RAIRO-Oper. Res. 42 (2008) 141–155
DOI: 10.1051/ro :2008007

RAIRO Operations Research
www.rairo-ro.org

LA DIFFÉRENTIATION AUTOMATIQUE ET SON UTILISATION EN OPTIMISATION *

Jean-Pierre Dussault [1]

Abstract. In this work, we present an introduction to automatic differentiation, its use in optimization software, and some new potential usages. We focus on the potential of this technique in optimization. We do not dive deeply in the intricacies of automatic differentiation, but put forward its key ideas. We sketch a survey, as of today, of automatic differentiation software, but warn the reader that the situation with respect to software evolves rapidly. In the last part of the paper, we present some potential future usage of automatic differentiation, assuming an ideal tool is available, which will become true in some unspecified future.

Résumé. Dans ce travail, nous présentons une introduction sur la différentiation automatique, son utilisation dans les logiciels usuels d'optimisation, et enfin des perspectives de l'apport que cette technique promet. L'accent est mis plutôt sur le potentiel de cette technique en optimisation. Ainsi, la présentation se veut simple ; nous mentionnons tout de même quelques aspects plus avancés. Les logiciels évoluant assez rapidement, ce document est daté, et il est certain que plusieurs aspects de la présentation ayant trait aux logiciels actuels seront bientôt caduques. Cependant, dans la dernière partie du document dédiée aux perspectives, nous définissons les attentes d'un outil "idéal" sans nous préoccuper de leur disponibilité aujourd'hui.

Mots Clés. Différentiation automatique, algorithmes numériques d'optimisation.

Classification Mathématique. 90C.

Received May 31, 2007 ; accepted November 28, 2007.

* *Recherche partiellement subventionnée par un octroi CRSNG # OGP0005491.*

[1] Département d'informatique, Faculté des Sciences, Université de Sherbrooke, Sherbrooke, Québec J1K 2R1, Canada ; Jean-Pierre.Dussault@USherbrooke.ca

Article published by EDP Sciences

1. Introduction

La technique aujourd'hui nommée *différentiation automatique* et notée DA par la suite débute à la fin des années cinquante (pour une bibliographie et un manuel sur le sujet, voir [6]). L'idée est séduisante : à partir du programme calculant une certaine fonction, produire automatiquement un programme calculant les dérivées de la fonction en question.

Depuis, des logiciels de manipulation symbolique, tels Macsyma, Maple, Mathematica ont été développés ; ces logiciels permettent la *différentiation symbolique* de fonctions. Quelle est donc la différence entre ces deux outils informatiques, les différentiations automatique et symbolique ? En résumé, la différentiation symbolique a pour objectif de fournir une *expression* de la dérivée en question à partir de l'expression de la fonction alors que la DA a l'objectif (plus modeste ?) de fournir un programme calculant la dérivée à partir du programme calculant la fonction.

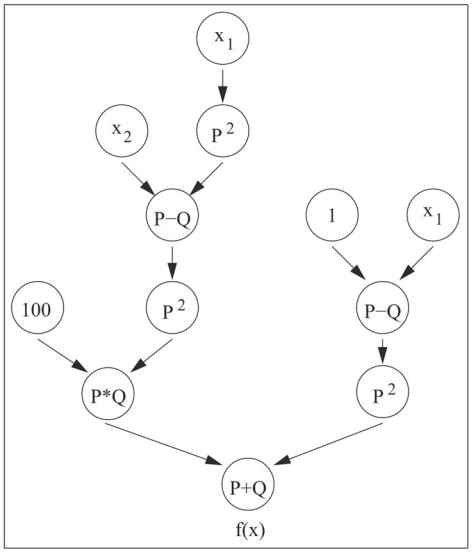
Un aspect important qui distingue la DA de la différentiation symbolique réside dans l'aspect informatique, le coût de calcul de la fonction et ses dérivées. En fait, la DA constitue un traitement symbolique appliqué au texte, ou code, du programme calculant une fonction, duquel résulte un programme souvent efficace pour calculer ses dérivées. La caractérisation de l'efficacité des calculs des dérivées est volontairement vague : nous verrons que les outils les plus souples imposent un compromis d'efficacité, mais que d'autres outils imposent un compromis de souplesse tout en proposant un calcul des dérivées plus efficace.

Quiconque a déjà utilisé un outil de différentiation symbolique comprend que le résultat n'a pas pour objectif une utilisation de calcul (numérique). Similairement, le résultat d'un outil de DA est inutile si on veut obtenir une expression mathématique de la dérivée d'une fonction.

Notre présentation concerne donc la DA. Nous résumerons d'abord la technique, ainsi que ses principales variantes. Nous mettrons en évidence les compromis en termes de coûts de calcul que ces variantes imposent. Cette partie de la présentation est de nature informatique : nous présentons un *outil* informatique, et ses propriétés en termes de complexité de calcul.

Dans un second volet, nous présentons l'utilisation de l'outil à un niveau 0, pourrait-on dire. Comme chacun le sait, les problèmes d'optimisation différentiable utilisent les gradients et Hessiens des fonctions objectif, ainsi que des contraintes, et donc sont susceptibles de bénéficier directement de l'outil de différentiation automatique. Nous donnons des indications sur les disponibilités de ces outils au sein de logiciels d'optimisation. Cette section est liée aux outils disponibles au moment de la rédaction de la présentation.

La présentation jusqu'à ce point concerne des résultats connus, bien qu'introduits selon le point de vue de l'auteur. Déjà en 1989, Griewank attirait l'attention de la communauté de programmation mathématique sur les bénéfices d'utiliser la DA [4]. Plus récemment, Moré a présenté les aspects pratiques de l'utilisation de la DA au sein du serveur NEOS [7].



On dénote par P l'unique branche en haut d'un noeud, ou encore sa branche de gauche, celle de droite étant dénotée par Q. Les deux noeuds de plus que les 10 étapes proviennent des constantes 1 et 100. Il faut noter que les opérations possibles aux noeuds peuvent être limitées, par exemple en ne considérant que l'addition, la soustraction passant par un opérateur unaire de négation.

FIGURE 1. GAO pour la fonction de Rosenbrock.

Dans la dernière partie, nous abordons un projet plus personnel : explorer comment la DA, en se diffusant, pourra influencer le développement de nouveaux algorithmes d'optimisation. Cette partie anticipe sur le développement et la diffusion de nouveaux outils de DA.

2. La différentiation automatique

Considérons l'exemple d'une fonction objectif $f : \mathbb{R}^n \to \mathbb{R}$. D'un point de vue informatique, le calcul de la fonction pour un vecteur x donné s'effectue par étapes. Par exemple, la fameuse fonction de Rosenbrock définie pour $x = (x_1, x_2)^t$ par $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ peut se calculer ainsi :

(1) $y_1 = [1\ 0]x; \quad (= x_1)$
(2) $y_2 = [0\ 1]x; \quad (= x_2)$
(3) $y_3 = y_1^2$;
(4) $y_4 = y_2 - y_3$;
(5) $y_5 = y_4^2$;
(6) $y_6 = 100y_5$;
(7) $y_7 = [1\ 0]x$;
(8) $y_8 = 1 - y_7$;
(9) $y_9 = y_8^2$;
(10) $f = y_6 + y_9$.

On peut représenter tout ce calcul à l'aide d'un graphe acyclique orienté (GAO) tel qu'illustré à la Figure 1.

L'outil informatique connu sous le nom de *différentiation automatique* consiste à transformer cette chaîne de calculs, ou encore ce GAO en un autre GAO qui représente cette fois le calcul de $\nabla f(x)$. L'outil de base est la règle d'enchaînement,

$$\nabla_x(f(g(x))) = \left(\nabla_y f(y)\Big|_{y=g(x)}\right) \cdot \nabla_x g(x).$$

2.1. Mode direct de la DA

Dans son mode direct, par exemple, il suffit d'accompagner la propagation des valeurs y_i par la propagation de leurs gradients $\nabla_x y_i$:

y_i	Valeur de y_i	∇y_i	Valeur de ∇y_i
$y_1 = [1\ 0]x\,;$	x_1	$\nabla y_1 = [1\ 0]\,;$	$[1\ 0]$
$y_2 = [0\ 1]x\,;$	x_2	$\nabla y_2 = [0\ 1]\,;$	$[0\ 1]$
$y_3 = y_1^2\,;$	x_1^2	$\nabla y_3 = 2y_1\nabla y_1\,;$	$[2x_1 \quad 0]$
$y_4 = y_2 - y_3\,;$	$x_2 - x_1^2$	$\nabla y_4 = \nabla y_2 - \nabla y_3\,;$	$[-2x_1 \quad 1]$
$y_5 = y_4^2\,;$	$(x_2 - x_1^2)^2$	$\nabla y_5 = 2y_4\nabla y_4\,;$	$[-4x_1(x_2 - x_1^2) \quad 2(x_2 - x_1^2)]$
$y_6 = 100y_5\,;$	$100(x_2 - x_1^2)^2$	$\nabla y_6 = 100\nabla y_5\,;$	$[-400x_1(x_2 - x_1^2) \quad 200(x_2 - x_1^2)]$
$y_7 = [1\ 0]x\,;$	x_1	$\nabla y_7 = [1\ 0]\,;$	$[1\ 0]$
$y_8 = 1 - y_7\,;$	$1 - x_1$	$\nabla y_8 = -\nabla y_7\,;$	$[-1\ 0]$
$y_9 = y_8^2\,;$	$(1 - x_1)^2$	$\nabla y_9 = 2y_8\nabla y_8\,;$	$[-2(1 - x_1) \quad 0]$
$f = y_6 + y_9\,;$	$100(x_2 - x_1^2)^2 + (1 - x_1)^2$	$\nabla f = \nabla y_6 + \nabla y_9.$	$\begin{pmatrix} -400x_1(x_2 - x_1^2) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{pmatrix}^t$

Ainsi, au fil des calculs des y_i, on accumule des résultats partiels permettant de calculer ∇y_i, et ultimement, $f(x)$ et $\nabla f(x)$.

2.2. Mode inverse de la DA

Une autre variante nommée mode inverse, travaille à rebours en propageant plutôt des *quantités adjointes*. Les quantités adjointes définies comme suit

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

obéissent à une relation de récurrence :

$$y_j^* = \sum \frac{\partial f_i}{\partial y_j} y_i^*,$$

où f_i dénote comment y_j contribue au calcul de y_i. Par exemple, y_4 contribue à y_5 par la relation $y_5 = y_4^2$, donc y_4^*, pour $j = 4$, ne contribuant qu'à y_5, un seul i est impliqué et $y_4^* = 2y_4 y_5^*$. Par contre, x intervient directement en trois endroits, dans les expressions de y_1, y_2 et y_7, et les adjointes de x sont accumulées à trois endroits, encadrés dans le tableau. La chaîne des calculs est initialisée avec $f^* \equiv \frac{\partial f}{\partial f} = 1$. Nous avons retenu la notation "+ =" du langage C pour dénoter l'accumulation.

y_i	Valeur de y_i
$y_1 = [1\ 0]x\,;$	x_1
$y_2 = [0\ 1]x\,;$	x_2
$y_3 = y_1^2\,;$	x_1^2
$y_4 = y_2 - y_3\,;$	$x_2 - x_1^2$
$y_5 = y_4^2\,;$	$(x_2 - x_1^2)^2$
$y_6 = 100y_5\,;$	$100(x_2 - x_1^2)^2$
$y_7 = [1\ 0]x\,;$	x_1
$y_8 = 1 - y_7\,;$	$1 - x_1$
$y_9 = y_8^2\,;$	$(1 - x_1)^2$
$f = y_6 + y_9\,;$	$100(x_2 - x_1^2)^2 + (1 - x_1)^2$

	Adjointes	Valeurs
0.	$x^* = 0\,;$	$[0\ 0]$
1.	$f^* = 1\,;$	1
2.	$y_9^* = f^*\,;$	1
3.	$y_8^* = y_9^*(2y_8)\,;$	$2(1 - x_1)$
4.	$y_7^* = -y_8^*\,;$	$-2(1 - x_1)$
5.	$\boxed{x^* += y_7^*[1\ 0]\,;}$	$[-2(1 - x_1) \quad 0]$
6.	$y_6^* = f^*\,;$	1
7.	$y_5^* = 100y_6^*\,;$	100
8.	$y_4^* = y_5^*(2y_4)\,;$	$200(x_2 - x_1^2)$
9.	$y_3^* = -y_4^*\,;$	$-200(x_2 - x_1^2)$
10.	$y_2^* = y_4^*\,;$	$200(x_2 - x_1^2)$
11.	$y_1^* = y_3^*(2y_1)\,;$	$-400x_1(x_2 - x_1^2)$
12.	$\boxed{x^* += y_2^*[1\ 0]\,;}$	$\begin{pmatrix} -2(1-x_1) - 400x_1(x_2 - x_1^2) \\ 0 \end{pmatrix}^t$
13.	$\boxed{x^* += y_1^*[0\ 1]\,;}$	$\begin{pmatrix} -2(1-x_1) - 400x_1(x_2 - x_1^2) \\ 200(x_2 - x_1^2) \end{pmatrix}^t$

2.3. Comparaison des deux modes

L'avantage du mode direct est qu'il peut être calculé en parallèle avec la fonction proprement dite, sans mémoriser une représentation totale du GAO pour le traiter ensuite à rebours. Cependant, si cette question d'espace mémoire n'est pas critique, le mode inverse permet de calculer une fonction $f : \mathbb{R}^n \to \mathbb{R}$ et son gradient en moins de cinq fois la quantité de calcul requis pour la fonction seule (voir par exemple [6]). Le mode direct, tant qu'à lui, entraîne un coût proportionnel à n évaluations de f.

Sur l'exemple de la fonction de Rosenbrock, on voit bien que le mode direct propage des vecteurs de dimension n (dimension 2 sur l'exemple) alors que le mode inverse propage des scalaires. On s'attend donc à un coût de calcul de n fois le coût de la fonction f pour obtenir le gradient ∇f en mode direct, mais un coût du même ordre que le coût de f en mode inverse.

Le coût en termes de mémoire peut s'avérer limitatif dans certaines situation, aussi il est possible de mettre en œuvre un compromis qui mémorise une partie du GAO, et recalcule le reste [5].

2.4. Utilisation coopérative des deux modes

Il est parfois commode d'utiliser les deux modes direct et inverse lors d'un même calcul. Par exemple, soit une fonction $f : \mathbb{R}^n \to \mathbb{R}$ pour laquelle nous voulons calculer le produit $\nabla^2 f(x)v$ pour un certain vecteur v constant. Une solution efficace consiste à évaluer $\phi(x) = \nabla f(x)v : \mathbb{R}^n \to \mathbb{R}$, fonction scalaire, en mode direct puis $\nabla^2 f(x)v = \nabla\phi(x)$ en mode inverse. Le calcul de $\phi(x)$ entraîne un coût de calcul beaucoup moindre que le calcul de $\nabla f(x)$: en effet, il suffit de propager les quantités scalaires $\nabla y_i v$ plutôt que les vecteurs ∇y_i, et donc le coût de calculer $\phi(x)$ est proportionnel au coût d'évaluer $f(x)$. L'utilisation du mode inverse pour calculer $\nabla\phi(x)$ entraîne tant qu'à lui un coût borné par 5 fois le coût de $\phi(x)$.

3. Mise en œuvre de la DA

Afin de mettre en œuvre la DA, il faut au préalable procéder à l'acquisition du GAO, ou de la liste des opérations, puis, traiter le graphe d'évaluation pour produire la dérivée.

Deux approches ont été développées, une fondée sur la transformation du code, l'autre sur la surcharge des opérateurs arithmétiques. Il faut remarquer que les outils de DA impliquent une interaction importante avec le langage utilisé pour coder les fonctions, que ce soit en exploitant les possibilités de surcharge des opérateurs du langage, ou encore la transformation du code. Cet aspect constitue encore un frein à l'utilisation généralisée de la DA : il faut en effet que les fonctions à manipuler soient codées dans un langage compatible avec les outils de calcul que l'on veut utiliser.

On ne peut malheureusement pas simplement accéder au code objet d'une fonction et en faire l'appel dans nos procédures d'optimisation, mais il faut au contraire avoir un plein accès au code source des fonctions.

3.1. Transformation de code

La première approche consiste à utiliser des outils semblables aux compilateurs pour traiter le code de la fonction, et produire le code de sa dérivée. Cette approche est la plus prometteuse en termes d'efficacité entre autres parce que les codes de la fonction et de sa dérivée sont par la suite compilés, bénéficiant ainsi de toute les optimisations des compilateurs modernes. Cette approche comporte cependant un certain manque de souplesse puisqu'il faut pré-traiter les fonctions. On peut atténuer ce défaut si on se permet de provoquer la transformation depuis un programme, en utilisant par exemple des appels aux commandes de transformation, produisant ainsi dynamiquement le code objet des dérivées au besoin.

3.2. Surcharge des opérateurs et fonctions élémentaires

La seconde approche consiste à procéder à l'acquisition du graphe en surchargeant les opérateurs usuels $(+, *, -, /)$ et les fonctions élémentaires (logarithmes, trigonométriques, exponentielles, etc) pour que le calcul de la fonction enregistre en parallèle le GAO (ou encore les ∇y_i dans le mode direct). Cette approche est la plus souple, mais cette souplesse a un prix en terme d'efficacité.

Un aspect délicat provient également des structures de contrôle utilisées dans le code de la fonction f : la plupart des environnements de programmation ne permettent pas de surcharger les structures de contrôle tels les comparaisons (`if`) et les boucles (`while, for`).

3.3. Utilisation du GAO

Nous venons de voir qu'une étape importante dans la mise en œuvre d'un outil de DA est l'acquisition du graphe de calcul de la fonction à dériver. Une fois le

TABLEAU 1. Logiciels de DA.

Logiciel	langage	Transformation de code	Surcharge des opérateurs	Mode direct	Mode inverse	Ordre
ADIC	C/C++	×		•		2
ADIFOR	FORTRAN	×		•		1
ADiMat	MatLab	×	×	•		2
Adol-C	C/C++		×	•	•	>2
AUTO_DERIV	FORTRAN		×	•		2
COSY	F–C/C++		×	•		>2
CppAD	C/C++		×	•	•	>2
DIFFCODE	Scilab		×	•		1
FADBAD++	C/C++		×	•	•	>2
FDADLib	C/C++		×	•		1
GRESS	FORTRAN	×		•	•	1
NAGWare	FORTRAN			•		1
OpenAD	F–C/C++	×		•	•	>2
PCOMP	FORTRAN	×		•	•	2
SciAD	Scilab		×	•	•	>2
TAF	FORTRAN	×		•	•	1
TAMC	FORTRAN	×		•	•	1
TAPENADE	FORTRAN	×		•	•	
TOMLAB/MAD	MatLab		×	•		1

graphe acquis, on peut envisager l'utiliser pour plusieurs autres fins. Deux utilisations retiennent l'attention : la propagation des structures de zéros dans les matrices Jacobiennes, et le calcul de différences finies.

3.3.1. *Structures de zéros*

Considérons une fonction $G(x) : \mathbb{R}^n \to \mathbb{R}^m$. La matrice Jacobienne $\nabla G(x)$ est possiblement creuse, ce qui signifierait que certaines composantes x_i n'influencent pas certaines composantes $G(x)_j$. En propageant les dépendances entre les composantes de G et les composantes de x, on obtient la structure des zéros de la matrice Jacobienne.

3.3.2. *Différences finies automatiques*

Nous avons vu que la clé conduisant au processus de DA réside dans la règle d'enchaînement des dérivées. Or, une telle règle peut être obtenue pour les différences finies. Il est donc possible de calculer directement $f(x+h) - f(x)$ à partir du GAO, sans passer par l'évaluation de $f(x+h)$; on suppose que $f(x)$ a pu être calculée lors de l'acquisition du graphe.

3.4. OUTILS ACTUELLEMENT DISPONIBLES ET LEURS CARACTÉRISTIQUES

Comme on peut le constater sur le tableau 1, sauf quelques exceptions, les outils actuellement disponibles sont liés aux langages FORTRAN (77 et 95), C et C++. Notons que ce tableau ne se prétend pas exhaustif, et les derniers développements peuvent être obtenus de sites WWW tels *www.autodiff.org*. Plusieurs outils ne supportent que les dérivées d'ordre 1. Comme le mode direct est plus simple, tous les logiciels le supportent, ce qui n'est pas le cas du mode inverse.

3.5. Perspectives

La méthodologie de la DA semble assez mature, mais pour l'instant, en 2005, les outils disponibles ne sont pas encore assez répandus, et encore trop dépendants de l'environnement informatique dans lequel ils sont utilisés. Cette situation évolue assez rapidement, cependant. Nous en traçons ici une esquisse.

Les outils répandus et stables sont de nature assez rigide. ADIFOR et ADIC permettent la transformation du code d'une fonction en un code enrichi calculant la fonction et sa dérivée d'ordre un, ainsi que l'exploitation de l'éparsité des matrices Jacobiennes.

Par ailleurs, ADOL-C est beaucoup plus flexible, conçu selon la technique de surcharge des opérateurs en C++.

OpenAD est un projet ambitieux conçu selon la technique de transformation de code. Une interface ("front end") fait la traduction du code dans un langage interne qui peut être traité pour fins de DA, d'optimisation de code, etc. Le résultat est alors retraduit dans le langage d'origine.

SciAD, développé en Scilab, est un outil très général. L'acquisition du graphe est réalisée par la surcharge des opérateurs, et son traitement permet la dérivation, la manipulation, la différence finie automatique.

CppAD est un nouveau venu, et ses spécifications d'utilisation sont similaires à ADOL-C, mais plus efficace selon les tests de son auteur.

En résumé, les outils actuels proposent des compromis, les plus efficaces étant fondés sur la transformation du code des fonctions, doivent être utilisés séparément des programmes les utilisant. Les outils conçus selon la surcharge des opérateurs s'utilisent plus facilement au sein des programmes d'application, mais leur utilisation entraîne une surcharge de calculs.

3.5.1. *Quantification du coût de la surcharge des opérateurs*

En utilisant un exemple fourni avec la distribution de ADOL-C, la fonction de Rosenbrock généralisée en plusieurs dimensions, nous pouvons nous faire une idée du coût associé à la technique de surcharge des opérateurs. En utilisant une dimension de 50 000 (pour avoir un coût de calcul significatif), l'évaluation de la fonction proprement dite prend 5×10^{-4} s, l'acquisition du GAO 5×10^{-2}, l'évaluation de f à partir du GAO 6.5×10^{-3} et l'évaluation de $\nabla f(x)$ 4×10^{-2} s.

4. Utilisation de la DA en optimisation

En optimisation différentiable, les gradients et Hessiens des fonctions objectif, et les Jacobiens des contraintes constituent les outils mathématiques permettant de caractériser les solutions (conditions d'optimalité) et également de produire des algorithmes de résolution. Il est donc naturel d'espérer une grande utilité de la part d'un outil automatisant le calcul des dérivées.

Un premier niveau d'utilisation d'un outil de type transformation de code est de produire automatiquement les dérivées des fonctions objectif et de contraintes, et

de fournir ces fonctions dérivées au logiciel d'optimisation. S'il modifie son modèle, l'utilisateur est donc responsable de mettre à jour le code de ses dérivées.

Une utilisation plus pratique consiste à concevoir le logiciel de résolution pour qu'il produise lui-même les fonctions de dérivées premières et secondes dont il a besoin. Ceci n'est cependant pas aussi simple qu'il le paraît : le logiciel de résolution doit manipuler le *code* des fonctions pour produire (lui même, ou en invoquant un logiciel de DA) les dérivées. Si une portion des fonctions objectif ou de contraintes n'est pas disponible sous forme source, ou encore si les fonctions sont composées de sous-programmes issus de diverses sources, écrits dans des langages variés, ou encore utilisant des librairies dont le code source n'est pas disponible, il est impossible d'appliquer directement un outil de DA.

Malgré ces difficultés, au moins trois tentatives de fournir un logiciel de résolution de programmes mathématiques produisant lui même les dérivées dont il a besoin ont été développées récemment.

4.1. FSQP

Le logiciel FSQP (*Feasible Sequential Quadratic Programming*) a été un des premiers, semble-t-il à offrir une interface avec un logiciel de DA, ADIFOR, atténuant le travail de l'usager qui n'a plus à fournir le code des dérivées de ses fonctions, ni de les produire lui-même en utilisant un outil de DA externe à FSQP.

4.2. NEOS

Le serveur Web NEOS offre les service de nombreux logiciels de résolution de programmes mathématiques. Afin d'offrir ce service au plus grand nombre d'utilisateurs, les fonctions objectif et de contraintes peuvent être spécifiées avec un langage de modélisation (GAMS, AMPL) ou encore en C, C++ ou FORTRAN dépendant des logiciels de résolution. La prochaine section traite de AMPL. Pour les logiciels qui acceptent des fonctions dans un langage de programmation, NEOS ne demande à l'usager que les fonctions proprement dites, les dérivées étant produite préalablement à l'utilisation du logiciel de résolution à l'aide de ADIFOR, ADIC ou ADOL-C selon le langage de programmation utilisé par l'usager.

Dans le contexte d'un serveur Web de logiciels de résolution de programmes mathématiques, ces choix sont tout-à-fait justifiés, et nul ne se plaindra des performances absolues de l'utilisation de la DA, et tous se réjouiront de ne pas avoir à fournir de gradients ni de Hessiens ou Jacobiens des fonctions des modèles. Malgré tout, tel que décrit dans [7], l'usager peut s'attendre à des performances élevées car les choix d'outils de DA ont été soigneusement implantés avec un souci d'efficacité.

4.3. AMPL

AMPL (*A Mathematical Programming Language*) est un langage algébrique de modélisation. Le langage, initialement défini pour formuler des modèles linéaires, a

bénéficié de plusieurs extensions, et permet la modélisation de virtuellement n'importe quel modèle de programmation mathématique. AMPL n'est pas un logiciel de résolution. Un produit commercial implante le traitement de modèles décrits selon ce langage, et fournit aux logiciels de résolution un cadre leur permettant de calculer les fonctions objectif et de contraintes, de connaître les bornes des variables, etc. Un point fort de cet outil réside dans le pré-traitement des modèles, qui a été naturellement enrichi de possibilités de DA.

Les modèles écrits en AMPL peuvent donc, via l'outil précédemment mentionné, fournir aux logiciels de résolution les dérivées premières, et les calculs d'un produit du Hessien par un vecteur ; on peut évidemment retrouver le Hessien par n produits avec les vecteurs canoniques.

Encore une fois, pour l'utilisateur, la situation est idéale.

5. Utilisation future de la DA en optimisation

Dans ce qui précède, nous avons survolé l'utilisation de la DA pour éviter de programmer manuellement les calculs de dérivées de fonctions. C'est déjà un grand pas. En effet, l'effort qu'un utilisateur doit déployer pour fournir des gradients et Hessiens de ses modèles, et les maintenir au fil de l'évolution du modèle, n'est pas du tout négligeable.

Dans ce qui suit, nous envisageons l'avenir, et amorçons la réflexion suivante : *si les calculs de dérivées sont facilement automatisés, à un ordre arbitraire, que pouvons-nous faire de plus ?*

Cette section décrit quelques premières tentatives de l'auteur pour exploiter ce nouvel outil informatique. Ces tentatives sont reliées aux intérêts pour les algorithmes de pénalité/barrière en programmation non-linéaire, plus particulièrement les techniques d'extrapolation. Ces techniques sont étroitement reliées aux concepts de trajectoire centrale dans les méthodes de point intérieur. Afin de simplifier l'exposé, limitons-nous à présenter une interprétation non-standard de la méthode de Newton-Raphson, et à en obtenir une généralisation d'ordre supérieur, immédiatement implantable si les outils appropriés de DA sont disponibles. Nous esquisserons également un algorithme d'extrapolation pour la fonction de pénalité quadratique dans le contexte d'optimisation avec contraintes d'égalité.

5.1. Méthode de Newton-Raphson généralisée

Plaçons-nous dans le contexte de résoudre un système $F(x) = 0$ avec une fonction $F : \mathbb{R}^n \to \mathbb{R}^n$ plusieurs fois continûment différentiable. Par rapport à ce qui précède, considérons que $F(x) = \nabla f(x)$, où notre problème est de résoudre $\min_{x\in\mathbb{R}^n} f(x)$.

La méthode itérative de Newton-Raphson consiste à résoudre la linéarisation du système

$$l_0(x) = F(x_0) + \nabla F(x_0)(x - x_0) = 0$$

pour définir le prochain point (disons x_1). Une interprétation inhabituelle consiste à remarquer qu'en x_0, $F(x_0) = r_0$, un certain résidu. En normalisant le résidu

$\bar{r}_0 = \frac{r_0}{\|r_0\|}$ et en écrivant $F(x_0) = \tau_0 \bar{r}_0$, et en utilisant une fonction implicite $x(\tau)$, telle que $x(\tau_0) = x_0$ on définit une extrapolation $\hat{x}_0^1 = x(\tau_0) + \dot{x}(\tau_0)(-\tau_0)$. Or, il se trouve que $l_0(\hat{x}_0^1) = 0$, et donc le point $\hat{x}_0^1$ est précisément le prochain point des itérations de la méthode de Newton-Raphson.

5.1.1. *Méthode de tenseurs*

On pourrait envisager de généraliser la méthode en utilisant une approximation de F d'ordre supérieur à un ; cette idée a conduit aux méthodes dites de tenseur, et permet une certaine amélioration notamment dans le cas de solutions où $\nabla F(x^*)$ n'est pas inversible. Cependant, la manipulation du tenseur d'ordre 2 est déjà suffisamment ardue qu'il semble impossible d'exploiter des modèles d'ordre plus grand que 2 dans ce contexte, même si leur définition à l'aide d'outils de DA est rendue accessible.

5.1.2. *Extrapolations d'ordres supérieurs*

On peut également généraliser la méthode en définissant un point $\hat{x}_0^2 = x(\tau_0) + \dot{x}(\tau_0)(-\tau_0) + \frac{1}{2}\ddot{x}(\tau_0)(\tau_0)^2$ [1]. On voit immédiatement qu'une extrapolation d'ordre 1 (itération de Newton-Raphson) devrait posséder un ordre de convergence quadratique, la généralisation d'ordre 2 un ordre de convergence cubique. On peut alors généraliser l'extrapolation à des ordres supérieurs :

$$\hat{x}^p(0) = x(\tau) + \dot{x}(\tau)(0-\tau) \ldots + \frac{1}{p!} x^{(p)}(\tau)(0-\tau)^p$$

nous procurant ainsi une itération possédant un ordre de convergence $p+1$.

Calculs des $\hat{x}^p$. Utilisant le théorème des fonctions implicites, nous obtenons

$$\nabla F(x_k)\dot{x}(\tau_k) - \bar{r}_k = 0,$$

qui nous permet de reconnaître en effet l'itération de Newton-Raphson. Pour obtenir les extrapolés d'ordre supérieur, nous utiliserons une formulation récursive :

$$\nabla F(x(\tau))x^{(p)} + B_p(\dot{x}, \ddot{x}, \ldots, x^{(p-1)}) = 0$$

où

$$B_p(\dot{x}, \ddot{x}, \ldots, x^{(p-1)}) = \nabla_\tau \left(\nabla F(x)\bar{x}^{p-1} + B_{p-1}(\dot{x}, \ddot{x}, \ldots, x^{(p-2)}) \right),$$

avec $B_1 = 0$ et $\bar{x}^p$ un vecteur constant de valeur $x^{(p)}$. On voit donc que les extrapolés sont obtenus par récurrence.

Par exemple, détaillons le calcul de $\hat{x}^2$. $B_2 = \nabla^2 F(x)\dot{x}(\tau)\dot{x}(\tau) = \nabla_\tau(\nabla F(x(\tau))\bar{x}^1)$ et donc,

$$\ddot{x}(\tau) = -\nabla F(x(\tau))^{-1} \bar{B}_2.$$

Maintenant, développons le calcul de B_2 en utilisant la DA. Définissons $\phi(x) = F(x)^t \bar{x}^1$, puis $\psi(x) = \nabla\phi(x)\bar{x}^1$. Alors, $B_2 = \nabla\psi(x)$. Comme les calculs impliquent

l'obtention des gradients de fonctions scalaires ϕ et ψ, chacun de coût borné par 5 fois le coût de la fonction F, le calcul de $\hat{x}^2$ s'obtient à un coût borné par 25 fois le coût de calculer F. D'un autre côté, obtenir ∇F coûte n fois le coût de calcul de F. Le coût de calcul de $F = \nabla f$ est quant à lui borné par 5 fois le coût de calcul de f. Il est raisonnable de considérer que le coût de calcul de f est d'au moins $\mathcal{O}(n^2)$, de sorte que les opérations qui dominent l'effort de calcul sont l'obtention de ∇F et la solution des systèmes linéaires.

Remarques sur l'efficacité. Les calculs de ∇_τ seront évidemment obtenus par DA. Remarquons d'emblée que l'obtention des $x^{(p)}$ provient d'un système d'équations linéaires dont la matrice $\nabla F(x_0)$ *est la même* pour tous les $x^{(p)}$. Par conséquent, le coût de calcul pour résoudre les systèmes linéaires sera $\mathcal{O}(n^3)$ pour le premier système, mais seulement $\mathcal{O}(n^2)$ pour les suivants si une méthode directe de résolution est retenue (Cholesky), et si les facteurs de Cholesky sont réutilisés lors des calculs subséquents.

Ainsi, l'utilisation des extrapolés d'ordre supérieur sera intéressante si les membres de droite des systèmes linéaires sont effectivement obtenus à moindre coût que $\mathcal{O}(n^3)$; autrement, il sera plus efficace d'utiliser une itération de Newton-Raphson usuelle.

Tel qu'observé dans [2], il est important de passer par les fonction ψ et ϕ pour calculer $\hat{x}^2$, même si on dispose déjà de ∇F. En effet, l'utilisation de $\nabla F(x)\bar{x}^1$ est significativement moins efficace, suffisamment pour que la méthode de Newton-Raphson usuelle soit préférable à la variante d'ordre 2 alors que le calcul recommandé conduit à une version d'ordre 2 légèrement plus efficace que l'itération usuelle.

Robustesse et efficacité. Les variantes d'ordre supérieur de l'itération de Newton-Raphson promettent une efficacité accrue, confirmée sur de petits exemples dans une version d'ordre 2[2]. De plus, comme on utilise un développement d'ordre supérieur de la fonction $x(\tau)$, on est en droit de s'attendre à pouvoir traiter des fonctions que la méthode de Newton-Raphson pure ne peut pas traiter. Par exemple, si $F(x) = \sqrt[3]{x}$, l'itération usuelle ne converge pas alors qu'une itération d'ordre 3 est exacte en une seule itération [1].

5.2. Méthodes de pénalité

L'interprétation inhabituelle de l'itération de Newton a été inspirée des techniques d'extrapolation de trajectoires au sein de méthodes de pénalités. Ces aspects sont étroitement reliés aux méthodes prédicteur-correcteur dans les algorithmes de point intérieur.

Plaçons-nous dans le contexte simple de l'optimisation sous contraintes d'égalité :

$$\begin{aligned} \min \quad & f(x) && (1)\\ \text{sujet à} \quad & g(x) = 0. && (2) \end{aligned}$$

La fonction de pénalité quadratique classique s'écrit

$$\phi(x,\rho) = f(x) + \frac{1}{2\rho}\|g(x)\|^2,$$

et ses conditions d'optimalité

$$\nabla_x \phi(x,\rho) = \nabla f(x) + \frac{g(x)^t}{\rho}\nabla g(x) = 0$$

peuvent se reformuler sous forme primale-duale :

$$\Phi(x,\lambda,\rho) = \begin{pmatrix} \nabla f(x) + \lambda \nabla g(x) \\ g^t(x) - \rho\lambda \end{pmatrix} = 0. \tag{3}$$

Lorsque $\rho = 0$, (3) constitue simplement les conditions d'optimalité du programme (1, 2). La notion d'extrapolé est classique pour ces méthodes, et consiste à observer que le système (3) définit par le théorème des fonctions implicites une fonction $x(\rho)$ et $\lambda(\rho)$, et ces fonctions sont différentiable en $\rho = 0$ au voisinage d'une solution x^* et λ^* satisfaisant aux conditions suffisantes de second ordre et pour laquelle $\nabla g(x^*)$ est une matrice de plein rang.

Or, sous ces mêmes conditions, en tenant compte du fait que les sous-problèmes pénalisés ne seront pas résolus de manière exacte, et en exprimant le résidu $\tau\bar{r}$, le système

$$\Phi(x,\lambda,\rho,\tau) = \begin{pmatrix} \nabla f(x) + \lambda \nabla g(x) \\ g^t(x) \end{pmatrix} - \begin{pmatrix} \tau\bar{r} \\ \rho\lambda \end{pmatrix} = 0 \tag{4}$$

définit des fonctions $x(\tau,\rho)$ et $\lambda(\tau,\rho)$ différentiables en $\tau = 0$ et $\rho = 0$. On se propose donc de définir un extrapolé pour estimer $x(0,\rho^+)$ à partir d'une solution approchée $x(\tau,\rho)$.

Tout comme dans les itérations de Newton d'ordre supérieur, l'utilisation d'extrapolés d'ordre supérieur définis comme

$$\hat{x}^p_{k+1} = \hat{x}^{p-1}_{k+1} + \frac{1}{p!}\sum_{j=0}^{p}\binom{p}{j}\frac{\partial^p x}{\partial\rho^{p-j}\partial\tau^j}(\rho_{k+1}-\rho_k)^{p-j}(-\tau_k)^j \tag{5}$$

conduit à un ordre de convergence asymptotique de $\frac{p+1}{2}$.

Le cas $p = 1$ a été étudié depuis plus longtemps, et il faut l'incorporer dans un schéma prédicteur-correcteur pour obtenir un ordre de convergence presque quadratique.

5.3. Différences finies automatiques

Dans le contexte de méthodes de descente d'optimisation, que ce soit de type recherche linéaire ou encore région de confiance, il faut comparer la fonction de mérite $m(x)$ en deux itérations successives.

Par exemple, considérons la minimisation d'une fonction deux fois continûment différentiable $f(x)$ par une méthode de descente de type recherche linéaire où le prochain itéré $x^+ = x + \theta d$, avec d une direction de descente ($\nabla f(x)d < 0$) et θ choisi pour que $f(x^+) < f(x)$. Les méthodes de descente de type région de confiance utilisent également une comparaison des valeurs de f en deux itérés successifs x et x^+. Dans d'autres contextes, d'algorithmes d'optimisation sous contrainte, la fonction de mérite $m(x)$ n'est pas forcément identifiée à la fonction objectif, mais tient compte également de la satisfaction des contraintes.

Observons que lorsque la fonction de mérite est différentiable et possède un minimum local à la solution, ses dérivées s'annulent et $m(x)$ croît quadratiquement au voisinage de la solution. Ceci implique que dès que les itérations se sont rapprochées de l'ordre ϵ de la solution, la fonction de mérite sera à ϵ^2 près de sa valeur à l'optimum. Une conséquence fâcheuse de cette observation est *qu'il est impossible de témoigner du progrès de l'algorithme* dès qu'il s'approche à $\sqrt{\epsilon_M}$ de la solution, où ϵ_M est la précision machine.

Robustesse et efficacité. L'utilisation de différences finies automatiques améliore grandement la robustesse de l'itération de Newton-Raphson, tel que documenté dans [3]. Une estimation plus précise de la descente peut également réduire le nombre d'itérations dans certains cas, mais nous n'espérons pas améliorer l'efficacité du calcul de $f(x+h)-f(x)$ par l'utilisation de la différence finie automatique qui calcule directement $\Delta_h f(x)$.

6. Conclusion

Nous avons introduit les idées de base de la différentiation automatique dans l'optique de son utilisation en optimisation.

Nous avons constaté que déjà, ces outils sont utilisés avec succès dans des environnements de résolution de programmes mathématiques tels le logiciel FSQP, le système AMPL et le serveur NEOS.

Pour les chercheurs en algorithmique d'optimisation, les perspectives sont encourageantes, et les possibilités d'utilisation avancée de ces techniques semblent sur le point d'éclore avec des projets tels SciAD, OpenAD, CppAD.

Un bémol cependant : il faut disposer du code source des fonctions que nous voulons traiter, contrairement à d'autres outils d'estimation des dérivées tels l'utilisation de différences finies, qui ne nécessitent que la boîte noire effectuant les calculs.

Remerciements. Les commentaires de Benoit Hamelin et Jean-Louis Merrien ont contribué à l'amélioration de mon manuscrit, et je les en remercie !

Références

[1] J.-P. Dussault, High order Newton-penalty Algorithms. *J. Comput. Appl. Math.* **182** (2005) 117–133.

[2] J.-P. Dussault and B. Hamelin, Implementation of high order Newton Algorithms, in *Scilab first international conference*, Dec. 2004.

[3] J.-P. Dussault and B. Hamelin, Robust descent in differentiable optimization using automatic finite differences. *Optimization Methods and Software* (2005). À paraître.

[4] A. Griewank, On automatic differentiation, in *Mathematical Programming : Recent Developments and Applications*, edited by M. Iri and K. Tanabe, Kluwer Academic Publishers (1989) 83–108.

[5] A. Griewank, Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software* **1** (1992) 35–54.

[6] A. Griewank, *Evaluating Derivatives : Principles and Techniques of Algorithmic Differentiation*. Frontiers in Appl. Math. **19** SIAM, Philadelphia, PA (2000).

[7] J.J. Moré, Automatic differentiation tools in optimization software, in *Automatic Differentiation of Algorithms : From Simulation to Optimization*, Computer and Information Science, Springer, New York, NY (2001) Chapter 2, 25–34.