin{aligned} \dot{z}_6 &= \frac{\partial(z_2 + z_5)}{\partial z_2} \times \dot{z}_2 + \frac{\partial(z_2 + z_5)}{\partial z_5} \times \dot{z}_5 \\ &= \dot{z}_2 + \dot{z}_5 \\ &= \begin{bmatrix} 2.4 & 1 \end{bmatrix} \end{aligned}$$

At z_7:
$$\begin{aligned} \dot{z}_7 &= 2z_6\dot{z}_6 \\ &= 2(-0.44)\begin{bmatrix} 2.4 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -2.112 & -0.88 \end{bmatrix} \end{aligned}$$

At z_8:
$$\begin{aligned} \dot{z}_8 &= \frac{\partial(Kz_7)}{\partial z_7} \times \dot{z}_7 \\ &= K\dot{z}_7 \\ &= 100\begin{bmatrix} -2.112 & -0.88 \end{bmatrix} \\ &= \begin{bmatrix} -211.2 & -88 \end{bmatrix} \end{aligned}$$

At z_9: as for vertex z_1, we associate no tangent to constants.

At z_{10}: $z_1 0 = z_3 \Rightarrow \dot{z}_{10} = \dot{z}_3 = \begin{bmatrix} 1 & 0 \end{bmatrix}$

At z_{11}:
$$\begin{aligned} \dot{z}_{11} &= -\dot{z}_{10} \\ &= \begin{bmatrix} -1 & 0 \end{bmatrix} \end{aligned}$$

At z_{12}:
$$\begin{aligned} \dot{z}_{12} &= \frac{\partial(K + z_{11})}{\partial z_{11}} \times \dot{z}_{11} \\ &= \dot{z}_{11} \\ &= \begin{bmatrix} -1 & 0 \end{bmatrix} \end{aligned}$$

At z_{13}:
$$\begin{aligned} \dot{z}_{13} &= 2z_{12}\dot{z}_{12} \\ &= 2(2.2)\begin{bmatrix} -1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -4.4 & 0 \end{bmatrix} \end{aligned}$$

At z_{14}:
$$\begin{aligned} \dot{z}_{14} &= \dot{z}_8 + \dot{z}_{13} \\ &= \begin{bmatrix} -211.2 & -88 \end{bmatrix} + \begin{bmatrix} -4.4 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -215.6 & -88 \end{bmatrix} \\ &= \nabla f(x) I_2 \end{aligned}$$

Figure 1.2: Calculations towards the computation of $\nabla f(x) I_2$, for the Rosenbrock function (see figure 1.1) in the forward mode of automatic differentiation. We remark that tangent derivatives (which, for vertex z_i, is noted $\dot{z}_i$), are propagated from predecessors to successors.

At z_{14}: the process is seeded with the weight matrix $1 = \tilde{z}_{14}$. We then can compute the adjoints of predecessor vertexes.
$\tilde{z}_8 = \frac{\partial(z_8+z_{13})}{\partial z_8} \times \tilde{z}_{14} = \tilde{z}_{14} = 1$
$\tilde{z}_{13} = \frac{\partial(z_8+z_{13})}{\partial z_{13}} \times \tilde{z}_{14} = \tilde{z}_{14} = 1$

At z_{13}: $\tilde{z}_{12} = \frac{\partial(z_{12}^2)}{\partial z_{12}} \times \tilde{z}_{13} = 2z_{12}\tilde{z}_{13} = 2(2.2)(1) = 4.4$

At z_{12}: we only compute $\tilde{z}_{11}$, since z_9 is bound to a constant, which aren't bound to an adjoint.
$\tilde{z}_{11} = \tilde{z}_{12} = 4.4$

At z_{11}: $\tilde{z}_{10} = \frac{\partial(-z_{10})}{\partial z_{10}} \tilde{z}_{11} = -\tilde{z}_{11} = -4.4$

At z_{10}: $\tilde{z}_{10}$ is one term of the computation of $\frac{\partial f}{\partial x_1}$. Hence, we project $\tilde{z}_{10}$ using the variable projection vector ($x_1 = \begin{bmatrix} 1 & 0 \end{bmatrix} x$), to obtain a term of the computation of $1 \times \nabla f(x)$:
$\mathcal{T}_1 = \tilde{z}_{10} \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} -4.4 & 0 \end{bmatrix}$.

At z_9: nothing to do.

At z_8: we only calculate $\tilde{z}_7$, since z_1 is bound to a constant.
$\tilde{z}_7 = \frac{\partial(z_1 \times z_7)}{\partial z_7} \times \tilde{z}_8 = z_1\tilde{z}_8 = 100(1) = 100$

At z_7: $\tilde{z}_6 = 2z_6\tilde{z}_7 = 2(-0.44)(100) = -88$

At z_6: $\tilde{z}_2 = \tilde{z}_5 = \tilde{z}_6 = -88$

At z_5: $\tilde{z}_4 = -\tilde{z}_5 = 88$

At z_4: $\tilde{z}_3 = 2z_3\tilde{z}_4 = 2(-1.2)(88) = -211.2$

At z_3: we define another gradient term here.
$\mathcal{T}_2 = \tilde{z}_3 \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} -211.2 & 0 \end{bmatrix}$

At z_2: the last gradient term is computed here; we obtain x_2 by the projection $x_2 = \begin{bmatrix} 0 & 1 \end{bmatrix} x$. Hence,
$\mathcal{T}_3 = \tilde{z}_2 \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -88 \end{bmatrix}$

At z_1: nothing to do.

Finally: we reconstitute the gradient by adding the terms computed at independent variable vertexes.
$1\nabla f(x) = \mathcal{T}_1 + \mathcal{T}_2 + \mathcal{T}_3 = \begin{bmatrix} -215.6 & -88 \end{bmatrix}$

Figure 1.3: Starting again from the computation graph of figure 1.1, we propagate adjoint derivatives (noted, for vertex z_i, $\tilde{z}_i$) using rules derived from equation 1.2, leading to the computation of $1 \times \nabla f(x)$.

that supports this mode propagates *adjoint derivatives*; for a vertex $z_k = f_k(z_i, z_j)$, this rule is stated as follows:

$$\frac{df}{dz_i} = \frac{\partial f_k}{\partial z_i} \times \frac{df}{dz_k} \qquad \frac{df}{dz_j} = \frac{\partial f_k}{\partial z_j} \times \frac{df}{dz_k} \tag{1.2}$$

This process is set up to yield $N\nabla f(x)$ by seeding the successor-less vertex with N.[4] This seed is propagated to predecessors, opposite to the edges of the graph onward to the predecessor-less vertexes. Of these vertexes, we ignore those that are bound to constants. The adjoint derivatives propagated to the independent variable vertexes carry a partial derivative result with respect to the variable bound to the vertexes. One then reconstitutes the partial derivatives that compose $N\nabla f(x)$ by adding adjoint derivatives bound to the same variable. Listing 1.3 illustrates the backward derivation of the Rosenbrock function evaluated at $(-1.2, 1)^t$.

A quick look at figure 1.3 is enough to note that the adjoint derivatives are scalar quantities, as opposed to the tangent derivatives propagated by the forward mode. This is

[4] We remark that, contrary to the forward mode, the weight matrix N used for seeding the backward mode selects *rows* of the Jacobian, rather than columns. It is obviously expected to have m columns. One may also observe that, when deriving gradients, because of the symmetry of the Hessian of $\mathcal{C}^2$ functions, both the forward and backward modes may be conveniently seeded to obtain directional second-order derivatives.

a manifestation of the performance advantage of the backward mode over the forward mode. Works by many authors in the 1980's has exhibited that the backward mode of AD has a runtime cost proportional to $C[f(x)]$ and that the cost of calculating both $f(x)$ and $\nabla f(x)$ using this mode has a cost bounded by $5C[f(x)]$. For a further introduction to the theory, the techniques and the performance analysis of automatic differentiation, we point the reader to [1, 20, 14, 15].

1.3 Tools inspired by AD

It is worth mentioning that derivation of some computation graph can, instead of resulting in a numerical evaluation of the derivative, yield in turn the computation graph of the derivative function. Using the appropriate method of graph construction (see section 2.1), one can obtain the graph of $g(x) = \nabla f(x)d$, which in turn is a $\mathbb{R}^n \to \mathbb{R}^m$ function that can be derived using the very same tools. Hence, an arbitrary order of derivation can be obtained, provided that the submitted function have a $\mathbb{R}^m$ image. Some interpreted environments even allow for a runtime decision of the derivation order. This feature is used, among others, by Dussault, with his high-order Newton method variants [9].

Other results than derivatives can be obtained by using some intermediate result propagation rule along the traversal of the computation graph. For example:

- Some applications obtain a Taylor polynomial approximation of $f(x)$ of order n using a forward traversal of the computation graph; see [4, 3, 2].
- For the resolution of large equation systems, some methods gain runtime efficiency by a clever use of the sparsity pattern of the Jacobian of the system, which pattern can be aligned by forward mode propagation; see [8, 7].
- In unconstrained optimization, evaluation of descent between successive iterates becomes inaccurate when one gets close to a local minimum. Finite difference $f(x+d) - f(x)$ can however be accurately evaluated using a forward propagation of intermediate computations; see [10].

Software packages to implement each of these procedures have this in common that they imply a traversal of the computation graph and a propagation of intermediate results using some rule.

2 Object-oriented design of an AD toolkit

Writing AD software then is a two-fold problem. First, one must build the computation graph of the program that evaluates $f(x)$. Next, one must traverse the computation graph, propagating intermediate results. The initialization of the process and the direction of this traversal depends on whether the traversal should be forward (from predecessor-less nodes to the unique successor-less node) or backward (from the successor-less node to the predecessor-less nodes).

2.1 Construction of the computation graph

This sub-problem has been largely examined in past literature ([5], [19]). There are two main approaches. The first is that of *source transformation*. It implies writing a precompiler that gathers the computation graph by parsing the code of $f(x)$. The output of the processing of the graph is a new module of code, which can be compiled into the software that uses it along the original. This approach is used, among others, by ADIFOR [6], an AD package for the FORTRAN language. The advantage of this approach is that the precompiler "sees" all the function evaluation code and the context in which it is called. This gives flexibility to the AD software in terms of the choice of the derivation strategy, in addition to an understanding of the control structures that articulate the code and define eventual ruptures of smoothness of f. It is also the approach that generates the least runtime overhead to the computation. However, every result expected from processing of the computation graph must be statically defined at compile time, disabling the dynamic acquisition of, for instance, higher-order derivatives.

The second approach implies *operator overloading*. It involves the development of an abstract data type T that redefines the standard operators of the implementation language. This type is implemented in such a manner that the code that evaluates $f(x)$ can be called with an argument $t(x)$ of type T, so the statements of the code, instead of "building" a numerical evaluation of f at point x, build the computation graph of f at this point. This is the approach used by ADOL-C [16], an AD library for the C++ language, and ADMAT [8], a tool for the Matlab numerical processing environment. The advantage here is that the language compiler or interpreter handles most of the complexity of the construction process. Moreover, the implementation can be made so the graph is an explicit data structure, which can be handled dynamically at runtime by AD tools; this flexibility obviously comes with a cost in terms of runtime overhead. In addition, for most languages that allow operator overloading, the control structures (such as conditionals and loops) cannot be overloaded, to the extent that they become transparent to the graph construction. Hence, discontinuities of f are undetectable to the AD software and the burden of managing these discontinuities is deferred to the user.

SCIAD is loosely inspired from ADMAT, given the similarities between Matlab and Scilab; it is not, however, a port nor a rewrite of ADMAT to Scilab. SCIAD uses operator overloading in the fashion detailed above, which yields the computation graph as an explicit data structure suitable for experimentation and, among other requirements we had set, dynamical choice of derivation order. This goal of the development of SCIAD justifies the fact that we always make explicit the representation of the computation graph, whereas other AD toolkits process forward traversals by tapping the implicit walk on the computation graph that operator overloading realizes. Some limited empirical experimentation has shown that traversing the graph is less expensive than building it on our software platform, and our usage pattern of AD tools involves performing various bookkeeping operation on the computation graph once it is set up.

2.2 Propagation of intermediate results through the graph

Consider now the problem of calculating results such as derivatives by traversing the graph. The order of traversal is dictated by an abstract propagation rule, which can be instantiated into a specific formula for each type of vertex we expect in the graph. A vertex is either associated to an independent variable, to a constant or to a standard unary or binary operator. In other words, we want to traverse a complex structure of similar yet specialized objects, updating state elements after the visit of each node.

This specification exactly matches that of the implementation problem solved by the object-oriented *Visitor* design pattern [12]. We have this data structure made of objects that share a common parent, hence that share a common functionality. Let us make this common functionality the ability to be *visited* by some object, which we will call the *visitor*. Such a visitor has the tools required to compute some information based on intermediate data it acquires all through its traversal of the structure. Figure 1.4 illustrates the UML structure of this design applied to automatic differentiation, while figure 1.5 articulates the collaboration between the elements.

Many factors justify this design. From a maintainability point of view, each operation can be implemented in its own software module, instead of across software modules for all objects of the vertex hierarchy. This also makes the addition of new operations an easier task. Moreover, lots of information and decisions are deferred to the visitor: it can accumulate and synthesize state, as well as decide the order in which the vertexes of the graph are visited. One can see that this design has the flaws of its forces: whereas adding new operations is easy, adding new vertexes implies the modification of all visitors. However, since we can decide in advance what set of standard operations we wish to handle, this disadvantage is moot.

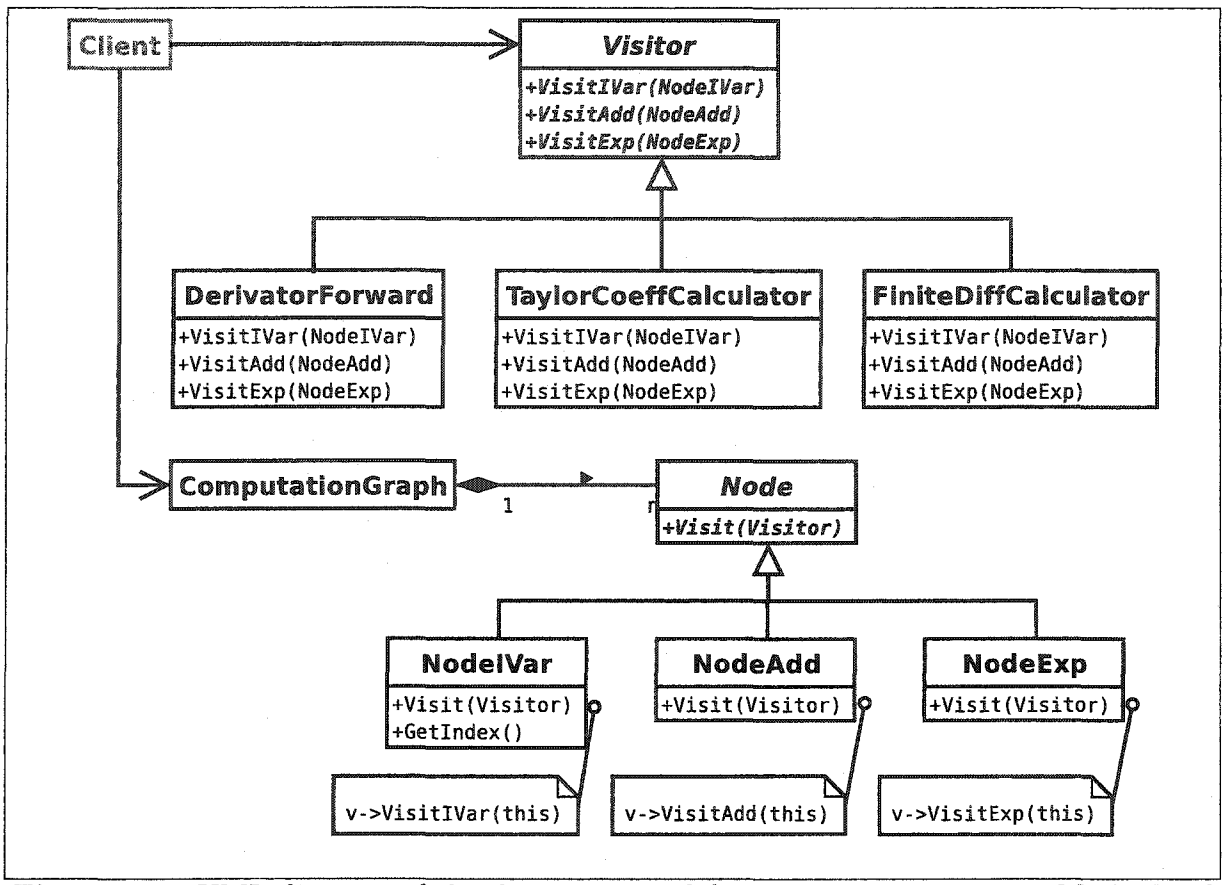


Figure 1.4: UML diagram of the design inspired from the visitor pattern. Methods of the visitor classes are only partially listed for readability purposes.

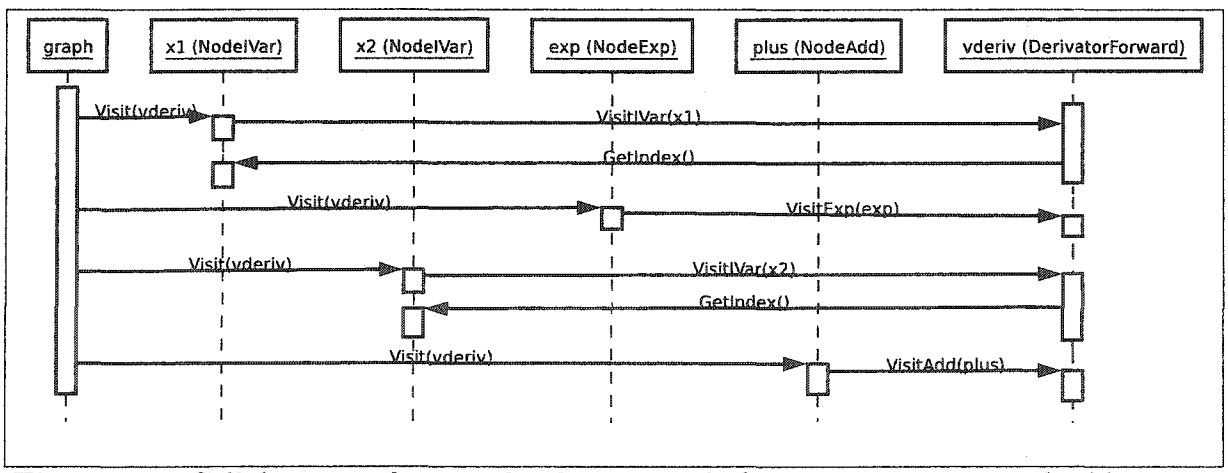


Figure 1.5: Collaboration diagram for the traversal of the computation graph of function $f(x) = e^{x_1} + x_2$. Time flows down the vertical axis; the rectangles represent the lifetime of method calls. Horizontal arrows denote the order of method invocation.

3 Implementation in a functional language

The purpose of this section is to delve into technical details about the implementation of the design proposed in section 2 with a functional programming language. In particular, we use the language devised in context of the Scilab [21] numerical processing environment, which gives constructs to easily manage and manipulate mathematical objects such as vectors, matrices and functions.

3.1 Structure of the computation graph

A peculiar feature of many functional languages is the absence of explicit *reference* constructs for data objects. In other terms, we have no "pointer" to refer to data elements dynamically allocated. Semantics of such languages would rather give control over the objects themselves instead of the memory zones that support them. In this line of thought, they offer high-level structure-building tools. In Scilab, these tools are *lists*, linear direct-access abstract containers, and *matrices*, two-dimensional arrays of numbers. Abstract data types are implemented using named *typed lists* (`tlists` and `mlists`). Primitive operators can then be overloaded to handle such typed lists.

As mentioned in section 2.2, each vertex of the graph is bound to either an independent variable, a constant or a unary or binary standard operator. The ease of manipulation of vectors and matrices in Scilab allows us to go beyond the association of simple scalars to each vertex: independent variables are scalars or vectors, constants are either scalars, vectors or matrices and standard operators fold such operands in a scalar, vector or matrix intermediate result. This feature keeps the the computation graphs small in size. For example, for a tool that only binds scalars to vertexes, the computation graph that represents a simple inner product is a string of scalar multiplications. Its size is then proportional to the length of the operand vectors. In contrast, for a tool such as ours, the inner product is simply represented by a three-nodes tree, the size of which does not depend on the operand vector length. This said, we note that the number of operations implied in computing the inner product is not diminished through this more concise representation. In our environment where we give the explicit representation of the computation graph, the economy lies in the storage of a lesser number of vertexes and edges to express the operation.

We obviously need to store some state for every vertex in a computation graph, so each *vertex* is implemented by a typed list. A graph being the coupling of a set of vertexes to a set of edges binding these vertexes, we make *computation graphs* typed lists with two members: the first is a list of vertexes, each having a unique index; the second is a $2 \times p$

matrix, for which each column $\begin{bmatrix} i \\ j \end{bmatrix}$ describes an edge using indexes in the vertex list. Edge orientation is from i to j.

One could have considered storing edge information in vertex lists. However, the construction of the computation graph using operator overloading involves the concatenation of lists of vertexes of two sub-graphs being composed together with some binary operation. Hence, vertexes being put at the end of the resulting list must be re-indexed at each new composition. It then appears more convenient to manage all indexing in a single matrix field of the computation graphs.

It may be conjectured that this implementation is less efficient than a reference-based data structure, since knowing the predecessors or the successors of an arbitrary vertex implies at least a binary lookup through the edge matrix. However, for the applications of AD we have considered, the visit of some vertex either implies information setup by its predecessors or propagates information to them. Hence, the edge matrix can be sorted by successor indexes, leading to a constant-time access to predecessor indexes if the visitor stores information as where it can find them as the traversal progresses.

3.2 Implementation of the Visitor design pattern

The design put forth in section 2.2 relies on the use of an object-oriented computing paradigm, in particular the ability to define *abstract* methods (so-called "virtual" methods of the C++ language) for objects. However, our functional implementation platform offers no explicit construct of this sort. Yet, functional languages allow us to manipulate functions as first-order objects. Hence, they can be set as values of the fields of a typed list.

This way, a *visitor* is implemented as a typed list that has a field named after each standard operator and function we overload. These fields are set to the appropriate functions when a visitor is created: for instance, when a `Derivator` (a visitor that calculates the derivative of a function in the backward mode) instance is created, its field `VisitIVar` is set as the function named `Derivator_VisitIVar`, its field `VisitAdd` is set as `Derivator_VisitAdd`, and so on. Each such visit function has an identical interface, so the nature of a vertex being visited is transparent to the entity responsible for processing the traversal. While this task could have been deferred to the visitor, since all our traversal operations are either forward (post-order) or backward (any order that visits a parent before its children – we have implemented breadth-first), we have made the traversal articulation a method of the computation graph. In this setup, some operation applied to the graph internally invokes the appropriate traversal articulation (forward or backward) with the visitor devised for the job; the end result of the operation

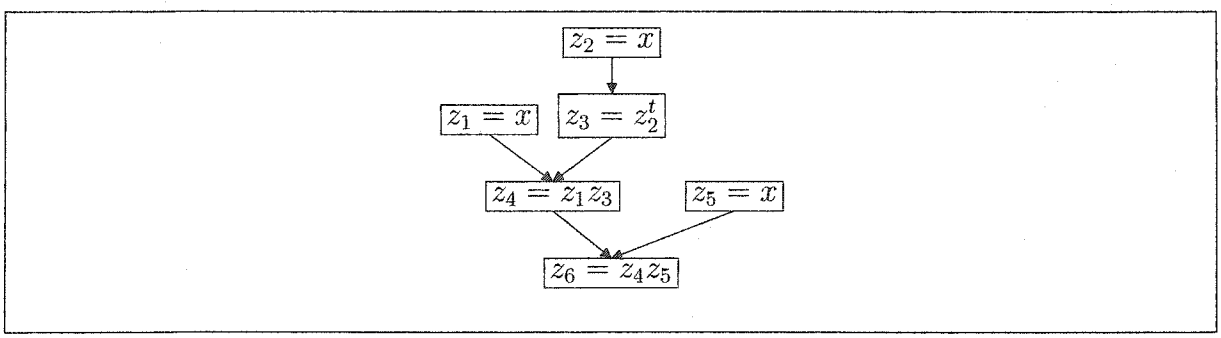


Figure 1.6: Computation graph of function $g(x) = (xx^t)x$.

At z_6: the process is seeded using weight matrix $I_n = \tilde{z}_6$. Let us first compute $\tilde{z}_4$. Since the intermediate function evaluation at z_4 is a matrix (xx^t), this adjoint is a tensor, for which each frame i is line i of $\tilde{z}_6$ multiplied by z_5^t. Hence, $\tilde{z}_4(i) = e_i x^t$. Moreover, $\tilde{z}_5 = \tilde{z}_6 z_4 = I_n x x^t = x x^t$.

At z_5: this is a non-projected independent variable node, such that its adjoint is in itself a term of the Jacobian computation.

At z_4: Let us first consider $\tilde{z}_1$. Vertex z_4 is bound to an outer product, for which the first operand z_1 is a column vector. Its adjoint is then expected to be a matrix for which each row i is the product of second operand z_3 and $\tilde{z}_4(i)^t$. Then,

$$\tilde{z}_1 = \begin{bmatrix} x^t x e_1^t \\ x^t x e_2^t \\ \vdots \\ x^t x e_n^t \end{bmatrix} = x^t x I_n.$$

Second, we compute $\tilde{z}_3$: the second operand z_3 of the outer product is a row vector. Its adjoint is also a matrix for which each column i is the product of $\tilde{z}_4(i)^t$ and first operand z_1. Thus,

$$\begin{aligned} \tilde{z}_3 &= \begin{bmatrix} x e_1^t x & x e_2^t x & \cdots & x e_n^t x \end{bmatrix} \\ &= \begin{bmatrix} x_1 x & x_2 x & \cdots & x_n x \end{bmatrix} \\ &= x x^t. \end{aligned}$$

At z_3: $\tilde{z}_2 = \tilde{z}_3^t = x x^t$

At z_2 and z_1: idem as z_5.

Finally: $I_n \nabla g(x) = \tilde{z}_5 + \tilde{z}_2 + \tilde{z}_1 = x^t x I_n + 2 x x^t$, as expected.

Figure 1.7: Backward-mode derivation of function $g(x)$ for which figure 1.6 displays the computation graph. Management of the tensorial adjoint derivatives using lists of *frames* is shown (frame j of the adjoint of vertex z_i is noted $\tilde{z}_i(j)$). Further formulas for the computation of adjoint derivatives are given in appendix A.

is obtained from some field of the visitor when the traversal process returns.

3.3 The backward mode derivation visitor

Automatic derivation is rather tricky to implement when computation graphs may contain operations over high-level objects such as vectors and matrices. Let us consider the particular case of the visitor that computes backward mode derivation. We specified that we allowed the derivation of $\mathbb{R}^n \to \mathbb{R}^m$ functions, for which the derivative is a $\mathbb{R}^{m \times n}$ matrix; in the particular case of $m = 1$, the gradient is a $\mathbb{R}^n$ row vector. Take the simpler case where the computation graph of a function $f(x)$ of interest is a tree; we name each node z_i, with $i \in \{1, 2, \cdots |V|\}$. As mentioned in section 1.2, the process is

Intermediate result	Adjoint structure
Scalar	Vector
Vector	Matrix
Matrix	Three-dimensional tensor

Table 1.1: Structure of the adjoint bound to each allowed intermediate result in a computation graph.

seeded by setting $\frac{df}{dz_{|V|}}$ to a $p \times m$ row selection matrix; for instance, using I_m will yield the full Jacobian of $f(x)$. Traversal of the computation graph using the backward mode derivation rule is then expected to propagate to nodes associated to a sub-vector x_S (of q components) of the independent variable $x \in \mathbb{R}^n$ a partial Jacobian result supported by a $p \times q$ matrix.

This proposition is generalized to all vertexes of the computation graph, as displayed in table 1.1. In our implementation, we limit vertexes to never be bound to an intermediate result more complex than a matrix. Products and transposition are harder to define for tensors than for two-dimensional numerical objects. Hence, we represent tensorial adjoint derivatives with lists of matrices, each of these matrices being a *frame* along the tensor's third dimension. Vertexes that have such adjoints are obviously associated to adjoint propagation formulas that handle this frame-by-frame representation. For instance, consider the function $g(x) = (xx^t)x$, with $x \in \mathbb{R}^n$; figure 1.6 shows its computation graph and figure 1.7 details its derivation using the backward mode.

The vertex-by-vertex construction of the graph of the derivative eventually composes the frames of tensorial adjoints with primitives that take matrix operands. Closing the image of all primitive operations allowed in the computation graph to the set of matrices allows for successive derivations, leading to a dynamical choice of the order of derivation. Indeed, this visitor, instead of propagating numerical adjoints, can cumulate the computation graphs of the adjoints, which can be summed into the computation graph of the Jacobian of some function. For $\mathbb{R}^n \to \mathbb{R}$ functions, this yields the computation graph of the gradient, which can be derived into the Hessian with no modification. For matrix Jacobians and Hessians, the operator overloading scheme devised to build the original computation graph can still be used to multiply the Jacobian or the Hessian by some direction vector, yielding a new vector function, which can be derived in turn.

3.3.1 Running time and memory usage complexity

Let us name $V[f(x)]$ the set of vertexes of the computation graph of $f(x)$. Since one node is added to this graph for each elementary operation composed into the process of

evaluating f at point x, we have $|V[f(x)]| \in O(C[f(x)])$.

Consider the process of traversing the computation graph of function $f(x)$ to obtain the graph of $\nabla f(x)$. Visiting any non-leaf vertex implies computing the adjoint derivatives for each of its predecessors. These adjoint quantities actually are computation graphs themselves. With a language that allows explicit reference manipulation, one defines an adjoint derivative for predecessor p of vertex v by adding a bounded number of nodes and edges to the graph at once associated to v. Hence, the adjoints at independent variable leaves are graphs for which size is bounded above by $|V[f(x)]|$. Obviously, the number of such leaf adjoints to compose to obtain the computation graph of $\nabla f(x)$ is bounded as well by $|V[f(x)]|$, so $C[\nabla f(x)] \in O(|V[f(x)]|)$, leading to $C[\nabla f(x)] \in O(C[f(x)])$.

However, Scilab allows no such explicit reference handling. Hence, setting up the graph of the adjoint derivative for vertex p that precedes vertex n imply a *deep copy* of the graph of the adjoint derivative set for vertex n. Knowing that the adjoint graphs of leaf vertexes grow to a size that is $O(|V[f(x)]|)$, their space complexity is $O(|V[f(x)]|^2)$. In addition, the only convention of parameter passing that Scilab supports is *by value*. Thus, the dispatch of the visitor object from the traversal coordinator and the visit methods imply a deep copy of this object, as well as when the methods returns. Since the visitor object has a data member to store the adjoint for each node, the object copying becomes the bottleneck of the traversal, yielding a run-time complexity in $O(|V[f(x)]|^3) \subset O(C[f(x)]^3)$. Indeed, empirical performance observations are disappointing for a naive implementation of the derivation visitor.

Fortunately, this issue has been resolved with the combination of two strategies. First, we shunt the dispatching of the visitor object by making it a global variable.[5] The traversal coordinator binds the visitor to a global symbol; visit methods, instead of explicitly receiving the visitor as one of their parameters, refer to it using this agreed symbol. As functional programming purists, we are aware of the kludgey nature of this solution, but until Scilab gives tools for passing parameters to function by reference, it is the only way to ease the performance stranglehold.

Second, we need to bound the growth of the graphs of the adjoints. To this end, we introduce *graph stubs*. These are one-vertex graphs that refer to the adjoint of another node or to a sub-graph of the graph of $f(x)$. Graph stubs set a constant upper bound to the number of vertexes of the adjoint associated almost[6] every vertex of the graph of

[5]This strategy is functionally and semantically equivalent to using the `return` or `resume` keywords of the Scilab language. In fact, this approach, would have even better since it documents better the object sharing convention between the caller and callees. Our global variable approach puts the shared objects at the root of the name space, so the sharing convention between a strict number of functions is less obvious.

[6]Exceptions are operands of outer products, as well as vertexes bound to a matricial intermediate result.

Operation	Relative cost
Memory store	2
Memory fetch	1
Addition	4
Multiplication	8
Division	24
Primitive function	247

Table 1.2: Elementary operations and their relative cost.

$f(x)$. The work of building the graph of the derivative from the graph of each adjoints is more involved, however : we have to replace stub vertexes with the graph it refers to. Yet, this supplemental work is amortized through memoization : once the graph of an adjoint has been included in that of the derivative, further stubs that refer to it are not substituted into a second copy. The produced graph of $\nabla f(x)$ is then rarely a tree, it more often is a more general acyclic oriented graph. However, such memoization constrains the size of the graph of $\nabla f(x)$ so it becomes almost $O(|V[f(x)]|)$. Therefore, $|V[\nabla f(x)]| \in O(|V[f(x)]|)$ and $C[\nabla f(x)] \in O(C[f(x)])$ when applying these strategies.

We sit this performance assessment on experiments led on a Sun Fire V880 with six UltraSPARC processors and four Gig of main memory running SunOS 5.8. The protocol consisted into comparing the number of elementary operations and the running time involved into evaluating various functions and their gradient at four points into their domain with that of evaluating only the functions. The functions we used were objective functions of the classical collection of unconstrained optimization problems by Moré, Garbow and Hillstrom [18].

First, figure 1.8 illustrates how our implementation of backward-mode derivation satisfies the theory [13], whereas $C[f(x)] + C[\nabla f(x)] \leq 5C[f(x)]$. Indeed, the highest cost ratio[7] we observe is on the order of 3.7. These cost measures are expressed in terms of *elementary operations*, which relate to basic processor instructions that pertain to our calculations. Table 1.2 lists what we consider such elementary operations and their relative cost. The last row of this table mentions "primitive functions", which encompass standard functions such as exponentials, logarithms, roots and so on. We abstract the actual implementation of these functions and reduce them to the evaluation of a six-terms Taylor series.

Second, figure 1.9 maps the run time needed to compute function and its gradient to the operational cost of computing only the function. Despite the presence of outliers, we observe a sufficient linear trend. We deduce that the real running time of evaluating a function and its gradient using our tool grows linearly with respect to the size of the

[7]The *cost ratio* we mention is that of evaluating the function and its gradient to that of evaluating only the function.

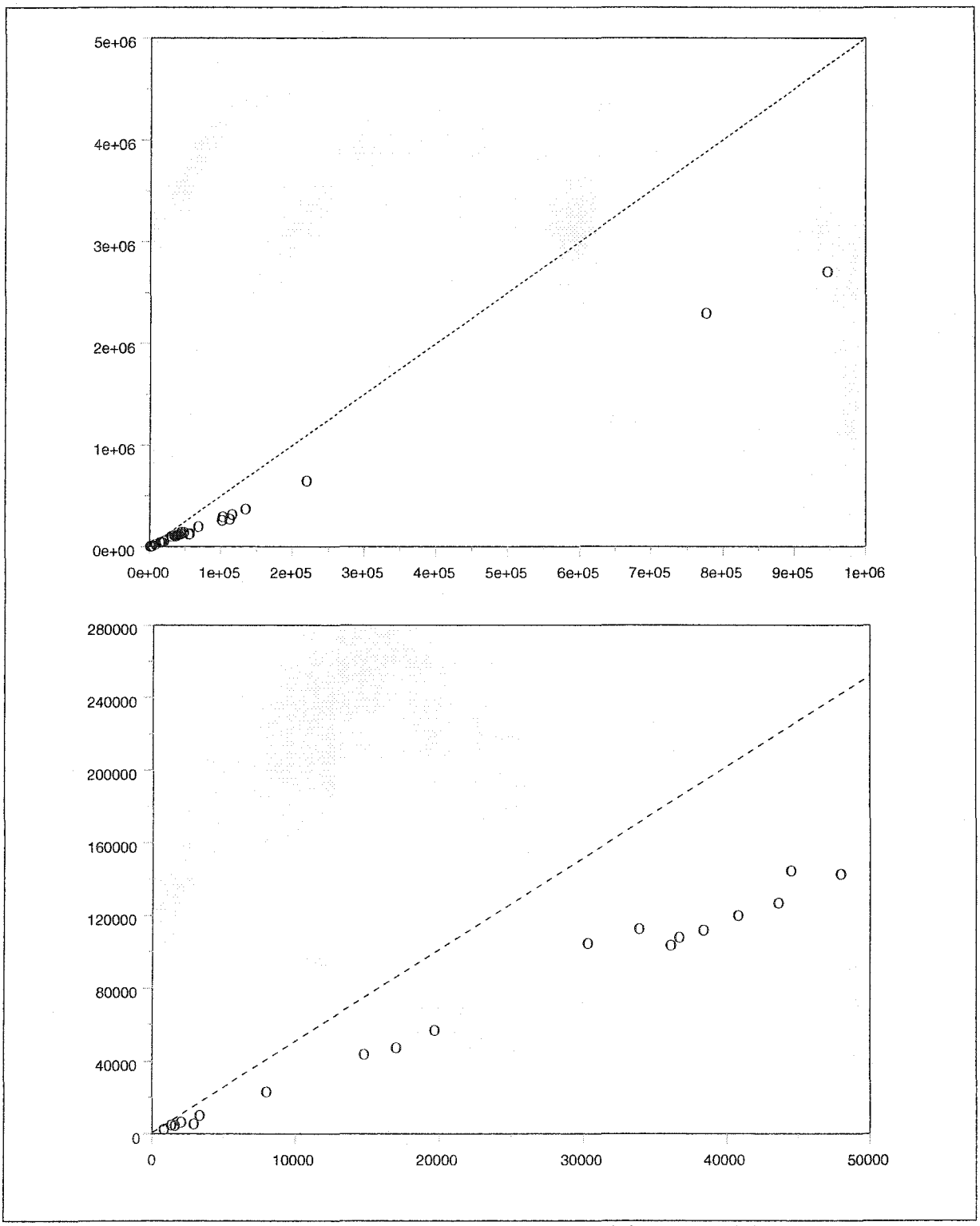


Figure 1.8: Elementary operation cost of evaluating $f(x)$ and $\nabla f(x)$ in function to that of evaluating only $f(x)$. The dashed line delimits the acceptable cost ratio region. The lower plot zooms in on the unclear $[0, 50000]$ interval of the domain.

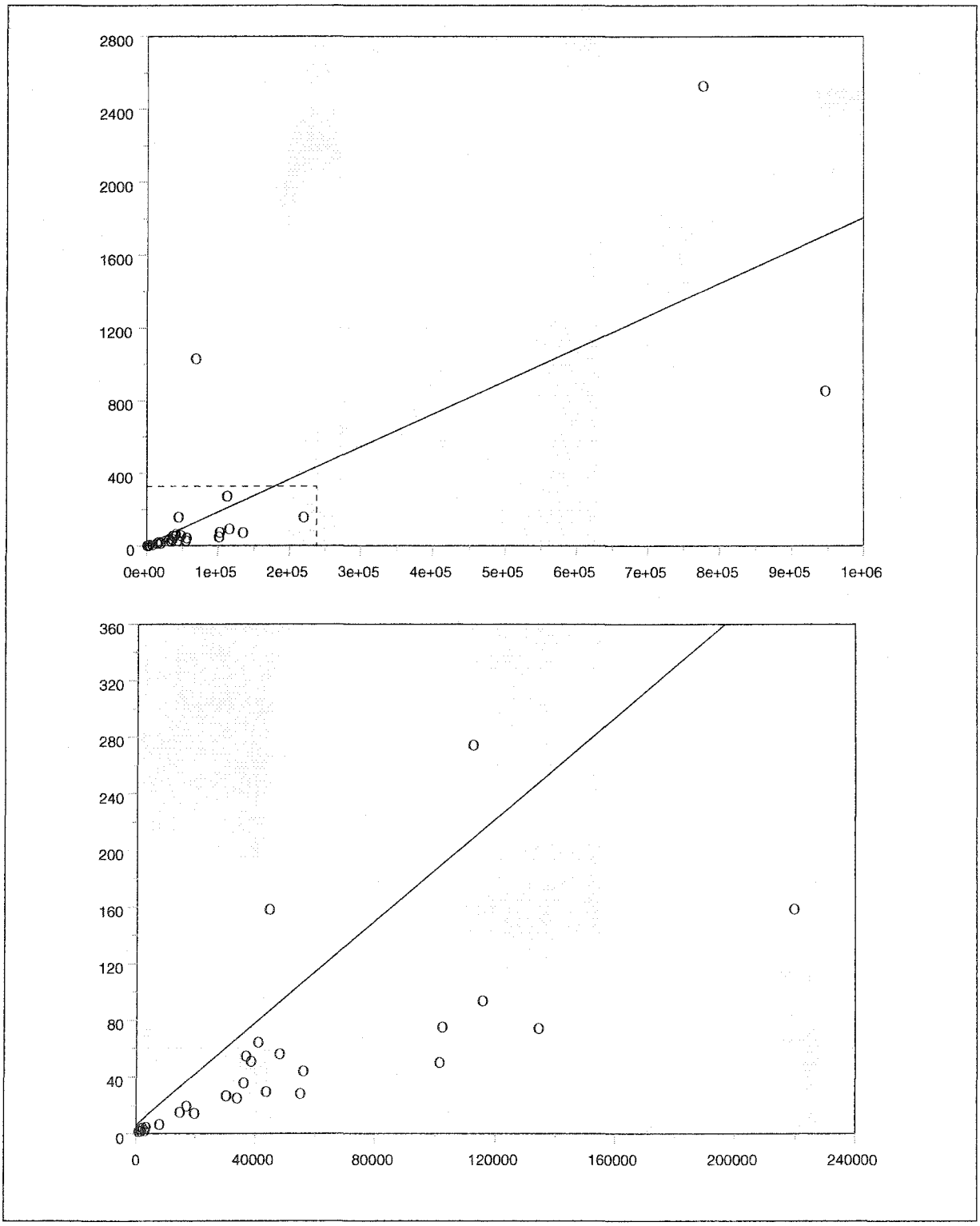


Figure 1.9: Run time of evaluating $f(x)$ and $\nabla f(x)$ in function to that of evaluating only $f(x)$. The lower plot zooms in on the marqueed zone. The correlation factor of the linear regression is 0.7670.

computation graph of the function. This comforts our assertion that the implementation strategies we have described effectively cuts through the cubic algorithmic complexity we had determined for the naive implementation of the visitor.

Conclusion

Using clever design strategies and overcoming target language challenges, we achieved the implementation of an easily maintainable and expandable AD toolkit in an open source high level numerical processing environment. We are satisfied with its flexibility in context of our experimentations in differentiable optimization ([10], [9]).

The main caveats SCIAD faces are performance issues. Further work is planned to resolve these issues, which mostly involve the graph construction processes, as well as putting forth tools to reuse graphs as much as possible. Full source code of SCIAD is available at `http://benoithamelin.com/sciad/sciad.html` and main releases will be submitted to INRIA for posting on the main Scilab web site [21].

References

[1] Michael C. Bartholomew-Biggs, Steve Brown, Bruce Christianson, and Laurence C. W. Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124:171–190, 2000.

[2] Claus Bendtsen and Ole Stauning. Tadiff, a flexible c++ package for automatic differentiation using taylor series expansion. Technical Report IMM-REP-1997-07, Department of Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby – Denmark, April 1997.

[3] Martin Berz. Algorithms for higher order automatic differentiation in many variables with applications to beam physics. In George F. Corliss and Andreas Griewank, editors, *Automatic Differentiation of Algorithms – Theory, Implementation and Applications*. SIAM, 1991.

[4] Martin Berz and Georg Hoffstätter. Computation and application of taylor polynomials with interval remainder bounds. *Reliable Computing*, 4(89–97), 1998.

[5] Christian Bischof and Andreas Griewank. Tools for the automatic differentiation of computer programs. In G. Alefeld, O. Mahrenholtz, and R. Mennicken, editors, *ICIAM/GAMM 95 : Issue 1 : Numerical Analysis, Scientific Computing, Computer Science*, pages 267 – 272, 1996.

[6] Christian H. Bischof, Alan Carle, Paul D. Hovland, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0 user's guide (Revision D). Technical report, Mathematics and Computer Science Division Technical Memorandum no. 192 and Center for Research on Parallel Computation Technical Report CRPC-95516-S, 1998.

[7] Thomas F. Coleman and Arun Verma. Structure and efficient jacobian calculation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation : Techniques, Applications and Tools*, pages 149 – 159. SIAM, 1996.

[8] Thomas F. Coleman and Arun Verma. ADMAT: An automatic differentiation toolbox for MATLAB. Technical report, Computer Science Department, Cornell University, 1998.

[9] Jean-Pierre Dussault. High order newton-penalty algorithms. *Journal of Computational and Applied Mathematics*, submitted for publication.

[10] Jean-Pierre Dussault and Benoit Hamelin. Robust descent in differentiable optimization using automatic finite differences. Technical Report 301, Université de Sherbrooke, 2003.

[11] L.M. Beda et al. Programs for automatic differentiation for the machine besm. Technical report, Moscow, 1959.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*, pages 331 – 344. Professional Computing Series. Addison-Wesley, 1994.

[13] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[14] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[15] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Society for Industrial & Applied Mathematics, January 1992.

[16] Andreas Griewank, David Juedes, Hristo Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[17] Dick Grune, Henri E. Bal, Ceriel Jacobs, and Koen G. Langendoen. *Modern Compiler Design.* John Wiley & Sons, 2000.

[18] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Algorithm 566: FORTRAN subroutines for testing unconstrained optimization software [C5 [E4]]. *ACM Transactions on Mathematical Software*, 7(1):136–140, March 1981. [ftp://ftp.mathworks.com/pub/contrib/v4/optim/uncprobs.tar].

[19] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1981.

[20] Louis B. Rall and George F. Corliss. An introduction to automatic differentiation. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 1–17. SIAM, Philadelphia, PA, 1996.

[21] Scilab group. *Scilab 2.7.* Institut National de Recherche en Informatique et Automatique - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 - LE CHESNAY Cedex - FRANCE, email : Scilab@inria.fr, 2003. `http://scilabsoft.inria.fr/`.

[22] R.E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7:463 – 464, 1964.

A Backward-mode derivation formulas

The trickiest part of the implementation of backward-mode automatic differentiation of high-level object computation graphs is to figure out how to compute the adjoint derivatives. We recall that the goal is, for some function $f : \mathbb{R}^n \to \mathbb{R}^m$ and co-tangent matrix $N : p \times m$, to compute $N\nabla f(x)$, a $p \times n$ matrix – when $m = 1$, $N\nabla f(x)$ is made to be a $1 \times n$ row vector. We also recall that every intermediate result allowed to be associated to a vertex of the association graph is a matrix.[8] Moreover, all independent variable vectors are column vectors by construction: row vectors of independent variables may be obtained by a transposition. Hence, the structure we expect for the adjoint derivative of some vertex depends on the corresponding intermediate result:

Scalar: the adjoint derivative is a $p \times 1$ column vector.

Column vector $s \times 1$**:** the adjoint derivative is a $p \times s$ matrix.

Row vector $1 \times s$**:** the adjoint derivative is a $s \times p$ matrix.

[8]Scalars and vectors are obviously allowed, being matrix specializations.

Matrix $s \times t$: the adjoint derivative is a $s \times t \times p$ tensor, that is, a list of p matrix frames of dimension $s \times t$.

The following table summarizes formulas used to compute adjoint derivatives in SCIAD, which formulas follow the rules mentioned above. The left member of equations used to describe the operators are the result of the computation of the said operator, so its adjoint is expected to be known when we visit the vertex bound to this operator. What we compute there are the respective adjoints of its operands. Note that some more primitives are implemented in SCIAD, notably trigonometric functions; aiming for conciseness, we give formulas for the most common primitives, which are the arithmetic operators, exponentiation (divided in exponential, which involve a constant base, and monomial, which rather involve a constant exponent) and natural logarithm.

It happens that adjoints are defined using sub-structures of matrices and vectors: we use the Matlab notation to represent these. Take some vector x: $x(i)$ denotes the ith component of vector x. In addition, take matrix A: we denote its ith column by $A(:, i)$ and its jth row by $A(j, :)$. Finally, the kth frame of a tensor T is denoted $T(:, :, k)$. We also let us name e^n the $\mathbb{R}^n$ column vector such that $\forall 1 \leq i \leq n : e^n_i = 1$.

Lastly, we note that Scilab, as Matlab, offers operators to apply element-wise (E-W) scalar operations to vectors and matrices. For instance, element-wise product $A. * B$ yields a matrix for which the element at row i and column j is the scalar product of A_{ij} and B_{ij}. Adjoint calculation formulas are given for such operators as well.

Primitive operator		Adjoint formula
Addition $z = x + y$		$\tilde{x} = \tilde{z}$
		$\tilde{y} = \tilde{z}$
Additive inversion $z = -x$		$\tilde{x} = -\tilde{z}$
Transposition $z = x^t$		
	Vector	$\tilde{x} = \tilde{z}^t$
	Matrix	$\tilde{x}(:, :, i) = \tilde{z}(:, :, i)^t$
Product $z = xy$		
	x and y scalars	$\tilde{x} = y\tilde{z}$
		$\tilde{y} = x\tilde{z}$
	$x : n \times 1$, y scalar	$\tilde{x} = y\tilde{z}$
		$\tilde{y} = \tilde{z}x$
	x scalar, $y : 1 \times n$	$\tilde{x} = \tilde{z}^t y^t$
		$\tilde{y} = x\tilde{z}$
	$x : 1 \times n$, $y : n \times 1$	$\tilde{x} = y\tilde{z}^t$
		$\tilde{y} = \tilde{z}x$

Primitive operator	Adjoint formula
$x : m \times 1,\ y : 1 \times n$	$\tilde{x}(i,:) = y\tilde{z}(:,:,i)^t$
	$\tilde{y}(:,i) = \tilde{z}(:,:,i)^t x$
$x : 1 \times m,\ y : m \times n$	$\tilde{x} = y\tilde{z}$
	$\tilde{y}(:,:,i) = x^t\tilde{z}(:,i)^t$
$x : m \times n,\ y : n \times 1$	$\tilde{x}(:,:,i) = \tilde{z}(i,:)^t y^t$
	$\tilde{y} = \tilde{z}x$
$x : m \times n,\ y : n \times q$	$\tilde{x}(:,:,i) = \tilde{z}(:,:,i)y^t$
	$\tilde{y}(:,:,i) = x^t\tilde{z}(:,:,i)$
Element-wise product $x\ .*\ y$	
Scalars	$\tilde{x} = y\tilde{z}$
	$\tilde{y} = x\tilde{z}$
Column vectors	$\tilde{x} = (e^p y^t)\ .*\ \tilde{z}$
	$\tilde{y} = (e^p x^t)\ .*\ \tilde{z}$
Row vectors	$\tilde{x} = (e^p y)^t\ .*\ \tilde{z}$
	$\tilde{y} = (e^p x)^t\ .*\ \tilde{z}$
Matrices	$\tilde{x}(:,:,i) = y\ .*\ \tilde{z}(:,:,i)$
	$\tilde{y}(:,:,i) = x\ .*\ \tilde{z}(:,:,i)$
Inversion $z = x^{-1}$	
Scalar	$\tilde{x} = -\tilde{z}x^{-2}$
Matrix	$\tilde{x}(:,:,i) = -(x^{-1})^t\tilde{z}(:,:,i)(x^{-1})^t$
E-W inversion $z = e^m(e^n)^t\ ./x$	
Scalar	$\tilde{x} = -\tilde{z}x^{-2}$
Column vector	Let $X = e^p x^t$; $\tilde{x} = -\tilde{z}\ ./(X\ .*\ X)$
Row vector	Let $X = x^t(e^p)^t$; $\tilde{x} = -\tilde{z}\ ./(X\ .*\ X)$
Matrix	$\tilde{x}(:,:,i) = -\tilde{z}(:,:,i)\ ./(x\ .*\ x)$
Exponential $z = K^x$ (K is a constant)	
Scalar	$\tilde{x} = \tilde{z}K^x \log K$
E-W exponential $z = \exp_K(x)$	
Scalar	$\tilde{x} = \tilde{z}\exp_K(x)\log K$
Column vector	$\tilde{x} = [e^p(\exp_K(x)\ .*\ \log K)^t]\ .*\ \tilde{z}$
Row vector	$\tilde{x} = [e^p(\exp_K(x)\ .*\ \log K)]^t\ .*\ \tilde{z}$
Matrix	$\tilde{x}(:,:,i) = \exp_K(x)\ .*\ \tilde{z}(:,:,i)\ .*\ \log K$
Monomial $z = x^K$ (K is a constant)	
Scalar	$\tilde{x} = K\tilde{z}x^{K-1}$
E-W monomial $z = \mathrm{pow}_K x$	
Scalar	$\tilde{x} = kx^{k-1}\tilde{z}$
Column vector	$\tilde{x} = [e^p(K\ .*\ \mathrm{pow}_{K-1}x)^t]\ .*\ \tilde{z}$
Row vector	$\tilde{x} = [e^p(K\ .*\ \mathrm{pow}_{K-1}x)]^t\ .*\ \tilde{z}$
Matrix	$\tilde{x}(:,:,i) = K\ .*\ \mathrm{pow}_{K-1}x\ .*\ \tilde{z}(:,:,i)$

Primitive operator	Adjoint formula
Natural logarithm $z = \log x$ (E-W)	
Scalar	$\tilde{x} = \tilde{z}/x$
Column vector	$\tilde{x} = \tilde{z} \,./(e^p x^t)$
Row vector	$\tilde{x} = \tilde{z} \,./(e^p x)^t$
Matrix	$\tilde{x}(:,:,i) = \tilde{z}(:,:,i) \,./x$

1.2 Compléments

1.2.1 Reconstruction ou réutilisation des graphes d'évaluation

Les graphes d'évaluation de SCIAD sont évidemment développés comme des instances d'un type abstrait de données, de même que les composantes de ces graphes – en particulier, chaque sommet d'un graphe est représenté par une `mlist`. Ceci dit, Scilab ne permet pas de manipuler facilement des références aux objets instanciés. Ces références sont en effet gérées par un mécanisme de copie sur écriture automatique. Par exemple, soit une variable `v` associée à une certaine liste. On peut faire en sorte qu'une nouvelle variable `w` réfère à la même liste par l'instruction `w = v`. Cependant, aussitôt qu'on veut changer l'un des éléments de la liste `w`, une copie de la liste originale (référée par `v`) est générée et `w` y est redirigée; c'est l'élément de cette nouvelle liste qui est modifié et la liste `v` demeure inchangée.

Ce mécanisme automatique gêne le fonctionnement de SCIAD, particulièrement lors de la construction du graphe d'évaluation d'une fonction. Ce processus étant réalisé par le biais de la surcharge des opérateurs, les listes définissant les graphes et leurs sommets sont constamment créées, copiées, détruites, tant et si bien que beaucoup de temps est consacré à la gestion de la mémoire. Sachant qu'un graphe dans SCIAD est construit en modélisant l'exécution de telle fonction f en un point donné x_0, il apparaît intéressant de réutiliser de quelque manière le graphe, dans le but de rentabiliser le temps investi dans sa construction. Cela peut être aisément fait à l'aide d'un visiteur postordre qui modifie les membres internes appropriés des sommets du graphe pour qu'il représente plutôt l'évaluation de f en un second point x_1.

Cependant, un tel visiteur n'est pas implanté dans SCIAD et les graphes d'évaluation ne sont pas réutilisés. La raison en est que le processus de construction du graphe, basé sur la surcharge des opérateurs, ne voit pas le programme complet d'évaluation de f. Prenons

l'exemple d'une fonction définie par branches : selon la valeur de l'argument de la fonction x, telle branche sera exécutée lors de l'appel $f(x)$. Cette sélection sera effectuée à l'aide d'une structure de contrôle invisible aux fonctions surchargeant les opérateurs. Ainsi, le graphe d'évaluation obtenu de l'exécution de $f(x)$ avec x une instance de notre type abstrait de données ne modélise qu'une des multiples branches de f. Penchons nous sur un second exemple, où f constitue une simulation stochastique paramétrique. Les nombres aléatoires générées sans égard au paramètre x sont propres à une seule exécution de f, de sorte que le graphe d'évaluation ne peut être réutilisé. L'alternative consiste donc à reconstruire les graphes pour chaque point du domaine où la dérivée doit être évaluée.

La littérature semble accorder peu d'attention à ce sujet. D'une part, les logiciels de différentiation automatique pour lesquels les graphes d'évaluation sont construits à l'aide d'un précompilateur (comme ADIFOR [BCH+98]) ne souffrent pas de ce problème. Ces logiciels traitent effectivement l'information liée aux structures de contrôle (conditionnelles et boucles). Les programmes qu'ils génèrent incluent donc le code de dérivation pour toutes les branches d'une fonction, pour reprendre cet exemple. D'autre part, les logiciels qui, à l'instar de SCIAD et ADOL-C [GJM+99], acquièrent le graphe d'évaluation par surcharge des opérateurs, ignorent le problème en déférant au modeleur la gestion des discontinuités de la fonction. Avec ADOL-C, les graphes peuvent être réutilisés; cependant, la validité de cette réutilisation n'est pas garantie par le logiciel. Au minimum, il faudrait donc que les outils nécessaires à une telle réutilisation soient développés dans SCIAD pour en permettre l'adoption au-delà de nos travaux.

1.2.2 Gestion des résultats intermédiaires

Dans un ordre d'idées analogue, afin d'accélérer le calcul de dérivées en mode préordre, SCIAD stocke dans chaque sommet du graphe d'évaluation de $f(x)$ le résultat numérique

intermédiaire de l'évaluation de f à ce sommet. On conçoit que, pour une fonction $f : \mathbb{R}^n \to \mathbb{R}^m$ avec $m + n$ grand, ce stockage des résultats intermédiaires implique un usage excessif des mémoires de la machine. En effet, le graphe d'évaluation d'une fonction a une taille proportionnelle au nombre d'opérations primitives qui la compose, d'où une complexité spatiale minimale $\Omega(\max\{m, n\})$, avant même d'avoir considéré l'usage mémoire des résultats intermédiaires rattachés à chaque noeud, lesquels peuvent être des matrices. À la rigueur, si ces résultats intermédiaires doivent être temporairement transférés au disque dur du fait des mécanismes de mémoire virtuelle du système d'exploitation sous-jacent, les gains en temps d'exécution espérés peuvent être réduits à néant. Notons en passant que les expériences qui nous intéressent concernent l'usage de dérivées d'ordre supérieur pour des problèmes de laboratoire. De ce fait, ce problème est pour nous négligeable dans l'immédiat.

Ceci dit, des solutions existent pour faire de SCIAD un outil intéressant dans la résolution de problèmes de grande taille. Divers travaux de Griewank (culminant avec [Gri92]) suggèrent de linéariser le graphe d'évaluation en une liste d'opérations élémentaires qui satisfait l'ordre partiel impliqué par les arêtes du graphe. Lorsque cette liste croît au-delà d'un certain seuil, on en tronque un certain préfixe. La dérivation en préordre implique une simple énumération de la fin au début d'une telle liste. Lorsqu'on arrive au bout tronqué, on élimine le suffixe déjà parcouru et on reconstruit le préfixe précédemment rejeté. Selon [Gri92], une bonne gestion des troncations et reconstructions du graphe permet d'atteindre une complexité $O(\max\{m, n\} \log \max\{m, n\})$ pour le temps d'exécution et $O(\log \max\{m, n\})$ pour l'usage mémoire.

1.2.3 Complexité du graphe d'évaluation de la dérivée

Néanmoins, la construction du graphe de $\nabla f(x)$ implique des problèmes d'espace préoccupants en regard de nos objectifs. En effet, reprenons l'exemple de la fonction

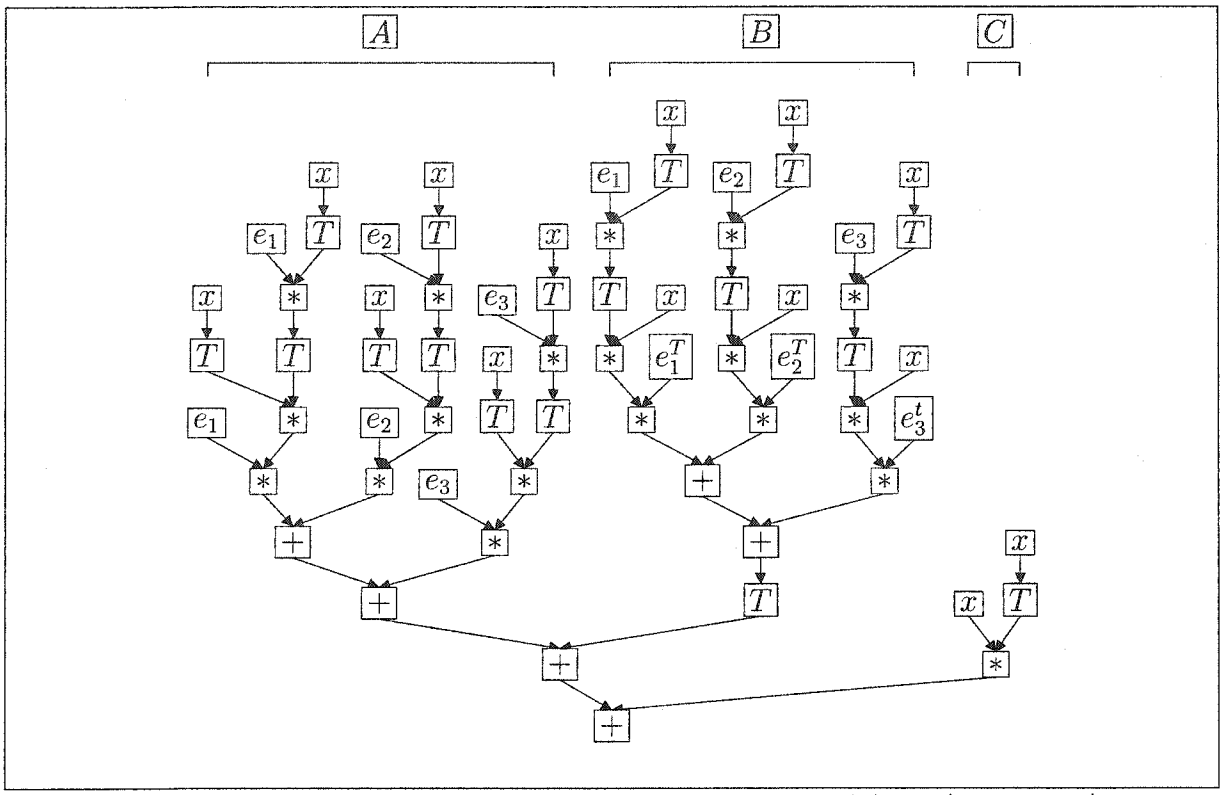


Figure 1.10: Graphe d'évaluation de la fonction $\nabla g(x) = x^t x I_3 + 2xx^t$.

$g(x) = (xx^t)x$ dont le graphe d'évaluation est représentée par la figure 1.6. La figure 1.7 illustre que $\nabla g(x) = x^t x I_n + 2xx^t$, comme prévu. Cependant, restreignant le domaine de la fonction à $\mathbb{R}^3$, l'usage des formules de dérivation exprimées en annexe A exprime $\nabla g(x)$ par le graphe d'évaluation de la figure 1.10.

On remarque de cette figure que la complexité spatiale de l'arbre croît avec la dimension du domaine de f : les groupes de sous-arbres A et B consistent respectivement en l'addition de trois sous-arbres ne différant que par deux sommets constants. Pour $x \in \mathbb{R}^n$, les groupes A et B du graphe de $\nabla g(x)$ seraient ainsi constitués de n tels sous-graphes analogues. Soyons à choisir une direction $d \in \mathbb{R}^n$ dans laquelle on veuille obtenir $\nabla^2 g(x)d$: il est possible d'obtenir cette information par la dérivation en préordre de $\nabla g(x)d$. Cependant, comme le graphe de $\nabla g(x)d$ comporte lui aussi des résultats intermédiaires matriciels, on assiste à une explosion exponentielle du nombre de noeuds dans

le graphe de $\nabla^2 g(x)d$, à mesure des dérivations successives. Ce phénomène compromet donc la possibilité d'obtenir les dérivées directionnelles d'ordre supérieur à 2.

Le problème découle effectivement de la présence de résultats intermédiaires matriciels dans un graphe d'évaluation. Lors du calcul en préordre de la dérivée de la fonction, l'adjointe rattachée à de tels noeuds consiste en un tenseur, dont notre représentation provoque l'explosion de la complexité du graphe de la dérivée. La solution à ce problème consiste en l'utilisation d'un visiteur spécial, un *éliminateur de matrices.* Considérant l'ensemble de primitives qui peuvent constituer nos graphes d'évaluation, de telles matrices viennent de l'une de deux sources : une matrice de constantes ou un produit externe de vecteurs variables. L'éliminateur de matrices devrait visiter le graphe d'évaluation en préordre et déterminer, pour chaque produit dont le résultat est une matrice, la cause de l'apparition de la matrice, ce à l'aide d'un ensemble d'heuristiques simples. Cette cause identifiée, le visiteur devrait modifier le graphe de manière à réorganiser les produits pour éviter l'apparition de matrices intermédiaires de variables. Ceci dit, cela n'est possible que pour le graphe d'évaluation d'une fonction vectorielle ($\mathbb{R}^n \rightarrow \mathbb{R}^m$). Ainsi, le visiteur devrait par ailleurs être paramétré à l'aide d'un vecteur de direction pour le produit à droite de la fonction matricielle $M(x)d$, ou encore d'un vecteur de pondération des lignes pour produit à gauche $vM(x)$.

Reprenons l'exemple de la fonction $\nabla g(x)$ (figure 1.10) pour démontrer le travail d'un tel éliminateur de matrices, paramétré par un vecteur de direction v multiplié à droite. À partir de la racine de l'arbre, le visiteur utilise la distributivité du produit sur l'addition pour propager le vecteur v vers les sept noeuds de produit les plus bas. Le visiteur traite d'abord le sous-arbre C. La racine de ce sous-arbre est associée au produit externe de x et de x^T; il suffit donc de remplacer son opérande de droite par le nouveau sous-graphe $x^T v$. La racine devient donc le produit d'un vecteur (x) et d'un scalaire et la matrice intermédiaire xx^T est effectivement éliminée.

Poursuivons l'exemple sur le sous-arbre le plus à droite du groupe B, représenté par la figure 11(a); le processus devrait être analogue pour les deux autres sous-arbres de B.

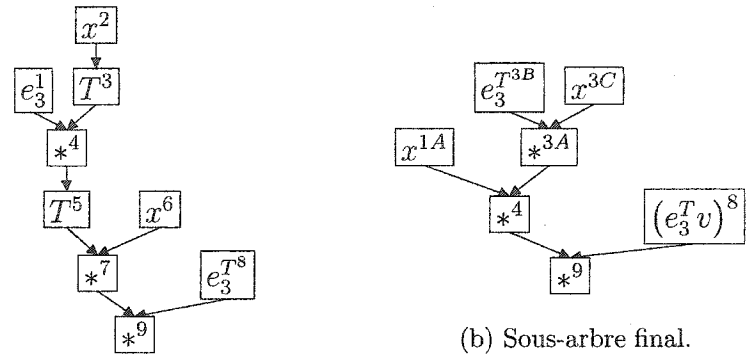
Noeud 9 Ce noeud représente déjà un produit externe entre le vecteur de variables associé au noeud 7 et le vecteur constant e_3^T du noeud 8 : en multipliant ce dernier par v, on obtient une constante scalaire au noeud 8, de sorte que le produit au noeud 9 devient celui d'un vecteur et d'un scalaire et la matrice intermédiaire a bien été éliminée.

Noeud 8 Rien à faire aux feuilles.

Noeud 7 On a ici le produit de la matrice définie au noeud 5 et du vecteur de variables x associé au noeud 6. Afin d'éliminer la matrice intermédiaire, on définit le vecteur x comme un vecteur de direction qu'on fera progresser plus haut dans le graphe. On place le sous-arbre enraciné au noeud 6 dans une zone mémoire temporaire du visiteur et on remplace le noeud 7 par le noeud 5, escamotant effectivement le graphe d'évaluation. Notons que cette réduction du graphe fait en sorte qu'on ne pourra pas visiter le noeud 6.

Noeud 5 La matrice associée à ce noeud constitue la transposition de la matrice définie au prédécesseur, le noeud 4. On note ce fait dans un champ approprié du visiteur, on détruit le noeud 5 et on lie le noeud 4 au noeud 9 (le successeur du noeud 5).

Noeud 4 On a ici le produit externe du vecteur constant e_3 (noeud 1) et du vecteur de variables indépendantes x^t (noeud 3). On commence par rétablir la transposition en la passant aux prédécesseurs, qu'on doit d'ailleurs réordonner pour satisfaire la propriété $(AB)^t = B^t A^t$. On se retrouve donc avec deux nouveaux prédécesseurs : le noeud 1A associé au vecteur de variables x et le noeud 3A associé au vecteur constant e_3^T. Ceci établi, on applique la même stratégie qu'au noeud 9 pour transformer le produit externe en produit d'un vecteur et d'un scalaire : on associe au



(a) Sous-arbre original.

(b) Sous-arbre final.

Figure 1.11: Sous-arbre de droite du groupe B de la figure 1.10, avant et après le passage d'un éliminateur de matrices. Les noeuds sont numérotés par leur exposant.

noeud 3A le graphe obtenu par le produit $e_3^T x$ (x étant le sous-arbre préservé lors de la visite du noeud 7). Le noeud 3A se retrouve associé à un produit et ses prédécesseurs sont les noeuds 3B (e_3^T) et 3C (x).

Noeud 3A Ce noeud est associé au produit de deux vecteurs, on n'y change rien.

Noeud 3C Rien à faire aux feuilles.

Noeud 3B Idem.

Noeud 1A Idem.

On se retrouve avec le graphe de la figure 11(b). Le travail du visiteur se poursuit sur les deux autres sous-arbres de B, puis s'attaque aux sous-arbres du groupe A. Le sous-arbre de droite de ce groupe est illustré à la figure 12(a).

Noeud 10 Ce noeud constitue le produit externe du vecteur constant e_3 (noeud 1) et d'un vecteur variable (noeud 9). On tâchera de réduire la représentation du vecteur du noeud 9 à un scalaire en y propageant le vecteur de direction v.

Noeud 9 Il s'agit du produit entre le vecteur x^T (noeud 3) et une matrice variable (noeud 8). On propage le vecteur de direction au noeud 8 pour le transformer en vecteur colonne, ce qui ferait associerait le noeud 9 à un produit interne (dont le résultat est un scalaire).

Noeud 8 La matrice associée à ce noeud constitue la transposition de la matrice du noeud 7. On note ce fait dans un champ approprié du visiteur, on détruit le noeud 8 et on définit une arête allant du noeud 7 au noeud 9.

Noeud 7 Ceci constitue le produit externe du vecteur e_3 (noeud 4) et du vecteur variable x^T (noeud 6). On commence par transformer les prédécesseurs de manière à appliquer la règle de distribution de la transposition $(AB)^T = B^T A^T$ (notée au noeud 8). On se retrouve avec les prédécesseurs $4A$, associé au vecteur variable x, et $6A$, associé au vecteur constant e_3^t. Pour transformer le produit externe en produit d'un vecteur colonne et d'un scalaire, on remplace le vecteur constant e_3^t associé au prédécesseur 6A par la constante scalaire $e_3^T v$.

Noeud 6A Rien à faire aux feuilles de l'arbre.

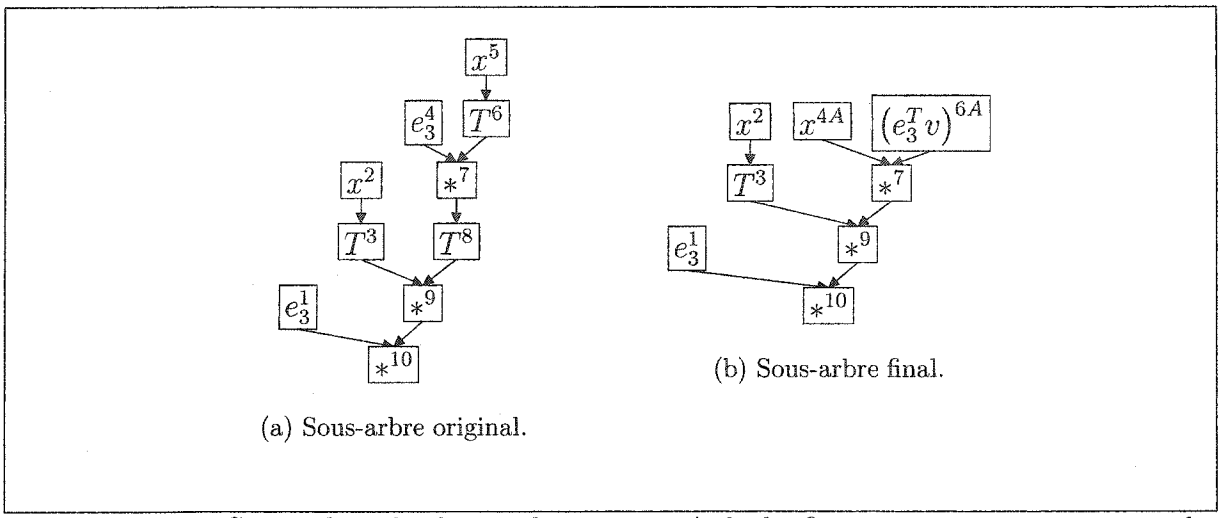
Noeud 4A Idem.

Noeud 3 Ceci est la transposition d'un vecteur. Comme il n'est pas question de matrice intermédiaire en ce noeud, on n'a rien à faire.

Noeud 2 Rien à faire aux feuilles de l'arbre.

Noeud 1 Idem.

On se retrouve avec le sous-arbre de la figure 12(b). Le visiteur complète son travail en réalisant la même séquence d'opérations sur les deux autres sous-arbres de A. L'implantation de ce visiteur constitue une tâche importante de nos travaux futurs, en



(a) Sous-arbre original.

(b) Sous-arbre final.

Figure 1.12: Sous-arbre de droite du groupe A de la figure 1.10, avant et après le passage d'un éliminateur de matrices. Les noeuds sont numérotés par leur exposant.

vue de faciliter le développement d'algorithmes nécessitant l'évaluation de dérivées directionnelles d'ordre adaptatif.

Ayant établi le design et l'implantation d'un outil de différentiation automatique, penchons nous sur les solutions qu'apporte cet outil à divers problèmes numériques dont souffrent les méthodes automatiques d'optimisation sans contrainte.

CHAPITRE 2

AMÉLIORATION DE LA ROBUSTESSE DES ALGORITHMES D'OPTIMISATION SANS CONTRAINTE

2.1 Préambule

Comme mentionné en introduction de ce mémoire, l'optimisation sans contrainte de fonctions différentiables est une discipline dont les méthodes sont fondées sur une théorie claire et bien acceptée dans la communauté. Cependant, au-delà des modèles et des approches fondés sur des théorèmes de convergence, les praticiens font face à des problèmes pour lesquels ces modèles et approches échouent.

L'une des difficultés que doivent gérer les méthodes automatisées d'optimisation consiste en la représentation des nombres sur la machine cible de leur implantation. En effet, nous travaillons sur des machines à états discrets finis sur lesquelles on ne peut manipuler que des approximations du concept d'infini. Par exemple, les nombres réels sont représentés à l'aide de nombres à virgule flottante, un modèle qui décrit les nombres à l'aide d'une *mantisse* et d'un *exposant*, chacun contenu dans une chaîne de bits de longueur fixe. De

ce fait, ces nombres à virgule flottante sont des approximations rationnelles à précision variable des nombres réels. Cette étape d'approximation a des conséquences importantes sur les processus numériques, dont certaines variantes d'implantation peuvent générer des résultats qui souffrent d'imprécisions significatives.

L'article qui suit décrit une difficulté de ce genre, difficulté propre à tous les algorithmes d'optimisation de fonctions différentiables contrôlés par une mesure de la *descente* de l'objectif entre deux itérations successives. Ce problème est bien connu et est généralement contourné à l'aide d'une heuristique triviale : par exemple, pour la méthode de Newton amortie, le calcul d'amortissement est ignoré lorsqu'on détecte être entré dans une certaine boule entourant un minimum local vers lequel l'algorithme apparaît converger. La solution exacte de cette difficulté implique la connaissance et la manipulation d'une représentation du processus de calcul de la fonction objectif; les techniques de différentiation automatique offrent cette connaissance et les outils de cette manipulation. L'article introduit en ce sens le concept de *différences finies automatiques*, une extension des calculateurs de dérivées automatiques qui se chargent plutôt de calculer précisément la quantité $\Delta_d f(x) = f(x+d) - f(x)$. On rapporte finalement les résultats d'expériences numériques sur des codes d'optimisation publics améliorés par un calculateur de différences finies automatiques.

Ma part dans l'élaboration de cet ouvrage a été plus modeste que pour le premier article présenté. Le coauteur, Pr. Jean-Pierre Dussault, a eu l'idée d'appliquer les outils de différentiation automatique à la résolution des erreurs d'annulation observées dans l'évaluation de la descente. Il a formulé la règle de calcul en chaîne des différences finies automatiques et a rédigé l'ouverture de cet article. De mon côté, j'ai implanté le calculateur de différences finies automatiques comme un *visiteur* du graphe d'évaluation d'une fonction (voir chapitre 1) et j'ai déterminé la complexité algorithmique d'une telle visite en regard de la dimension du domaine de la fonction objectif. Pr. Dussault ayant déniché les programmes d'optimisation de Kelley [Kel99] et le code de la banque de problèmes

expérimentaux de Moré, Garbow et Hillstrom [MGH81], j'ai articulé et exécuté les expériences, à la suite desquelles j'ai compilé et analysé les résultats.

Soumis pour publication à *Optimization Software and Methods*; révision en cours.

Robust descent in differentiable optimization using automatic finite differences*

Rapport de recherche 301
Département de Mathématiques et d'Informatique
Faculté des Sciences
Université de Sherbrooke

Jean-Pierre Dussault†
Benoit Hamelin‡

Abstract

Descent algorithms use comparisons of a merit function to ensure the global convergence of generated iterates to at least a stationary point, even to solutions satisfying the second order necessary optimality conditions. Unfortunately, the comparison of the merit function values may fail to be precise enough for estimates reaching the square root of the machine precision. We present a clever way to compare merit function values ensuring that full machine precision may be reached.

Keywords. Unconstrained optimization, descent methods, automatic differentiation.

*This research was partially supported by NSERC grant OGP0005491

†Professeur titulaire, département de Mathématiques et Informatique, Université de Sherbrooke, Sherbrooke (Québec), Canada J1K 2R1. e-mail: Jean-Pierre.Dussault@USherbrooke.ca

‡Université de Sherbrooke, e-mail: benoit@benoithamelin.com

Introduction

We consider descent algorithms for solving

$$\min_{x \in \Re} f(x) \tag{2.1}$$

where it is understood that we are seeking a *local* minimum of f, and that f is $\mathcal{C}^2(\Re^n)$. Since f is differentiable, it is well known that necessary optimality conditions for equation (2.1) state that the local minimum x^* is a stationary point, that is $\nabla f(x^*) = 0$; moreover, $\nabla^2 f(x^*)$ is positive semi-definite. If, for some stationary point $\bar{x}$, $\nabla^2 f(\bar{x})$ is positive definite, then $\bar{x}$ is a local minimum for f (sufficient conditions). For basic theory of differentiable optimization, see for example [15].

The fact that we seek a minimizer of f makes it possible to devise algorithms which will converge to a stationary point, even to a point satisfying the necessary conditions starting from *any* x_0. This property is named *global convergence*, and consists in following a decreasing path, that is, generating a sequence x_k such that $f(x_{k+1})$ is sufficiently smaller than $f(x_k)$, so that if the sequence x_k accumulates, it does so at a stationary point.

When using a descent algorithm, whether of line search type or trust region family, sufficient reduction of the objective function is assessed by the obvious comparison $f(x_{k+1}) - f(x_k) \leq \epsilon_k < 0$. For instance, in line search algorithms, the usual Armijo condition is written $f(x_{k+1}) - f(x_k) \leq \tau_0 \nabla f(x_k)(x_{k+1} - x_k)$. Trust regions globalization use ratios of the form $\frac{f(x_{k+1})-f(x_k)}{m(x_{k+1})-m(x_k)} \leq \tau$ to adjust the trust region radius, and such tests may be rewritten $f(x_{k+1}) - f(x_k) \leq (m(x_{k+1}) - m(x_k))\tau$.

Close to a stationary point, we should have that $f(x_k) - f(x_*) \sim \mathcal{O}(\|x_k - x^*\|^2)$ since stationarity of x^* means vanishing first order derivatives $\nabla f(x^*) = 0$. This means that whenever $\|x_k - x^*\|$ becomes close to the square root of the machine precision $\sqrt{\epsilon_M}$, comparing objective function values involves comparing quantities differing by a quantity of the order of the machine precision ϵ_M. Thus, *numerically*, $f(x_{k+1}) - f(x_k) = 0$ so that the condition $f(x_{k+1}) - f(x_k) \leq \epsilon_k$ cannot be true!

In order to reliably avoid such numerical problems, we propose a novel solution based on the computation of the finite difference $\Delta_d f(x_k) \stackrel{def}{=} f(x_k + d) - f(x_k)$ using techniques inspired by forward or backward automatic differentiation algorithms. The resulting computation naturally gets rid of the troublesome cancellation errors.

The remaining of the paper is organized as follows. We detail the descent problem some more in section 1. In section 2, we then present a sane scheme of computation of merit function descent, dubbed *automatic finite differences*. Section 3 exposes details of the

implementation of this scheme and of enhanced unconstrained optimization procedures. We finally report on numerical experimentations in section 4.

1 Details on the descent evaluation problem

1.1 An example

Consider for example the well known Rosenbrock function:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

Its minimum point is $x^* = (1, 1)^t$ for which $f(x^*) = 0$. Since $f(x^*)$ vanishes, the descent test involves comparing quantities close to 0, so that in floating point arithmetics, there should be no problem, unless the zero value is obtained from non vanishing mutually canceling quantities, which is not the case here. For example, pick $\epsilon = 1 \times 10^{-15}$ and $x = (1 + \epsilon, 1 - \epsilon)^t$ and $y = (1 - \epsilon, 1 - \epsilon)^t$. Working with a machine precision $\epsilon_M = 2.220446049250313081 \times 10^{-16}$, we get $f(y) - f(x) = -9.370065180311890068 \times 10^{-28}$.

By just modifying slightly the function: $\tilde{f}(x) = 100 + f(x)$, we see that $\tilde{f}(x^*) = 100$, so that close to x^*, the descent test involves comparing values close to 100, and roughly, when $\|x^{k+1} - x^k\|$ becomes close to the square root of the machine precision, $f(x^{k+1}) - f(x^k)$ should be of the order of the machine precision. For example, pick $\epsilon = 1 \times 10^{-9}$ and $x = (1 + \epsilon, 1 - \epsilon)^t$ and $y = (1 - \epsilon, 1 - \epsilon)^t$. This time, *numerically*, $f(y) - f(x) = 0$.

Next, we rewrite f as $\hat{f}(x) = 100 + f(x) - 100$, and we get the same limitation as for $\tilde{f}$, illustrating that even if the objective function value at the optimum vanishes, when it involves cancellations of non vanishing quantities, numerical difficulties will arise.

1.2 Failure of Armijo backtracking line search

We insist to present a further warning concerning such numerical problems. We observed that a simple code, using a simple Armijo backtracking line search would reach ϵ_M on the function $\tilde{f}$ while a very similar code failed. Close examination of the implementation revealed that the line search was coded as

$$\text{while } f(x_k + \theta d_k) - f(x_k) > \tau_0 \nabla f(x_k) d_k, \text{ do } \theta \leftarrow \theta/2$$

for the apparently unsuccessful code and

$$\text{while } f(x_k + \theta d_k) > f(x_k) + \tau_0 \nabla f(x_k) d_k, \text{ do } \theta \leftarrow \theta/2$$

for the "successful" one. This strange behavior is explained by the fact that for the unsuccessful code, the "while" condition is always true since the left hand side *numerically* vanishes ($f(x_k + \theta d_k)$ is always numerically equal to $f(x_k)$) and the right hand side is negative. For the "successful" code, the right hand side is equal to $f(x_k)$ since $\tau_0 \nabla f(x_k) d_k$ numerically vanishes compared with $f(x_k)$, and as above, $f(x_k)$ is numerically equal to $f(x_k + \theta d_k)$, so the condition is always false.

When used with Newton's or quasi-Newton directions, one expects to use a unitary step size close to a solution. If $f(x + d)$ does not evaluate smaller than $f(x)$, the overall algorithm fails, and the actual reason is often that the descent test is unreliable due to cancellation errors. In such cases, plain Newton method would converge nicely without ever bothering with the step size. By cleaning the descent test from numerical artifacts, the nice local behavior of such variants of Newton's method is preserved.

2 Accurate evaluation of descent

A solution to the numerical problem described in the previous section is offered by tools derived from *automatic differentiation*. It is a technique that allows to compute derivatives of a function by systematic application of differentiation rules, including in particular the chain rule. It may be viewed as a transformation of the *computational graph* associated with the function into the computational graph of the derivative. Given a set of *elementary* mathematical functions $(+, -, *, /, \ln, \sin, \cos, \ldots)$ together with their differentiation rule (e.g. $\cos(u)' = -\sin(u) \cdot u'$) and the chain rule $\frac{\partial f(g(x,y))}{\partial x} = \left.\frac{\partial f(z)}{z}\right|_{z=g(x,y)} \cdot \frac{\partial g(x,y)}{\partial x}$, one may automatically reconstruct (implicitly or explicitly) a computational graph of derivatives for a given function. For an introduction to automatic differentiation biased towards optimization, see for example [18], [17], and [10].

2.1 Automatic descent computation

Section 1.2 summarized the problem as accurately computing $\Delta_{\theta d} f(x) = f(x+\theta d) - f(x)$ (with $\theta > 0 \in \mathbb{R}$ and $d \in \mathbb{R}^n$) when x gets close to an accumulation point of $\nabla f(x)$. Defining $h(\theta) = f(x+\theta d)$, we have a real univariate function for which we want to obtain $h(\theta) - h(0)$, the latter term dominating the MacLaurin development of $h(\theta)$.

For any real univariate function $h(x)$, we can use a technique similar to the *forward mode* of automatic differentiation to compute an n-order Taylor polynomial approximation of $h(x)$ around some anchor point a ([1], [3], [13]). This technique propagates the set of $n + 1$ Taylor polynomial coefficients through the computational graph. In our case, we

are interested in accurately computing $h(\theta) - h(0)$; we then acquire the computational graph of $h(\theta)$ and use the Taylor coefficient propagation technique to gather coefficients of the development around $a = 0$. The 0-order coefficient being $h(0)$, $h(\theta) - h(0)$ can then be computed by suppressing the first element from the coefficient vector, and cumulating the other Taylor terms.

Following the reasoning from the introduction, for x_k in some neighbourhood close to the accumulation point x^*, the descent $\Delta_{\theta d} f(x_k)$ is dominated by the second order term of its Taylor development. We are then tempted to systematically compute coefficients up to the second order and approximate descent using a two-terms Taylor approximation. However, since this observation is only locally true, we need a computation scheme that works up to an order appropriate to allow for precise evaluation of $\Delta_h f(x)$. With many Taylor coefficient propagation tools ([1]), this order must be fixed before the actual propagation.

In fact, one does not wish to resort to plain Taylor series to evaluate a given function; efficient implementations take advantage of the structure of the function, and often of faster converging series or processes. Hence, for a complex function, it is next to impossible to know in advance the order of development that is required to reach the desired accuracy at every intermediate node, and if the evaluation proves inaccurate, there is no possibility to reuse the computation. In that case, higher-order coefficients need to be propagated from scratch. We then devise a software tool that dynamically adapts the order of the information propagation to a given numerical precision.

2.2 Automatic finite differences (AFD)

As it is well known, finite difference operators bear close resemblance to derivative operators. The actual computation of $\Delta_d f(x)$ may be decomposed into elementary parts corresponding to a sample of mathematical functions, as is the case for derivatives. Some complications may occur due to the fact that finite differences of usual function such as the exponential or the logarithm do not correspond to usual functions; their derivatives, of course, are usual functions.

For example, consider the logarithmic function. We have $\Delta_d \log(x) = \log(x+d) - \log(x) = \log(1 + (x/d)) = \Delta_{x/d} \log(1)$. Fortunately, most systems/languages include the function $log1p(z) = \log(1+z)$, so that nodes including logarithmic function are already computable using usual software.

Unfortunately, the situation is less favourable for other functions, such as the exponential function. Consider $\Delta_d \exp x = \exp(x+d) - \exp x = (\exp x)(\exp d - 1)$. Clearly, it suffices to define the function $\mathcal{E}(d) \stackrel{def}{=} \exp d - 1$ to express $\Delta_d \exp x$. Direct computation of $\mathcal{E}$

avoids the cancellation errors caused by the fact that, close to $d = 0$, $\exp d$ is close to 1. Since the function $\mathcal{E}$ is not re-defined in well known environments, we have to provide an implementation to evaluate it. Our implementation resort to a limited Taylor series expansion, with a dynamic adjustment of the order. However, for efficiency purposes, the function $\mathcal{E}$ should be implemented cleverly, as is the case for the usual implementation of the exponential function. Also, differences of other usual functions may be reduced to some unitary case such as `log1p`, or $\mathcal{E}$, whose efficient implementation should be added to the standard libraries.

2.2.1 An example

As an example, suppose we apply Newton-Raphson's method to $\min f(x) = \exp(x^2)$ on a machine working to $\epsilon_M = 2.220446049250313 \times 10^{-16}$. For $x = 1 \times 10^{-9}$, both $[f(x + d_N) - f(x)]$ and $[f(x - d_N) - f(x)]$ evaluate to 0 for $d_N = -f'(x)/f''(x)$.

Now, let us evaluate $\Delta_d f(x)$. This will serve as an introduction to the chain rule using the function $f(x) = f_1(f_2(x))$ with $f_1(x) = \exp x$ and $f_2(x) = x^2$. $\Delta_d f(x) = \Delta_{\Delta_d f_2(x)} f_1(f_2(x))$. Now, $z = \Delta_d f_2(x)$ works out to be $z = 2x + d$, so that, taking into account that $f_1(x) = \exp x$, $\Delta_z f_1(f_2(x)) = \exp[f_2(x)](\mathcal{E}(z))$. Again for $x = 1 \times 10^{-9}$, using this technique with $\mathcal{E}(z)$ approximated by $z + z^2/2 + z^3/6$, we get $\Delta_{d_N} f(x) = -1 \times 10^{-18}$ and $\Delta_{-d_N} f(x) = 3 \times 10^{-18}$.

2.2.2 The chain rule for finite differences

The key to forward-mode automatic differentiation is the application of the derivation chain rule. To forward-propagate finite differences through a computation graph, such a chain rule is defined. It differs from the derivation chain rule in that the finite step propagates into composite functions, while its limit in the case of derivatives is constant.

Therefore,

$$\Delta_h f_1(f_2(x)) = \Delta_{\Delta_h f_2(x)} f_1(z)\big|_{z=f_2(x)} \tag{2.2}$$

Primitives for computation of finite differences of elementary functions at each node of the computation graph are easily derived from rule 2.2.

2.2.3 Algorithmic complexity

Take function $f(x)$, for which we describe the cost of application as $C(f)$. Previous works in automatic differentiation ([8], [9]) have shown that the cost of evaluating its

gradient using the forward mode is $O(nC(f))$. Effectively, evaluating the gradient implies a forward traversal of the computation graph of $f(x)$; at each node $z_k = f_k(z_i, z_j) \in \mathbb{R}^m$ $(m \geq n)$ of this graph, a $\mathbb{R}^n$ vector term is added to the then-partially computed gradient, yielding a cost of $O(nC(z_k))$ at each node. When such a traversal process is used to compute $\Delta_d f(x)$ instead of $\nabla f(x)$, the partial finite difference result being propagated is a $\mathbb{R}^m$ vector. Hence, the cost of computing a finite difference of $f(x)$ is proportional to that of computing an application of $f(x)$. A bound on the proportionality constant can be inferred by taking a look at the elementary operation for which the finite difference formula is the most expensive. This node would be the multiplication node $z_k = z_i \times z_j$, for which the finite difference propagation formula is

$$\Delta_h(z_i \times z_j) = z_i \times \Delta_h z_j + [\Delta_h z_i] \times z_j + [\Delta_h z_i] \times [\Delta_h z_j]$$

We then see that the computation of one application of $f(x)$ and one finite difference cost no more than 5 times the cost of one application.

3 Implementation issues

We have implemented an automatic differentiation framework in the Scilab [19] environment. Our implementation acquires the computation graph of functions using operator overloading, like ADOL-C [12] and ADMAT [7] [6]. This scheme is opposed to graph acquisition by source precompilation, which is the method used in ADIFOR [4]. This framework allowed us to implement automatic finite differences in the same fashion as forward-mode automatic derivation (for further discussion, see [18], [5], [11]), using calculations derived from equation 2.2.

Obviously, the building and traversal of the computation graph implies a significant software overhead when compared to a simple subtraction, especially in context of an operator-overloading AD framework. Two cases are to be considered here. On one hand, for an optimization code where gradients of the function are already computed using some mode of automatic derivation, most of the overhead is already incurred and "paid for". Effectively, the computation graph that AFD works on is that of $f(x)$, the same as that used by the gradient evaluator. Since traversing the graph is usually a process less expensive than building the graph, the running time offset appears acceptable. On the other hand, if derivatives are computed through user code or somehow approximated, the construction of an explicit computation graph may be omitted, alleviating the runtime overhead. Indeed, when tapped using operator overloading, the function evaluation process implicitly corresponds to a forward traversal of the computation tree. Overloaded operators have then the task of computing the finite difference using a set step vector and primitives derived from equation (2.2).

For both these cases, we propose a heuristic to avoid the automatic computation when the iterate is still far enough from the accumulation point. We've stated that the numerical problems we've described arise when $\frac{\Delta_d f(x)}{\min\{|f(x)|,|f(x+d)|\}}$ evaluates on the order of the square root of the machine precision, which in our case is on the order of 10^{-16}. Neglecting constants, we start using automatic finite differencing as soon as the usual computation (simple subtraction) of $f(x+d) - f(x)$ drops below the quartic root of the machine precision, here 10^{-4}. Otherwise, the subtraction result is used.

4 Numerical experiments

In order to assess the effectiveness of our technique, we report on experimentations on three public procedures taken from the T.C. Kelley public optimization codes [14]. We present observation on `ntrust`, a dogleg trust region variant, `cgtrust`, a Steihaug-Toint conjugate gradient trust region variant, and `bfgswopt`, a limited memory quasi-Newton variant, all available from `http://www4.ncsu.edu/~ctk/matlab\_darts.html`. Actually, we produced Scilab translations using the `mfile2sci` utility. Slight alterations to the codes were added to avoid crashing on divisions by zero: we trap the offending denominator, and exit gracefully.

We use the Moré, Garbow and Hillstrom [16] collection of test functions adapted to Matlab (available at `ftp://ftp.mathworks.com/pub/contrib/v4/optim/uncprobs.tar`) by Chaya Gurwitz, Livia Klein, and Madhu Lamba and further translated to Scilab again using the `mfile2sci` utility and some hand tuning.

We stress here that we do not intend to provide comparisons between the Kelley's codes, but to illustrate that our technique may improve *any* "off-the-shelf" code by simply replacing descent tests $f(x+d) - f(x)$ by a suitable call to $\Delta_d f(x)$. Since we are interested to accurate solutions, we set our stopping criterion as $\|\nabla f(x)\| \leq 10^{-12}$.

We illustrate the performance enhancement bound to automatic finite differences using *progress diagrams*. These are graphs that illustrate $\log \|\nabla f(x_k)\|$ as a function of k. Our stopping criterion $\|\nabla f(x_k)\| < 10^{-12}$ is presented as a dashed lined. Figure 2.1 presents progress diagrams for some problems where the AFD-enabled code converges more quickly than the plain code.

On many problem instances, both implementations behave identically. This reflects the fact that for many of those test problems, the minimum objective function value is zero, and no cancellation error occurs. Nevertheless, a few of those tests benefit by using automatic finite differencing. For instance, for the second problem of the collection (the Freudenstein and Roth function), all variants approach a point with an objective function

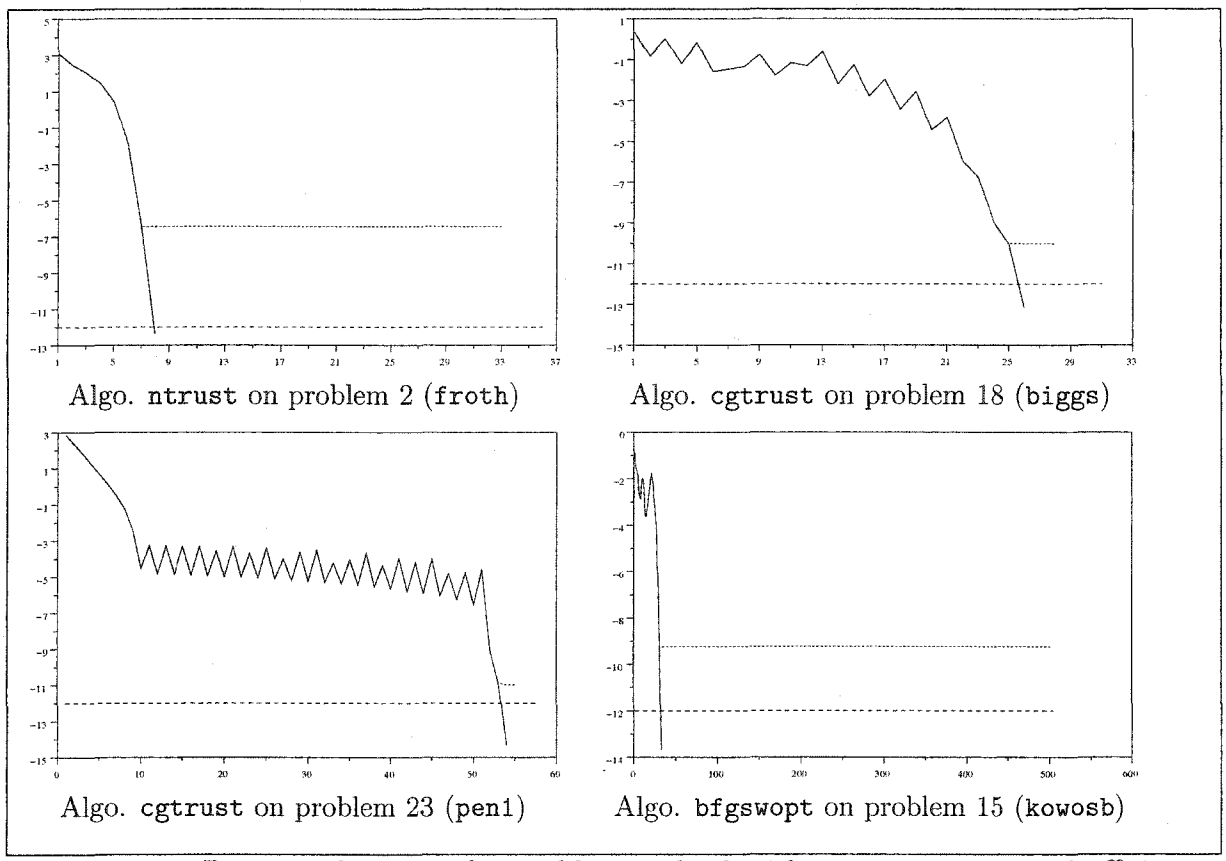


Figure 2.1: Progress diagrams for problems solved with more accuracy and efficiency using AFD. The horizontal dashed line indicates the stopping criterion accuracy that must be reached by the optimizers. The other two lines plot $\log \|\nabla f(x)\|$ – a measure of the distance to the solution – for the compared variants of the procedure : the dotted line is bound to the original procedure and the plain line is bound to the variant set up to use automatic finite differences to compute the descent test.

value close to 49. All three plain codes are stuck to a precision close to the square root of the machine precision, and using AFD technique, succeed in reaching the full machine precision. Similar behavior is observed for many other problem instances. A less striking yet positive impact is remarked for some instances where either the accurate solution is reached in a lesser number of iterations, either the optimizer stalls closer to the stopping criterion. Finally, for a small subset of problems, the original code performs better than the AFD-altered optimizer. Further analysis lead us to bind this behavior to a failure of the direction calculation scheme at the iterate accurate descent evaluation leads to.

Tables 2.1, 2.2 and 2.3 recall our empirical results for problems where the behavior of the original optimizers and that of the AFD-enabled procedure differ. We mark stalls by suffixing an asterisk to the iteration count and we emphasize the gradient norm for

the variant that stops the closest to the set criterion. In addition, we mark crashes[1] by suffixing a cross to the iteration count.

Conclusion

We have observed a phenomenon limiting *all* descent algorithms, namely the impossibility to assess descent further than the square root of the machine precision. For some instances, techniques such as non-monotone line searches may circumvent the problem. We proposed a novel solution based on evaluating descent directly, without comparing function values. Some limited numerical experience showed that our approach succeeds in stabilizing "off-the-shelf" codes for problem instances affected by this phenomenon. Many other difficulties may arise, such as Newton iterations stuck by something else than the descent test. We successfully reduce the numerical noise from descent tests.

The promising behavior of our experimental implementation is very encouraging, to the extent that we are confident that this tool might become a staple of high-precision applications. This implementation is part of the SCIAD library, a set of automatic differentiation tools for the Scilab environment. Full source code of SCIAD is available at `http://benoithamelin.com/sciad/sciad.html` and main releases will be submitted to INRIA for posting on the main Scilab web site [19].

References

[1] Claus Bendtsen and Ole Stauning. Tadiff, a flexible c++ package for automatic differentiation using taylor series expansion. Technical Report IMM-REP-1997-07, Department of Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby – Denmark, April 1997.

[2] Martin Berz. Algorithms for higher order automatic differentiation in many variables with applications to beam physics. In George F. Corliss and Andreas Griewank, editors, *Automatic Differentiation of Algorithms – Theory, Implementation and Applications*. SIAM, 1991.

[3] Martin Berz and Georg Hoffstätter. Computation and application of taylor polynomials with interval remainder bounds. *Reliable Computing*, 4(89–97), 1998.

[1] What we call a *crash* is the termination of the optimization procedure due to an abnormal numerical condition, such as the trust region radius being reduced to zero or the line search failing.

Problem	Regular variant			AFD variant		
	Iter.	$f(x^*)$	$\|\nabla f(x^*)\|$	Iter.	$f(x^*)$	$\|\nabla f(x^*)\|$
1. rose	37	2.02×10^{-30}	$\mathbf{5.12 \times 10^{-14}}$	38	6.59×10^{-27}	2.08×10^{-13}
2. froth	33†	49.0	3.60×10^{-7}	8	49.0	$\mathbf{4.26 \times 10^{-13}}$
10. meyer	1001*	7.09×10^{4}	1.09×10^{5}	1001*	6.09×10^{4}	89.8
12. box	46†	0.0756	$\mathbf{8.52 \times 10^{-9}}$	51*	0.0756	2.27×10^{-7}
14. wood	4679	1.80×10^{-26}	3.90×10^{-13}	4682	1.89×10^{-27}	$\mathbf{1.87 \times 10^{-13}}$
16. bd	25*	8.58×10^{4}	2.45×10^{-6}	101*	8.58×10^{4}	$\mathbf{8.14 \times 10^{-12}}$
17. osb1	70†	5.46×10^{-5}	9.34×10^{-10}	46	5.46×10^{-5}	$\mathbf{1.35 \times 10^{-13}}$
18. biggs	233†	5.66×10^{-3}	$\mathbf{1.41 \times 10^{-10}}$	501*	5.29×10^{-3}	2.49×10^{-3}
21. rosex	37	4.04×10^{-30}	$\mathbf{7.25 \times 10^{-14}}$	38	1.32×10^{-26}	2.94×10^{-13}
24. pen2	179	9.38×10^{-6}	9.42×10^{-13}	174	9.38×10^{-6}	$\mathbf{8.63 \times 10^{-13}}$
32. lin	29†	2.00	7.08×10^{-9}	3	2	$\mathbf{9.27 \times 10^{-16}}$
34. lin0	39†	3.89	1.10×10^{-8}	7	3.89	$\mathbf{2.55 \times 10^{-13}}$

Table 2.1: Comparative numerical results for algorithm `ntrust`.

Problem	Regular variant			AFD variant		
	Iter.	$f(x^*)$	$\|\nabla f(x^*)\|$	Iter.	$f(x^*)$	$\|\nabla f(x^*)\|$
2. froth	13†	49.0	5.24×10^{-7}	11	49.0	$\mathbf{5.69 \times 10^{-14}}$
3. badscp	158†	2.72×10^{-20}	6.42×10^{-12}	1001*	2.72×10^{-20}	6.42×10^{-12}
6. jensam	12†	0.265	1.61×10^{-12}	10	0.265	$\mathbf{1.05 \times 10^{-14}}$
10. meyer	754†	89.9	49.9	1001*	87.9	$\mathbf{6.31 \times 10^{-5}}$
16. bd	13†	8.58×10^{4}	1.75×10^{-5}	1001*	8.58×10^{4}	$\mathbf{7.55 \times 10^{-11}}$
18. biggs	27†	5.66×10^{-3}	9.44×10^{-11}	25	5.66×10^{-3}	$\mathbf{7.24 \times 10^{-14}}$
23. pen1	54†	2.25×10^{-5}	1.02×10^{-11}	53	2.25×10^{-5}	$\mathbf{4.38 \times 10^{-15}}$
32. lin	6†	2.00	2.18×10^{-10}	3	2.00	$\mathbf{9.53 \times 10^{-16}}$

Table 2.2: Comparative numerical results for algorithm `cgtrust`.

Problem	Regular variant			AFD variant		
	Iter.	$f(x^*)$	$\|\nabla f(x^*)\|$	Iter.	$f(x^*)$	$\|\nabla f(x^*)\|$
2. froth	101*	49.0	2.13×10^{-9}	13	49.0	$\mathbf{8.57 \times 10^{-14}}$
6. jensam	101*	0.265	3.47×10^{-10}	15	0.265	$\mathbf{2.51 \times 10^{-13}}$
8. bard	101*	8.21×10^{-3}	6.29×10^{-12}	31	8.21×10^{-3}	$\mathbf{5.00 \times 10^{-15}}$
15. kowosb	501*	3.08×10^{-4}	5.92×10^{-10}	33	3.08×10^{-4}	$\mathbf{2.12 \times 10^{-14}}$
16. bd	188†	8.58×10^{4}	2.51×10^{-5}	200*	8.58×10^{4}	$\mathbf{8.14 \times 10^{-12}}$
18. biggs	59†	5.66×10^{-3}	1.63×10^{-11}	57	5.66×10^{-3}	$\mathbf{8.50 \times 10^{-13}}$
23. pen1	192†	2.25×10^{-5}	2.21×10^{-12}	191	2.25×10^{-5}	$\mathbf{5.85 \times 10^{-16}}$
24. pen2	241*	9.38×10^{-6}	1.39×10^{-11}	240	9.38×10^{-6}	$\mathbf{5.37 \times 10^{-13}}$
33. lin1	201*	2.14	6.68×10^{-9}	107	2.14	$\mathbf{5.79 \times 10^{-13}}$

Table 2.3: Comparative numerical results for algorithm `bfgswopt`.

[4] Christian H. Bischof, Alan Carle, Paul D. Hovland, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0 user's guide (Revision D). Technical report, Mathematics and Computer Science Division Technical Memorandum no. 192 and Center for Research on Parallel Computation Technical Report CRPC-95516-S, 1998.

[5] Christian H. Bischof, Paul D. Hovland, and Boyana Norris. Implementation of automatic differentiation tools. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 98–107. ACM Press, 2002.

[6] Thomas F. Coleman and Arun Verma. ADMAT: An automatic differentiation toolbox for MATLAB. Technical report, Computer Science Department, Cornell University, 1998.

[7] Thomas F. Coleman and Arun Verma. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Trans. Math. Softw.*, 26(1):150–175, 2000.

[8] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[9] Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In Panos M. Pardalos, editor, *Complexity in Nonlinear Optimization*, pages 128–161. World Scientific Publishers, 1993.

[10] Andreas Griewank. Computational differentiation and optimization. In J.R. Birge and K.G. Murty, editors, *Mathematical Programming : State of the Art 1994*, pages 102–131. The University of Michigan, Michigan, 1994.

[11] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* Society for Industrial & Applied Mathematics, January 1992.

[12] Andreas Griewank, David Juedes, Hristo Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[13] Gershon Kedem. Automatic differentiation of computer programs. *ACM Transactions on Mathematical Software*, 4(2), June 1980.

[14] Tim Kelley. *Iterative Methods for Optimization.* SIAM, 1999.

[15] David G. Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[16] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Algorithm 566: FORTRAN subroutines for testing unconstrained optimization software [C5 [E4]]. *ACM Transactions on Mathematical Software*, 7(1):136–140, March 1981. [ftp://ftp.mathworks.com/pub/contrib/v4/optim/uncprobs.tar].

[17] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[18] Louis B. Rall and George F. Corliss. An introduction to automatic differentiation. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 1–17. SIAM, Philadelphia, PA, 1996.

[19] Scilab group. *Ψlab 2.7*. Institut National de Recherche en Informatique et Automatique - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 - LE CHESNAY Cedex - FRANCE, email : Scilab@inria.fr, 2003. [http://www-rocq.inria.fr/scilab/].

2.2 Complément – Formules de propagation des différences finies

À l'instar de l'appendice A, nous présentons pour compléter la liste des formules employées pour propager un vecteur intermédiaire de différences finies à l'aide d'une visite en postordre du graphe d'évaluation d'une fonction. Ces formules s'appuient sur la règle 2.2. Notez que les opérations dont le nom est suffixé d'une astérisque désignent une opération appliquée élément par élément des vecteurs ou matrices données comme opérandes.

Opérateur primitif	Formule de propagation des différences finies
Addition $f(x) = p(x) + q(x)$	$\Delta_h f(x) = \Delta_h p(x) + \Delta_h q(x)$
Opposé $f(x) = -p(x)$	$\Delta_h f(x) = -\Delta_h p(x)$
Transposition $f(x) = p(x)^T$	$\Delta_h f(x) = [\Delta_h p(x)]^T$
Produit $f(x) = p(x)q(x)$	$\Delta_h f(x) = \Delta_h p(x)\Delta_h q(x) + p(x)\Delta_h q(x) + [\Delta_h p(x)]q(x)$
Produit* $f(x) = p(x) .* q(x)$	$\Delta_h f(x) = \Delta_h p(x) .* \Delta_h q(x) + p(x) .* \Delta_h q(x) + [\Delta_h p(x)] .* q(x)$
Inversion $f(x) = p(x)^{-1}$	$\Delta_h f(x) = -[\Delta_h p(x) + p(x)]^{-1}[\Delta_h p(x)][p(x)]^{-1}$
Inversion* $f(x) = 1 ./ p(x)$	$\Delta_h f(x) = -\Delta_h p(x) ./ [p(x)\Delta_h p(x) + p(x)^2]$
Exponentielle $f(x) = K^{p(x)}$ (K est une constante)	$\Delta_h f(x) = K^{p(x)} \left[\frac{\Delta_h p(x)}{\log K}\right] \sum_{i=0}^{\infty} \frac{1}{(i+1)!} \left[\frac{\Delta_h p(x)}{\log K}\right]^i$
Exponentielle* $f(x) = \exp_K p(x)$	Même formule que pour l'exponentielle régulière, sauf que les produits, élévations exponentielles et divisions sont remplacés par les opérations élément par élément correspondantes.
Monôme $f(x) = p(x)^K$ (K est une constante)	On procède en trois temps. D'une part, si l'exposant est négatif, on compose le monôme avec une inversion et on poursuit avec un exposant $K = E + F$. Le terme E de l'exposant constitue sa partie entière. Une telle opération est décomposée en un produit récursif de carrés, auquel on applique les formules de propagation des différences finies du produit et de l'élévation au carré. Cette dernière s'exprime comme : $$\Delta_h p(x)^2 = p(x)\Delta_h p(x) + [\Delta_h p(x)]p(x) + [\Delta_h p(x)]^2$$ Le terme F de l'exposant constitue sa partie fractionnaire. On exprime alors $p(x)^F$ à l'aide d'un développement de Taylor autour de $a = p(x)$. Assumant que $p(x)^{K-i}$ existe pour tout i, on déduit : $$\Delta_h p(x)^F = \sum_{i=1}^{\infty} \frac{1}{i!} \left[\prod_{j=1}^{i} (K - j + 1)\right] p(x)^{K-i}[\Delta_h p(x)]^i$$ Comme $f(x) = p(x)^E p(x)^F$, on obtient la différence finie finale à l'aide de la règle du produit.
Monôme* $f(x) = \text{pow}_K x$	On emploie le même algorithme que pour les monômes réguliers, sauf qu'on remplace les produits, divisions et élévations exponentielles par les opérations élément par élément correspondantes.
Logarithme* naturel $f(x) = \log p(x)$	$\Delta_h f(x) = \log[1 + \Delta_h p(x) ./ p(x)]$

L'élaboration du calculateur de différences finies sur une base logicielle commune au module de dérivation constitue un point en la faveur du design basé sur le concept de visiteur. Cet aspect nous a permis de nous concentrer sur le développement des formules et des calculs associés aux différences finies : le code définissant la mécanique de visite et la construction du graphe a été réutilisé sans effort supplémentaire.

En réalisant les expériences numériques dont les résultats ont été rapportés, nous avons réalisé que les codes d'optimisation de Kelley obtiennent les dérivées de second ordre des diverses fonctions par la méthode numérique des différences finies. Nous nous sommes alors interrogés sur l'intérêt d'employer les différences finies automatiques pour évaluer des dérivées directionnelles. Le chapitre qui suit approfondit et expérimente cette idée.

CHAPITRE 3

DÉRIVATION À L'AIDE DES DIFFÉRENCES FINIES AUTOMATIQUES

3.1 Introduction

Nous rappelons que nous considérons l'implantation d'algorithmes de résolution de programmes non linéaires sans contrainte :

$$\min_{x \in \mathbb{R}^n} f(x) \tag{3.1}$$

Les hypothèses quant aux propriétés de f et les conditions d'optimalité d'un minimum local ont été décrites en introduction.

Les algorithmes qui nous intéressent sont dits *de descente* : ils génèrent une suite $\{x_k\}$ de points successifs tels que $f(x_{k+1}) < f(x_k)$. Cette descente doit être contrôlée selon un certain critère de *suffisance*. Par exemple, pour les algorithmes de recherche linéaire, la condition de suffisance de la descente s'exprime par le critère d'Armijo :

$$f(x_{k+1}) - f(x_k) \leq \tau_0 \nabla f(x_k)(x_{k+1} - x_k)$$

Un autre exemple : les algorithmes de régions de confiance contrôlent le rayon de cette dernière en comparant la descente de la fonction objectif à celle d'un modèle quadratique

de cette fonction :

$$\frac{f(x_{k+1}) - f(x_k)}{m(x_{k+1}) - m(x_k)} \leq \tau$$

Tous ces mécanismes de contrôle en viennent à évaluer la descente de la fonction objectif, qui s'exprime comme une différence finie :

$$f(x_{k+1}) - f(x_k) = f(x_k + d_k) - f(x_k) = \Delta_{d_k} f(x_k)$$

Nous avons discuté au chapitre 2 que, proche d'un point stationnaire, l'évaluation naïve de telles différences finies est entachée d'erreurs d'annulation. La solution réside dans l'utilisation d'un logiciel de calcul des différences finies s'appuyant sur le graphe d'évaluation de la fonction objectif, à la mode de la différentiation automatique, système que nous avons dénommé *différences finies automatiques.*

Lorsque le programme d'évaluation du gradient et du Hessien de f n'est pas explicitement implanté par l'usager, on peut recourir à la différentiation automatique pour évaluer ces quantités, de même qu'à la technique des différences finies. Par exemple, on peut obtenir une approximation de la dérivée de f dans la direction d par la différentiation finie à gauche :

$$\nabla f(x)d = \lim_{\epsilon \to 0} \frac{f(x + \epsilon d) - f(x)}{\epsilon} = \lim_{\epsilon \to 0} \frac{\Delta_{\epsilon d} f(x)}{\epsilon} \tag{3.2}$$

Évidemment, cette technique numérique souffre des mêmes problèmes que l'évaluation de la descente. De ce fait, nous nous proposons d'expérimenter le calcul de dérivées directionnelles d'ordre 2 à l'aide des différences finies automatiques. Nous comparerons les résultats à ceux obtenus de l'exécution de ces mêmes expériences à l'aide des différences finies numériques, ainsi qu'à l'aide du mode en préordre de la différentiation automatique.

3.2 Diverses approches de dérivation

On note dans un premier temps que l'usage des différences finies constitue une approche fondamentalement inefficace du calcul de gradients ou de Hessien complets. En effet,

lorsqu'on emploie la formule basée sur la limite (équation (3.2)) pour calculer le gradient de $f : \mathbb{R}^n \rightarrow \mathbb{R}$, il faut évaluer cette formule pour chacun des n vecteurs e_i de la base canonique du domaine. De ce fait, considérant que l'évaluation de f en x ait un coût en temps d'exécution $C(f)$, le coût d'évaluation de son gradient est $C(\nabla f) \in O(nC(f))$.

Ainsi, l'usage des différences finies n'est proposé que pour le calcul de dérivées directionnelles. Comme décrit dans les algorithmes de la section 3.3, les algorithmes d'optimisation qui nous intéressent n'ont besoin d'obtenir d'information de courbure (dérivée de second ordre) que dans une direction précise à chaque itération interne. Par ailleurs, notre librairie de problèmes expérimentaux [MGH81] offre le code d'évaluation du gradient des fonctions objectifs de chacun des problèmes.

En résumé, nous nous penchons sur la comparaison empirique de la performance de trois méthodes de dérivation directionnelle de second ordre :

Différences finies numériques On emploie une définition alternative de la dérivation directionnelle par approximation du calcul de limite, définition dite des *différences finies symétriques*, reconnue comme ayant un comportement numérique plus stable.

$$\nabla^2 f(x)d = \lim_{\epsilon \rightarrow 0} \frac{\nabla f\left(x + \frac{\epsilon}{2}d\right) - \nabla f\left(x - \frac{\epsilon}{2}d\right)}{\epsilon} \tag{3.3}$$

Différences finies automatiques On emploie l'approche des différences finies à droite (équation (3.2)), puisque cette définition reprend les quantités évaluées par le visiteur implanté pour les calculs de descente. On espère que l'avantage des différences finies symétrique sera contré par la meilleure précision du processus de calcul des différences finies.

Différentiation automatique en préordre Ayant le graphe d'évaluation de $\nabla f(x)$, on peut calculer $\nabla^2 f(x)d = \nabla(\nabla f(x)d)$. Comme le Hessien des fonctions au moins deux fois différentiables est une matrice symétrique, on a

$$\nabla^2 f(x)d = [\nabla(\nabla f(x)d)]^T \tag{3.4}$$

Incidemment, la quantité $\nabla[\nabla f(x)d]$ constitue exactement le résultat calculé par un logiciel de dérivation automatique en préordre paramétré par le vecteur d^T de pondération des lignes.

3.3 Algorithmes d'optimisation

Nous avons expérimenté les trois variantes de dérivation sur deux procédures d'optimisation sans contrainte. Pour chacune, la direction du prochain pas est acquise par une variante de l'algorithme du gradient conjugué, dont les itérations emploient les dérivées directionnelles de second ordre.

La première procédure consiste en une méthode de Newton tronquée. La recherche linéaire consiste en une simple boucle de test du critère d'Armijo, implantée de manière robuste à l'aide des différences finies automatiques (voir chapitre 2). Le pseudo-code de la procédure est donnée comme l'algorithme 3.1.

La seconde procédure consiste en une procédure de régions de confiance de Steihaug-Toint. La réduction de l'objectif entre deux itérations successives est mesurée à l'aide des différences finies automatiques. Cette réduction est comparée à celle prévue selon un modèle quadratique de la fonction objectif, de manière à réduire ou à augmenter le rayon de la région de confiance. Le pseudo-code de cette procédure est donné par l'algorithme 3.3. On mentionne au passage que ce code ne s'apparente ni ne s'inspire de celui de la procédure `cgtrust` expérimentée au chapitre 2. Ceci explique les différences entre les résultats obtenus de l'exécution de `cgtrust` et de la procédure ci-introduite, lesquels sont comparés dans le cadre du tableau 3.1 (le lecteur se référera à la section 3.4.1 pour la caractérisation des *pannes* et leur indication sur le tableau).

À travers les différents algorithmes, on définit les fonctions suivantes appartenant à la librairie SCIAD :

Entrées :

1. $x_0 \in \mathbb{R}^n$, le point de départ.
2. Procédure `f(x)` d'évaluation de $f : \mathbb{R}^n \to \mathbb{R}$ en un point x. La surcharge des opérateurs permet d'obtenir le graphe d'évaluation $G_{f(x)}$ en passant un argument $\mathcal{X}$ du type approprié de SCIAD.
3. Procédure `g(x)` d'évaluation de $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ en un point x. Encore une fois, l'usage d'un argument $\mathcal{X}$ d'un type abstrait de données défini dans le cadre de SCIAD permet d'obtenir le graphe d'évaluation du gradient, $G_{\nabla f(x)}$.
4. Procédure `h(x, d, G)` d'évaluation du Hessien de f au point x, dans la direction d; le graphe d'évaluation du gradient est passé comme le paramètre G.
5. Précision $\varepsilon \in \mathbb{R}$ à atteindre sur le critère d'arrêt.

Sortie : $x^* \in \mathbb{R}^n$, un minimum local de f.

Code :

$\tau_0 = 1/4$
$\mathcal{X} \leftarrow$ `SCIAD`(x_0) – Construction d'un objet propre à la construction des graphes d'évaluation.
$G_{f(x)} \leftarrow$ `f`$(\mathcal{X})$
$G_{\nabla f(x)} \leftarrow$ `g`$(\mathcal{X})$
$\nabla f \leftarrow$ `VALEUR`$(G_{\nabla f(x)})$
$\mathcal{N} \leftarrow 0$
TANT QUE $\|\nabla f\| > \varepsilon$ **FAIRE**
  $d \leftarrow$ `GC_NEWTON`$(\mathtt{h}, \mathtt{VALEUR}(\mathcal{X}), \nabla f, G_{\nabla f(x)})$ (voir algorithme 3.2)
  $\theta \leftarrow 1$
  TANT QUE `DFA`$(G_{f(x)}, \theta d) > \tau_0 \theta \nabla f d$ **FAIRE**
    $\theta \leftarrow \theta/2$
  FIN
  $\mathcal{X} \leftarrow$ `SCIAD`$(\mathtt{VALEUR}(\mathcal{X}) + \theta d)$
  SI $\|\theta d\| < 0.01\epsilon$ **ALORS**
    $\mathcal{N} \leftarrow \mathcal{N} + 1$
    SI $\mathcal{N} = 10$ **ALORS**
      Rompre la boucle : la procédure est en panne.
    FIN SI
  SINON
    $\mathcal{N} \leftarrow 0$
  FIN SI
  $G_{f(x)} \leftarrow$ `f`$(\mathcal{X})$
  $G_{\nabla f(x)} \leftarrow$ `g`$(\mathcal{X})$
  $\nabla f \leftarrow$ `VALEUR`$(G_{\nabla f(x)})$
FIN
$x^* \leftarrow$ `VALEUR`$(\mathcal{X})$

Figure 3.1: Code d'optimisation axé sur la méthode de Newton tronquée.

Entrées :

1. Procédure `h(x,d,G)` de calcul du Hessien directionnel de f au point x, dans la direction d. Le graphe d'évaluation de $\nabla f(x)$ est passé comme le paramètre G.
2. $x \in \mathbb{R}^n$, le point courant de la boucle de Newton principale.
3. Le gradient au point courant $\nabla f \in \mathbb{R}^n$.
4. Le graphe d'évaluation de $\nabla f(x)$, $G_{\nabla f(x)}$.

Sortie : $d \in \mathbb{R}^n$, la solution optimale du programme quadratique $\min_d q(d)$.

Code :

$p \leftarrow -\nabla f$
$Q_p \leftarrow \mathtt{h}(x, p, G_{\nabla f(x)})$
$\epsilon_1 \leftarrow \min\{0.01, \|\nabla f\|^2\}$
$\epsilon_2 \leftarrow 10^{-12}$
SI $p^T Q_p \leq \epsilon_2 \|p\|^2$ **ALORS**
  $d \leftarrow -\nabla f^T$
SINON
  $d \leftarrow 0$
  $\nabla q \leftarrow \nabla f$
  BOUCLE
    $\vartheta \leftarrow -\frac{\nabla q p}{p^T Q_p}$
    $d \leftarrow d + \vartheta p$
    $\nabla q \leftarrow \mathtt{h}(x, d, G_{\nabla f(x)})^T + \nabla f$
    SI $\|\nabla q\| < \epsilon_1$ **ALORS**
      Rompre la boucle.
    FIN SI
    $\beta \leftarrow \frac{\nabla q Q_p}{p^T Q_p}$
    $p \leftarrow -\nabla q^T + \beta p$
    $Q_p \leftarrow \mathtt{h}(x, p, G_{\nabla f(x)})$
    SI $p^T Q_p \leq \epsilon_2 \|p\|^2$ **ALORS**
      Rompre la boucle.
    FIN SI
  FIN BOUCLE
FIN SI

Figure 3.2: Procédure `GC_NEWTON` : algorithme de minimisation de l'objectif quadratique $q(d) = \frac{1}{2} d^T \nabla^2 f(x) d + \nabla f(x) d$ par la méthode du gradient conjugué.

Entrées :

1. $x_0 \in \mathbb{R}^n$, le point de départ.
2. Procédure `f(x)` d'évaluation de la fonction objectif $f : \mathbb{R}^n \to \mathbb{R}$ en un point x. La surcharge des opérateurs permet d'obtenir le graphe d'évaluation $G_{f(x)}$ de $f(x)$ en passant un paramètre $\mathcal{X}$ du type abstrait de données de SCIAD.
3. Procédure `g(x)` d'évaluation du gradient de f, $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$, en un point x. À nouveau, l'usage d'un paramètre $\mathcal{X}$ du type abstrait de données de SCIAD permet d'obtenir le graphe d'évaluation de $\nabla f(x)$ par surcharge des opérateurs.
4. Procédure `h(x, d, G)` de calcul du Hessien de f au point x, dans la direction d. Le graphe d'évaluation du gradient est passé à ce sous-programme par le paramètre G.
5. $\varepsilon \in \mathbb{R}$, la précision à atteindre sur le critère d'arrêt de la procédure.

Sortie : $x^* \in \mathbb{R}^n$, un minimum local de f.

Code :

$\mathcal{X} \leftarrow$ `SCIAD`(x_0)
$\Delta \leftarrow \|x_0\|$
$G_f \leftarrow$ `f`$(\mathcal{X})$
$G_{\nabla f(x)} \leftarrow$ `g`$(\mathcal{X})$
$\nabla f \leftarrow$ `VALEUR`$(G_{\nabla f(x)})$
$\mathcal{N} \leftarrow 0$
TANT QUE $\|\nabla f\| > \varepsilon$ **FAIRE**
  $d \leftarrow$ `GC_RC`$(\mathtt{h}, \mathtt{VALEUR}(\mathcal{X}), \nabla f, G_{\nabla f(x)}, \Delta)$ (voir algorithme 3.4)
  $\delta_f \leftarrow$ `DFA`$(G_{f(x)}, d)$
  $\delta_m \leftarrow \frac{1}{2} d^T$ `h`$(\mathtt{VALEUR}(\S), d, G_{\nabla f(x)}) + \nabla f d$
  SI $\delta_m = 0$ **ALORS**
    Rompre la boucle : la région de confiance a été réduite à néant.
  FIN SI
  $R \leftarrow \frac{\delta_f}{\delta_m}$
  SI $R < 0.25$ **ALORS**
    $\Delta \leftarrow \Delta/2$
  SINON
    $\mathcal{X} \leftarrow$ `SCIAD`$(\mathtt{VALEUR}(\mathcal{X}) + d)$
    $G_{f(x)} \leftarrow$ `f`$(\mathcal{X})$
    $G_{\nabla f(x)} \leftarrow$ `g`$(\mathcal{X})$
    $\nabla f \leftarrow$ `VALEUR`$(G_{\nabla f(x)})$
  FIN SI
  SI $R > 0.75$ and $|\|d\| - \Delta| < \epsilon$ **ALORS**
    $\Delta \leftarrow 2\Delta$
  FIN SI
  SI $\Delta < 0.01\epsilon$ **ALORS**
    $\mathcal{N} \leftarrow \mathcal{N} + 1$
    SI $\mathcal{N} = 10$ **ALORS**
      Rompre la boucle : la procédure est en panne.
    FIN SI
  SINON
    $\mathcal{N} \leftarrow 0$
  FIN SI
FIN
$x^* \leftarrow x$

Figure 3.3: Code d'optimisation par régions de confiance.

Entrées :

1. Procédure `h(x,d,G)` d'évaluation du Hessien de f au point x, dans la direction d. Le graphe d'évaluation du gradient en x est passé par le paramètre G.
2. $x \in \mathbb{R}^n$, point courant de la boucle externe (appelant de cette procédure).
3. $\nabla f \in \mathbb{R}^n$, le gradient de la fonction objectif en x.
4. $G_{\nabla f(x)}$, le graphe d'évaluation de $\nabla f(x)$.
5. $\Delta \in \mathbb{R}$, le rayon de la région de confiance.

Sortie : $d \in \mathbb{R}^n$, la solution optimale du programme $\min_d q(d)$.

Code :

$p \leftarrow -\nabla f^T$
$Q_p \leftarrow \mathtt{h}(x, p, G_{\nabla f(x)})$
$d \leftarrow 0$
$\nabla q \leftarrow \nabla f$
$\epsilon \leftarrow 10^{-12}$
BOUCLE
  SI $\|p\|^2 < \epsilon^2$ **ALORS**
    Rompre la boucle : la solution ne sera plus significativement modifiée.
  FIN SI
  $\alpha \leftarrow \frac{-p^T d + \sqrt{(p^T d)^2 - p^T p (d^T d - \Delta^2)}}{p^T p}$
  SI $p^t Q_p < \epsilon \|p\|^2$ **ALORS**
    $d \leftarrow d + \alpha p$
    Rompre la boucle : on détecte une courbure négative, aussi on utilise la solution à la limite de la région de confiance.
  FIN SI
  $\theta \leftarrow -\frac{\nabla q p}{p^T Q_p}$
  SI $\theta > \alpha$ **ALORS**
    $d \leftarrow d + \alpha p$
    Rompre la boucle : le pas de gradient conjugué sort de la région de confiance, on le bloque à la frontière.
  FIN SI
  $d \leftarrow d + \theta p$
  $\nabla q \leftarrow \mathtt{h}(x, d, G_{\nabla f(x)})^T + \nabla f$
  SI $\|\nabla q\| < \epsilon$ **ALORS**
    Rompre la boucle : on a atteint la précision désirée sur la solution.
  FIN SI
  $\beta \leftarrow \frac{\nabla q Q_p}{p^T Q_p}$
  $p \leftarrow -\nabla q^T + \beta p$
  $Q_p \leftarrow \mathtt{h}(x, p, G_{\nabla f(x)})$
FIN BOUCLE

Figure 3.4: Procédure `CG_RC` : algorithme du gradient conjugué pour solutionner le programme quadratique $\min_{\|d\| \leq \Delta} q(d) = \frac{1}{2} d^T \nabla^2 f(x) d + \nabla f(x) d$.

Problème		Code «maison»		`cgtrust`	
		$\|\nabla f(x)\|$	Nb. itér.	$\|\nabla f(x)\|$	Nb. itér.
1	rose	$1{,}65 \times 10^{-13}$	27	$2{,}75 \times 10^{-14}$	26
2	froth	$6{,}48 \times 10^{-13}$	10	$5{,}69 \times 10^{-14}$	11
3	badscp	$4{,}63 \times 10^{-11}$	200*	$6{,}42 \times 10^{-12}$	1000*
4	badscb	0	100	0	39
5	beale	$8{,}00 \times 10^{-15}$	9	$6{,}84 \times 10^{-13}$	8
6	jensam	$1{,}90 \times 10^{-14}$	11	$1{,}05 \times 10^{-14}$	10
8	bard	$1{,}83 \times 10^{-13}$	10	$6{,}70 \times 10^{-14}$	14
9	gauss	$6{,}55 \times 10^{-13}$	3	$2{,}19 \times 10^{-14}$	3
10	meyer	$1{,}31 \times 10^{+05}$	500*	$6{,}31 \times 10^{-5}$	775*
12	box	$1{,}90 \times 10^{-13}$	23	0	19
13	sing	$8{,}44 \times 10^{-13}$	30	$3{,}26 \times 10^{-13}$	29
14	wood	$2{,}48 \times 10^{-13}$	53	$4{,}89 \times 10^{-13}$	52
15	kowosb	$7{,}37 \times 10^{-13}$	14	$1{,}74 \times 10^{-13}$	15
16	bd	$8{,}14 \times 10^{-12}$	50*	$7{,}55 \times 10^{-11}$	50*
17	osb1	0,00219	100*	$1{,}12 \times 10^{-13}$	84
18	biggs	0,000186	100*	$7{,}24 \times 10^{-14}$	25
21	rosex	$7{,}02 \times 10^{-13}$	27	$3{,}01 \times 10^{-14}$	26
22	singx	$8{,}44 \times 10^{-13}$	31	$3{,}26 \times 10^{-13}$	29
23	pen1	$2{,}52 \times 10^{-16}$	48	$4{,}38 \times 10^{-15}$	53
24	pen2	$7{,}58 \times 10^{-13}$	154	$9{,}08 \times 10^{-14}$	193
25	vardim	$5{,}29 \times 10^{-14}$	13	$1{,}51 \times 10^{-14}$	11
26	trig	$8{,}56 \times 10^{-13}$	15	$2{,}28 \times 10^{-13}$	12
28	bv	$9{,}26 \times 10^{-13}$	5	$9{,}37 \times 10^{-16}$	4
29	ie	$4{,}26 \times 10^{-13}$	4	$6{,}51 \times 10^{-13}$	7
30	trid	$4{,}71 \times 10^{-13}$	7	$2{,}28 \times 10^{-13}$	13
31	band	$2{,}88 \times 10^{-13}$	9	$6{,}08 \times 10^{-13}$	14
32	lin	$2{,}02 \times 10^{-13}$	5	$9{,}53 \times 10^{-16}$	3
33	lin1	$3{,}48 \times 10^{-12}$	100*	$5{,}79 \times 10^{-13}$	2
34	lin0	$9{,}48 \times 10^{-13}$	37	$2{,}55 \times 10^{-13}$	2

Tableau 3.1: Comparaison des résultats entre le code d'optimisation par régions de confiance de la figure 3.3 et la procédure `cgtrust`. La variante du code «maison» comparable à `cgtrust` est celle où les hessiens directionnels sont calculés à l'aide de différences finies numériques.

`SCIAD`(x) Construit un graphe d'évaluation d'un seul sommet, lequel sommet représente une variable indépendante associée à la valeur x. En passant un tel objet au sous-programme d'évaluation d'une fonction, les opérateurs surchargés réalisent la construction du graphe d'évaluation de la fonction, lequel est renvoyé au terme du sous-programme.

`VALEUR`(G) Renvoie la valeur numérique rattachée au noeud sans successeur du graphe d'évaluation G.

`DFA`($G_{f(x)}, d$) Évalue la quantité $\Delta_d f(x)$ en utilisant sur le graphe $G_{f(x)}$ le visiteur de calcul des différences finies automatiques.

3.4 Expérimentations numériques

Toutes les expériences ont été menées sur une machine de modèle Sun Fire V880 dotée de six processeurs UltraSPARC et de 4 Go de mémoire principale. Le protocole expérimental consiste simplement en l'exécution des six procédures d'optimisation (deux méthodes, trois approches de dérivation) sur la banque de problèmes de Moré, Garbow et Hillstrom [MGH81], portée à Scilab depuis un port précédent du FORTRAN à Matlab [GKL95]. La table 3.2 définit, pour chacun des problèmes de cette collection, leur dimensionnalité et leur taille. Pour chaque variante algorithmique et chaque problème, nous visons l'atteinte du critère d'arrêt avec une précision meilleure à $\epsilon = 10^{-12}$.

3.4.1 Convergence des procédures

Les tableaux 3.3 et 3.4 illustrent le nombre d'itérations de la boucle principale et la norme du gradient au point où chacune des variantes des deux procédures ont interrompu leur progression. Pour certains problèmes, on observe une situation de *panne*, où la procédure cesse de générer des éléments de la suite $\{x_k\}$ tels que la descente soit significative, alors

Problème		Dimension	Taille	Problème		Dimension	Taille
1	rose	2	2	18	biggs	6	13
2	froth	2	2	21	rosex	4	4
3	badscp	2	2	22	singx	4	4
4	badscb	2	3	23	pen1	4	5
5	beale	2	3	24	pen2	4	8
6	jensam	2	2	25	vardim	5	7
8	bard	3	15	26	trig	5	5
9	gauss	3	15	28	bv	7	7
10	meyer	3	16	29	ie	7	7
12	box	3	10	30	trid	7	7
13	sing	4	4	31	band	10	10
14	wood	4	6	32	lin	7	9
15	kowosb	4	11	33	lin1	7	10
16	bd	4	20	34	lin0	7	11
17	osb1	5	33				

Tableau 3.2: Description des problèmes de la collection UNCprobs [MGH81]. Chacun de ces problèmes consiste en la minimisation de la norme au carré d'une fonction $f(x) = \|g(x)\|^2$, où $g : \mathbb{R}^n \leftarrow \mathbb{R}^m$. Le paramètre n correspond à la dimention du problème; le paramètre m, à sa taille.

que le critère d'arrêt n'est pas encore satisfait. Pour pallier à cette éventualité, les procédures sont artificiellement interrompues lorsque dix pas d'une longueur inférieure à 0.01ϵ ont été effectués ou au bout d'un certain nombre limite d'itérations principales. Le nombre d'itérations est marqué d'une astérisque lorsqu'une panne est survenue.

3.4.2 Performance des procédures

Nous avons noté au chapitre 2 que la présente implantation de la construction des graphes d'évaluation dans l'environnement Scilab implique une lourdeur logicielle significative. De ce fait, le temps d'exécution absolu des variantes de nos algorithmes qui calculent $\nabla^2 f(x)d$ par un visiteur du graphe d'évaluation de $\nabla f(x)$ est biaisé. Nous avons donc instrumenté notre code pour baser plutôt nos comparaisons sur le *coût opérationnel* de chaque variante, tel que nous l'avons introduit au chapitre 1.

Il apparaît pertinent d'exprimer ces résultats sous deux angles. Dans un premier temps, nous présentons le coût opérationnel total des variantes exécutées sans panne sur chaque problème. Ces résultats donnent une idée du coût réel de l'emploi des diverses variantes pour résoudre complètement des cas précis. Ils sont explicités dans les tableaux 3.5 et 3.6.

Dans un deuxième temps, nous considérons une comparaison où on exécute un nombre égal d'itérations principales pour chacune des variantes. Ceci nous permet d'estimer le coût de l'évaluation des dérivées secondes sans égard à la convergence. Ces résultats sont établis dans le cadre des tableaux 3.7 et 3.8.

3.4.3 Analyse des résultats

Mentionnons d'emblée que, malgré que la méthode de régions de confiance apparaisse un peu moins robuste que la méthode de Newton tronquée, les résultats qu'on obtient sont très semblable pour les deux approches algorithmiques d'optimisation. Par exemple, ce qui caractérise la variante de Newton-Raphson avec calcul des dérivées secondes par différences finies numériques caractérise la variante correspondante de l'algorithme de régions de confiance. Nous ne distinguerons donc que les variantes algorithmiques utilisant une méthode de dérivation différente. Ainsi, le sigle DFN désignera ces variantes où $\nabla^2 f(x)d$ est évaluée par différences finies numériques; DFA, les variantes où ces dérivées sont évaluées par différences finies automatiques; DA, les variantes où ces dérivées sont évaluées par différentiation automatique en préordre.

On note premièrement une certaine amélioration de la propriété de convergence des variantes DFA et DA des algorithmes. En effet, avec la méthode de Newton tronquée, sur cinq problèmes où DFN échoue, trois sont solutionnés à la précision requise par DFA et DA; avec la méthode de régions de confiance, DFN tombe en panne sur six problèmes, pour l'un desquels DFA et DA atteignent une solution à la précision requise. Pour un seul problème, DFN arrive à une solution à la précision voulue, alors que DFA et DA

tombent en panne.[2] Du reste, DFA et DA sont à toute fin pratique équivalentes quant à cette propriété.

En ce qui concerne le coût opérationnel des diverses variantes, la tendance est moins nette. Sur 58 couples méthode-problème considérés sous les trois variantes, DFA est plus efficace que DFN en 13 instances et coûte au moins le double de DFN en 17 instances. Par ailleurs, la variante DA apparaît plus efficace que DFN pour 31 couples méthode-problème, alors qu'elle exécute plus du double des opérations pour six couples. On observe que plusieurs des couples problème-méthode où DFA et DA sont plus économiques que DFN concernent les problèmes de plus petite échelle de l'ensemble (fonction objectif simple et faible dimensionnalité). L'opposé est aussi vrai : les problèmes où DFA et DA sont le plus coûteuses comparativement à DFN sont ceux de plus grande échelle (fonction objectif complexe et grand nombre de variables). Finalement, on observe que pour presque tous les couples problème-méthode, DA performe beaucoup plus efficacement que DFA.

Deux facteurs doivent être considérés pour expliquer ces résultats. Premièrement, le fait que DA apparaisse plus économique que DFA n'est pas surprenant. Le processus de calcul des différences finies automatiques constitue un parcours en postordre du graphe d'évaluation du gradient, dans le cadre duquel un vecteur est initialement rattaché aux sommets associés aux variables indépendantes. Cela correspond à l'initialisation de la dérivation en postordre avec une matrice initiale réduite à un seul vecteur. Les formules de propagation des quantités intermédiaires de ces deux procédés se ressemblent, par ailleurs. Tenant de [Gri89] que le mode en préordre de la différentiation automatique est intrinsèquement plus économique en temps de calcul que le mode en postordre, on peut s'attendre à observer la même chose pour ce qui a trait au calcul des différences finies automatiques.

[2]On constate toutefois que cette panne survient en un point très rapproché de l'ensemble $\{x \in \mathbb{R}^n : \|\nabla f(x)\| < 10^{-12}\}$.

Deuxièmement, les mesures du nombre d'opérations effectivement réalisées par les variantes DFA et DA sont faussées par un détail de l'implantation de SCIAD. Soit le graphe d'évaluation d'une expression dont le résultat consiste en une matrice $M_{m\times n}$. L'opération qui consiste en l'ajout d'une ligne correspondant au vecteur $L_{1\times n}$ à cette matrice[3] est implantée dans SCIAD à l'aide de produits et d'addition. En effet, reprenant la terminologie et la notation de Scilab, la matrice résultant de la *concaténation verticale* de M et L peut être exprimée comme une expression algébrique :

$$[M;L] = \begin{bmatrix} I_{m\times m} \\ 0_{1\times m} \end{bmatrix} M + \begin{bmatrix} 0_{m\times 1} \\ 1 \end{bmatrix} L \tag{3.5}$$

Incidemment, pour chacun des problèmes expérimentaux, le code d'évaluation de la fonction objectif construit dans ce but un certain vecteur de taille m élément par élément. Lorsqu'on passe comme paramètre à ce code un germe de graphe d'évaluation SCIAD, le vecteur est construit par $m-1$ applications de l'équation 3.5. Ensuite, lorsqu'on calcule plutôt le gradient de la fonction, le code d'évaluation construit une matrice de taille $m \times n$ (où n constitue la dimensionnalité du problème) en plus du vecteur mentionné. Encore une fois, l'utilisation de SCIAD avec ce code réalise la construction par un procédé arithmétique analogue à celui de l'équation 3.5. On s'attend donc à ce que DFA et DA, qui construisent et utilisent le graphe d'évaluation du gradient, exécutent de nombreux produits et additions que la variante DFN n'exécute pas. On s'attend aussi à ce que ce nombre d'opérations «inattendues» augmente avec l'échelle des problèmes considérés. Cela corrobore nos observations des coûts relatifs de DFA et DA en regard de DFN, pour la résolution complète des programmes comme pour l'exécution d'un nombre égal d'itérations pour chaque variante.

Il se dégage donc de ces résultats trois faits :

[3]En langage Scilab, cette opération peut être réalisée avec l'une des instructions `M = [M;L]` ou `M($+1,:) = L`.

1. L'usage de méthodes automatiques de calcul des dérivées secondes directionnelles n'améliore que marginalement la robustesse d'une procédure d'optimisation.

2. Pour les problèmes de plus grande échelle, une implantation peu sophistiquée de certaines opérations de la librairie de différentiation automatique peut diminuer significativement la performance d'une procédure qui utilise une technique automatique de calcul des dérivées secondes.

3. La différentiation automatique en préordre est plus efficace que les différences finies automatiques pour l'évaluation de dérivées directionnelles de second ordre.

On conçoit de ces faits que si les différences finies automatiques sont à rejeter comme technique de calcul des hessiens directionnels, la différentiation automatique en préordre demeure attrayante. En effet, nos expériences montrent qu'en plus d'augmenter quelque peu la robustesse de la procédure d'optimisation, cette technique permet des économies substantielles de temps de calcul par rapport à la technique des différences finies triviales. Il faut cependant que la librairie de différentiation automatique employée n'emprunte pas les mêmes raccourcis d'implantation que SCIAD, attendu que cela mène à des coûts supplémentaires qui croissent avec la taille du problème.

Cette recommandation est valable dans la mesure où le problème d'optimisation considéré doit être résolu avec une précision très élevée. Des méthodes d'optimisation sans contrainte telles que L-BFGS [MN00, Noc], qui n'utilisent qu'une information de second ordre relativement pauvre, sont reconnues pour leur efficacité dans le cadre de l'atteinte d'une solution à une précision moins élevée que celle considérée dans ce chapitre. Il serait intéressant de comparer la performance d'une telle procédure à celle des méthodes présentées ici, pour lesquelles les hessiens directionnels seraient obtenus par différentiation automatique en préordre. On pourrait utiliser SCIAD pour ce faire, attendu que les inefficacités mentionnées soient préalablement solutionnées.

Problème		DF numériques		DF automatiques		Différentiation auto.	
		$\|\nabla f(x)\|$	Nb. itér.	$\|\nabla f(x)\|$	Nb. itér.	$\|\nabla f(x)\|$	Nb. itér.
1	rose	$3{,}48 \times 10^{-13}$	23	$1{,}38 \times 10^{-14}$	22	$1{,}38 \times 10^{-14}$	22
2	froth	$2{,}84 \times 10^{-13}$	8	$5{,}69 \times 10^{-14}$	7	$5{,}69 \times 10^{-14}$	7
3	badscp	$8{,}87 \times 10^{-12}$	174*	$8{,}55 \times 10^{-13}$	161	$4{,}44 \times 10^{-16}$	162
4	badscb	$5{,}90 \times 10^{+07}$	100*	0,00	5	0,00	5
5	beale	$6{,}22 \times 10^{-15}$	11	0,00	10	$6{,}22 \times 10^{-15}$	10
6	jensam	$3{,}14 \times 10^{-15}$	9	$3{,}14 \times 10^{-15}$	8	$3{,}14 \times 10^{-15}$	8
8	bard	$6{,}89 \times 10^{-14}$	11	$7{,}75 \times 10^{-16}$	8	$7{,}77 \times 10^{-16}$	8
9	gauss	$3{,}21 \times 10^{-16}$	4	$4{,}40 \times 10^{-16}$	3	$3{,}28 \times 10^{-17}$	3
10	meyer	$5{,}76 \times 10^{+05}$	21*	$2{,}24 \times 10^{-05}$	500*	0,000122	215*
12	box	$6{,}78 \times 10^{-13}$	33	$2{,}26 \times 10^{-16}$	121	$2{,}26 \times 10^{-16}$	107
13	sing	$5{,}47 \times 10^{-13}$	34	$7{,}59 \times 10^{-13}$	28	$7{,}59 \times 10^{-13}$	28
14	wood	$2{,}02 \times 10^{-13}$	56	0,00	159	$4{,}50 \times 10^{-14}$	159
15	kowosb	$1{,}55 \times 10^{-13}$	11	$4{,}62 \times 10^{-16}$	10	$4{,}26 \times 10^{-16}$	10
16	bd	$8{,}14 \times 10^{-12}$	23*	$8{,}16 \times 10^{-12}$	19*	$8{,}15 \times 10^{-12}$	19*
17	osb1	0,000317	100*	$4{,}78 \times 10^{-14}$	61	$1{,}57 \times 10^{-13}$	59
18	biggs	$1{,}20 \times 10^{-13}$	48	$2{,}52 \times 10^{-15}$	39	$1{,}72 \times 10^{-15}$	39
21	rosex	$1{,}41 \times 10^{-13}$	24	$1{,}95 \times 10^{-14}$	22	$1{,}95 \times 10^{-14}$	22
22	singx	$5{,}47 \times 10^{-13}$	34	$7{,}59 \times 10^{-13}$	28	$7{,}59 \times 10^{-13}$	28
23	pen1	$3{,}30 \times 10^{-15}$	36	$3{,}36 \times 10^{-15}$	35	$3{,}36 \times 10^{-15}$	35
24	pen2	$5{,}29 \times 10^{-13}$	124	$8{,}50 \times 10^{-15}$	115	$3{,}98 \times 10^{-14}$	114
25	vardim	$7{,}63 \times 10^{-13}$	13	$4{,}86 \times 10^{-14}$	11	$5{,}70 \times 10^{-14}$	11
26	trig	$6{,}77 \times 10^{-14}$	14	$2{,}12 \times 10^{-13}$	12	$2{,}08 \times 10^{-13}$	12
28	bv	$4{,}29 \times 10^{-16}$	5	$2{,}70 \times 10^{-16}$	4	$1{,}36 \times 10^{-16}$	4
29	ie	$8{,}49 \times 10^{-14}$	4	$2{,}01 \times 10^{-16}$	4	$1{,}71 \times 10^{-16}$	4
30	trid	$1{,}18 \times 10^{-14}$	8	$8{,}85 \times 10^{-15}$	7	$1{,}11 \times 10^{-14}$	7
31	band	$1{,}16 \times 10^{-14}$	9	$1{,}33 \times 10^{-14}$	8	$6{,}51 \times 10^{-13}$	23
32	lin	$1{,}10 \times 10^{-13}$	4	$4{,}81 \times 10^{-15}$	1	$1{,}57 \times 10^{-16}$	1
33	lin1	$5{,}79 \times 10^{-13}$	5	$7{,}51 \times 10^{-12}$	11*	$7{,}51 \times 10^{-12}$	11*
34	lin0	$9{,}48 \times 10^{-13}$	4	$2{,}55 \times 10^{-13}$	3	$2{,}55 \times 10^{-13}$	3

Tableau 3.3: Résultats de convergence pour les expériences sur les variantes de la méthode de Newton tronquée.

Problème		DF numériques		DF automatiques		Différentiation auto.	
		$\|\nabla f(x)\|$	Nb. itér.	$\|\nabla f(x)\|$	Nb. itér.	$\|\nabla f(x)\|$	Nb. itér.
1	rose	$1{,}65 \times 10^{-13}$	27	$4{,}97 \times 10^{-14}$	26	$4{,}97 \times 10^{-14}$	26
2	froth	$6{,}48 \times 10^{-13}$	10	$8{,}57 \times 10^{-14}$	8	$5{,}71 \times 10^{-14}$	8
3	badscp	$4{,}63 \times 10^{-11}$	200*	$6{,}89 \times 10^{-12}$	200*	$2{,}92 \times 10^{-11}$	200*
4	badscb	0,00	100	0,00	37	0,00	37
5	beale	$8{,}00 \times 10^{-15}$	9	$7{,}02 \times 10^{-15}$	8	$5{,}69 \times 10^{-15}$	8
6	jensam	$1{,}90 \times 10^{-14}$	11	$1{,}27 \times 10^{-14}$	10	$3{,}14 \times 10^{-15}$	10
8	bard	$1{,}83 \times 10^{-13}$	10	$6{,}36 \times 10^{-16}$	14	$4{,}51 \times 10^{-16}$	14
9	gauss	$6{,}55 \times 10^{-13}$	3	$2{,}52 \times 10^{-16}$	3	$1{,}66 \times 10^{-16}$	3
10	meyer	$1{,}31 \times 10^{+05}$	500*	0,000325	500*	0,00129	500*
12	box	$1{,}90 \times 10^{-13}$	23	$1{,}44 \times 10^{-14}$	19	$1{,}41 \times 10^{-14}$	19
13	sing	$8{,}44 \times 10^{-13}$	30	$7{,}08 \times 10^{-13}$	28	$7{,}08 \times 10^{-13}$	28
14	wood	$2{,}48 \times 10^{-13}$	53	$1{,}29 \times 10^{-13}$	48	$4{,}27 \times 10^{-14}$	48
15	kowosb	$7{,}37 \times 10^{-13}$	14	$4{,}58 \times 10^{-13}$	14	$4{,}80 \times 10^{-13}$	14
16	bd	$8{,}14 \times 10^{-12}$	50*	$8{,}14 \times 10^{-12}$	50*	$8{,}14 \times 10^{-12}$	50*
17	osb1	0,00219	100*	$8{,}41 \times 10^{-14}$	50	$3{,}02 \times 10^{-13}$	47
18	biggs	0,000186	100*	0,0438	100*	0,0436	100*
21	rosex	$7{,}02 \times 10^{-13}$	27	$7{,}02 \times 10^{-14}$	26	$7{,}02 \times 10^{-14}$	26
22	singx	$8{,}44 \times 10^{-13}$	30	$7{,}08 \times 10^{-13}$	28	$7{,}08 \times 10^{-13}$	28
23	pen1	$2{,}52 \times 10^{-16}$	48	$8{,}62 \times 10^{-13}$	47	$8{,}62 \times 10^{-13}$	47
24	pen2	$7{,}58 \times 10^{-13}$	154	$3{,}14 \times 10^{-13}$	141	$2{,}78 \times 10^{-13}$	142
25	vardim	$5{,}29 \times 10^{-14}$	13	$1{,}46 \times 10^{-13}$	11	$4{,}94 \times 10^{-13}$	11
26	trig	$8{,}56 \times 10^{-13}$	15	$9{,}14 \times 10^{-14}$	13	$8{,}87 \times 10^{-14}$	13
28	bv	$9{,}26 \times 10^{-13}$	5	$5{,}79 \times 10^{-13}$	4	$5{,}79 \times 10^{-13}$	4
29	ie	$4{,}26 \times 10^{-13}$	4	$2{,}94 \times 10^{-13}$	4	$2{,}94 \times 10^{-13}$	4
30	trid	$4{,}71 \times 10^{-13}$	7	$1{,}33 \times 10^{-14}$	6	$1{,}03 \times 10^{-14}$	6
31	band	$2{,}88 \times 10^{-13}$	9	$4{,}22 \times 10^{-13}$	8	$4{,}54 \times 10^{-13}$	23
32	lin	$2{,}02 \times 10^{-13}$	5	$5{,}41 \times 10^{-16}$	2	$1{,}34 \times 10^{-15}$	2
33	lin1	$3{,}48 \times 10^{-12}$	100*	$1{,}45 \times 10^{-12}$	100*	$4{,}62 \times 10^{-12}$	100*
34	lin0	$9{,}48 \times 10^{-13}$	37	$2{,}55 \times 10^{-13}$	2	$2{,}55 \times 10^{-13}$	49

Tableau 3.4: Résultats de convergence pour les expériences sur les variantes de l'algorithme de régions de confiance.

Problème		Coût absolu			Coût relatif		
		DF num. [A]	DF auto. [B]	Diff. auto. [C]	B/A	C/A	C/B
1	rose	383411	269039	214515	0,70	0,56	0,80
2	froth	218298	116695	92727	0,53	0,42	0,79
3	badscp	-	5779268	3779893	-	-	0,65
4	badscb	-	63405	45569	-	-	0,72
5	beale	280438	279666	189077	1,0	0,67	0,68
6	jensam	538322	269970	176474	0,50	0,33	0,65
8	bard	4179503	14809582	6142534	3,5	1,5	0,41
9	gauss	4667876	5019517	2462601	1,1	0,53	0,49
12	box	14286282	105747262	42874928	7,4	3,0	0,41
13	sing	3347830	5282655	2388728	1,6	0,71	0,45
14	wood	6253437	33897495	14936376	5,4	2,4	0,44
15	kowosb	6158139	23788426	8647239	3,9	1,4	0,36
17	osb1	-	$> 2^{31}$	1071251229	-	-	$< 0,50$
18	biggs	101215605	196554576	70382015	1,9	0,70	0,36
21	rosex	1278174	1331082	690726	1,0	0,54	0,52
22	singx	3148430	5969514	2526766	1,9	0,80	0,42
23	pen1	3308153	2899098	1505853	0,88	0,46	0,52
24	pen2	67459391	99843757	31948139	1,5	0,47	0,32
25	vardim	1450605	1500277	788753	1,0	0,54	0,53
26	trig	6965648	15489853	5199367	2,2	0,75	0,34
28	bv	1739453	6023513	1946287	3,5	1,1	0,32
29	ie	1561169	10488915	3812679	6,7	2,4	0,36
30	trid	2001805	9210847	2836417	4,6	1,4	0,31
31	band	7416555	90560085	108818273	12	15	1,2
32	lin	660751	127061	94695	0,19	0,14	0,75
34	lin0	428211	618395	412715	1,4	0,96	0,67
Moyenne					2,8	1,6	0,54

Tableau 3.5: Comparaison des coûts de la résolution complète pour les variantes de la méthode de Newton tronquée.

Problème		Coût absolu			Coût relatif		
		DF num. [A]	DF auto. [B]	Diff. auto. [C]	B/A	C/A	C/B
1	rose	677017	493829	273818	0,73	0,40	0,55
2	froth	312240	179830	120282	0,58	0,39	0,67
4	badscb	1023085	705516	436941	0,69	0,43	0,62
5	beale	354232	314699	182310	0,89	0,51	0,58
6	jensam	697600	335151	206421	0,48	0,30	0,62
8	bard	4637083	38670503	12696031	8,3	2,7	0,33
9	gauss	3931079	6945285	3111447	1,8	0,79	0,45
12	box	14002886	24652962	8257522	1,8	0,59	0,33
13	sing	3727766	4970304	1992016	1,3	0,53	0,40
14	wood	5597098	17985110	5626711	3,2	1,0	0,31
15	kowosb	6563394	35675287	11915955	5,4	1,8	0,33
17	osb1	-	$> 2^{31}$	961294221	-	-	$< 0,45$
21	rosex	2194974	2464251	844216	1,1	0,38	0,34
22	singx	3506278	5611173	2113069	1,6	0,60	0,38
23	pen1	4844136	4629743	2157995	0,96	0,45	0,47
24	pen2	63134810	120539921	39641287	1,9	0,63	0,33
25	vardim	2394226	2709465	1583652	1,1	0,66	0,58
26	trig	6829082	16419386	5136637	2,4	0,75	0,31
28	bv	2022609	6030932	1953706	3,0	0,97	0,32
29	ie	3534268	15327202	5312848	4,3	1,5	0,35
30	trid	2568543	9159318	2784888	3,6	1,1	0,30
31	band	11117050	255756043	148270818	23	13	0,58
32	lin	363289	263257	182342	0,72	0,50	0,69
34	lin0	4545127	484806	6538916	0,11	1,4	13
Moyenne					3,0	1,4	0,99

Tableau 3.6: Comparaison des coûts de la résolution complète pour les variantes de la méthode de régions de confiance.

Problème		Nb. itér.	Coût absolu			Coût relatif		
			DF num. [A]	DF auto. [B]	Diff. auto. [C]	B/A	C/A	C/B
1	rose	22	354365	269039	214515	0,76	0,61	0,80
2	froth	7	181574	116695	92727	0,64	0,51	0,79
3	badscp	161	6866056	5779268	3742200	0,84	0,55	0,65
4	badscb	5	44258	63405	45569	1,4	1,0	0,72
5	beale	10	238279	279666	189077	1,2	0,79	0,68
6	jensam	8	431462	269970	176474	0,63	0,41	0,65
8	bard	8	2698766	14809582	6142534	5,5	2,3	0,41
9	gauss	3	3235052	5019517	2462601	1,6	0,76	0,49
10	meyer	21	107225993	107225561	33963958	1,0	0,32	0,32
12	box	33	14286282	14285535	12542615	1,0	0,88	0,88
13	sing	28	2638466	5282655	2388728	2,0	0,91	0,45
14	wood	56	6253437	6252613	5113257	1,0	0,82	0,82
15	kowosb	10	5356108	23788426	8647239	4,4	1,6	0,36
16	bd	19	19687114	223692270	238298201	11	12	1,1
17	osb1	59	203470899	$> 2^{31}$	1071251229	> 11	5,3	$< 0,50$
18	biggs	39	67365441	196554576	70382015	2,9	1,0	0,36
21	rosex	22	1078398	1331082	690726	1,2	0,64	0,52
22	singx	28	2483334	5969514	2526766	2,4	1,0	0,42
23	pen1	36	3307329	2899098	1505853	0,88	0,46	0,52
24	pen2	114	61959463	61959463	31948139	1,0	0,52	0,52
25	vardim	11	1036802	1500277	788753	1,4	0,76	0,53
26	trig	12	5664041	15489853	5199367	2,7	0,92	0,34
28	bv	4	1269250	6023513	1946287	4,7	1,5	0,32
29	ie	4	1561169	10488915	3812679	6,7	2,4	0,36
30	trid	7	1610230	9210847	2836417	5,7	1,8	0,31
31	band	8	6081115	90560085	20403182	15	3,4	0,23
32	lin	1	74870	127061	94695	1,7	1,3	0,75
33	lin1	5	488432	487377	599045	1,0	1,2	1,2
34	lin0	3	319614	618395	412715	1,9	1,3	0,67
Moyenne						3,2	1,6	0,57

Tableau 3.7: Comparaison des coûts d'un nombre égal d'itérations pour les variantes de la méthode de Newton tronquée.

Problème		Nb. itér.	Coût absolu			Coût relatif		
			DF num. [A]	DF auto. [B]	Diff. auto. [C]	B/A	C/A	C/B
1	rose	26	669613	493829	273818	0,74	0,41	0,55
2	froth	8	266734	179830	120282	0,67	0,45	0,67
3	badscp	200	6835603	5113624	2735896	0,75	0,40	0,54
4	badscb	37	356715	705516	436941	2,0	1,2	0,62
5	beale	8	338096	314699	182310	0,93	0,54	0,58
6	jensam	10	672970	335151	206421	0,50	0,31	0,62
8	bard	10	4637083	27997585	12695716	6,0	2,7	0,45
9	gauss	3	3931079	6945285	3111447	1,8	0,79	0,45
10	meyer	20	8450774	64614434	24522756	7,6	2,9	0,38
12	box	19	11566384	24652962	8257522	2,1	0,71	0,33
13	sing	28	3575290	4970304	1992016	1,4	0,56	0,40
14	wood	48	4902309	17985110	5626711	3,7	1,1	0,31
15	kowosb	14	6563394	35675287	11915955	5,4	1,8	0,33
16	bd	50	26086345	534640443	172331753	20	6,6	0,32
17	osb1	47	151352644	$> 2^{31}$	961294221	> 14	6,4	$< 0,45$
18	biggs	20	32185083	159422148	46582437	5,0	1,4	0,29
21	rosex	26	2173559	2464251	844216	1,1	0,39	0,34
22	singx	28	3361592	5611173	2113069	1,7	0,63	0,38
23	pen1	47	4821022	4629743	2157995	0,96	0,45	0,47
24	pen2	141	56954804	120539921	39499954	2,1	0,69	0,33
25	vardim	11	2298097	2709465	1583652	1,2	0,69	0,58
26	trig	13	5997276	16419386	5136637	2,7	0,86	0,31
28	bv	4	1968542	6030932	1953706	3,1	0,99	0,32
29	ie	4	3534268	15327202	5312848	4,3	1,5	0,35
30	trid	6	2468424	9159318	2784888	3,7	1,1	0,30
31	band	8	10994720	255756043	78332199	23	7,1	0,31
32	lin	2	146856	263257	182342	1,8	1,2	0,69
33	lin1	100	9938750	18348159	11748433	1,8	1,2	0,64
34	lin0	2	231053	484806	304445	2,1	1,3	0,63
Moyenne						4,3	1,6	0,45

Tableau 3.8: Comparaison des coûts d'un nombre égal d'itérations pour les variantes de l'algorithme de régions de confiance.

CONCLUSION ET PERSPECTIVES

En définitive, nous avons introduit les techniques de différentiation automatique comme des méthodes d'évaluation de la dérivée de fonctions $\mathbb{R}^n \to \mathbb{R}^m$ qui consiste en une transformation du graphe d'évaluation de ces fonctions. Nous avons réalisé l'implantation d'outils logiciels basé sur ces méthodes. L'architecture orientée objet modulaire de cet ensemble d'outils permet le développement d'extensions novatrices aux techniques de base décrites dans la littérature. De telles extensions peuvent être définies pour résoudre divers problèmes liés à la plate-forme d'implantation, ainsi qu'à l'usage des outils sous certaines conditions – notamment, la dérivation d'ordre supérieur.

Ce mécanisme d'extension peut aussi servir à mettre au point des logiciels de calcul différentiel s'attaquant à des problèmes qui dépassent le cadre de la dérivation. Nous avons en effet formulé un problème numérique qui survient dans les algorithmes d'optimisation sans contrainte de fonctions différentiables en termes d'erreurs d'annulation associées à l'évaluation de différences finies de la fonction objectif. Nous avons alors implanté un calculateur automatique de différences finies comme une extension parallèle des outils de différentiation automatique, lequel évite ces erreurs d'annulation; diverses expériences empiriques confirment la performance de cet outil pour la recherche d'une solution avec une précision élevée sur le critère d'arrêt de la procédure d'optimisation.

Finalement, nous avons comparé l'usage du calculateur automatique de différences finies aux techniques traditionnelles de différentiation automatique pour l'évaluation de dérivées

directionnelles de second ordre. Il ressort de nos expériences que la performance des différences finies automatiques est inférieure à la différentiation automatique en préordre; le processus de calcul des premières se compare plutôt, en effet, à celui exécuté dans le cadre de la différentiation automatique en postordre, laquelle est intrinsèquement moins performante que le mode en préordre.

Les perspectives de développement de nos outils sont intéressantes, attendu que les divers problèmes de performance rencontrés puissent être convenablement résolus. En ce sens, les travaux prévus en vu du développement de visiteurs qui réalisent les heuristiques évoquées au chapitre 1 (complément 1.2.3) ouvrent la possibilité de calculer des dérivées d'ordre supérieur à un coût acceptable. Cette ouverture serait mise à profit dans le cadre d'un algorithme d'optimisation dit *de Newton d'ordre supérieur* [Duson]. Alors que la méthode de Newton-Raphson traditionnelle emploie l'information des dérivées de premier et de second ordre, un tel algorithme calcule les pas successifs en utilisant des dérivées d'ordre supérieur. Il arrive ainsi à contourner certaines situations de divergence rencontrées avec des fonctions non continues en ce minimum local vers lequel les itérations se dirigent.

Dans un contexte plus large, les techniques de différentiation automatique gagnent en popularité dans la communauté de l'optimisation. Des langages de modélisation mathématique populaires tels que AMPL [FGK90] et COSY [Ber02] incluent dans leur compilateur des outils de différentiation automatique efficaces, rendant les techniques et leurs bénéfices plus accessibles aux développeurs de solveurs. La différentiation automatique éclot aujourd'hui de son statut expérimental et s'établit comme un outil de base de l'optimisation. Cette progression constitue un cadre stimulant pour la suite de nos recherches.

BIBLIOGRAPHIE

[BCH+98] Christian H. Bischof, Alan Carle, Paul D. Hovland, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0 user's guide (Revision D). Technical report, Mathematics and Computer Science Division Technical Memorandum no. 192 and Center for Research on Parallel Computation Technical Report CRPC-95516-S, 1998.

[Ber02] Martin Berz. COSY INFINITY Version 8.1 User's Guid and Reference Manual. Department of Physics and Astronomy MSUHEP-20704, Michigan State University, 2002.

[DS83] John E. Dennis and Robert B. Schnabel. *Numerical methods for nonlinear equations and unconstrained optimization.* Prentice Hall, 1983.

[Duson] Jean-Pierre Dussault. High order newton-penalty algorithms. *Journal of Computational and Applied Mathematics*, soumis pour publication.

[FGK90] Robert Fourer, David M. Gay, and Brian W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36:519–554, 1990.

[GJM+99] Andreas Griewank, David Juedes, Hristo Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of Algorithms written in C/C++. Technical report, Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[GKL95] Chaya Gurwitz, Livia Klein, and Madhu Lamba. UNCPROBS - a MATLAB library of test functions for unconstrained optimization. `ftp://ftp.mathworks.com/pub/contrib/v4/optim/uncprobs`, 1995.

[Gri89] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[Gri92] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1), 1992.

[JS02] Xavier Jonsson and Serge Steer. Diffcode. `ftp://ftp.inria.fr/INRIA/Scilab/contrib/DIFFCODE/`, août 2002.

[Kel99] Tim Kelley. *Iterative Methods for Optimization.* SIAM, 1999.

[MGH81] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Algorithm 566: FORTRAN subroutines for testing unconstrained optimization software [C5 [E4]]. *ACM Transactions on Mathematical Software*, 7(1):136–140, March 1981. [ftp://ftp.mathworks.com/pub/contrib/v4/optim/uncprobs.tar].

[MN00] J. L. Morales and J. Nocedal. Automatic preconditioning by limited memory quasi-newton updating. *SIAM Journal of Optimization*, 10:1079–1096, 2000.

[Noc] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematical Computing*, 35:773–782.

[Sci03] Scilab group. *Scilab 2.7.* Institut National de Recherche en Informatique et Automatique - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 - LE CHESNAY Cedex - FRANCE, email : Scilab@inria.fr, 2003. `http://scilabsoft.inria.fr/`.