Domain of Validity of Derivatives Computed by Automatic Differentiation

Mauricio Araya-Polo, Laurent Hascoët

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Domain of Validity of Derivatives Computed by Automatic Differentiation

Mauricio Araya-Polo — Laurent Hascoët

N° 5237

June 2004

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5237--FR+ENG

Domain of Validity of Derivatives Computed by Automatic Differentiation

Mauricio Araya-Polo , Laurent Hascoët

Thème NUM — Systèmes numériques
Projet TROPICS

Rapport de recherche n° 5237 — June 2004 — 16 pages

Abstract: Automatic Differentiation (AD) is a technique to obtain derivatives of the functions computed by a computer program. Due to the control flow, these derivatives are valid only in a certain domain around the current input values. We investigate methods to evaluate this domain. This results in a new specific mode of AD that returns additional information on the domain of validity.

Key-words: derivative, validity, automatic, differentiation, control flow, input domain.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Domaine de Validité des Dérivées Calculées par la Différentiation Automatique

Résumé : La Différentiation Automatique (DA) est une technique pour obtenir les dérivées de fonctions calculées par un programme informatique. À cause du flot de contrôle, ces dérivées ne sont valides que dans un certain domaine autour de la valeur d'entrée actuelle. Nous recherchons une méthode pour évaluer ce domaine. Cela nous amènè à définir un nouveau mode spécifique de DA qui renvoie des informations complémentaires sur le domaine de validité.

Mots-clés : differentiation automatique, dérivées, validité, flot de contrôle, input domaine.

1 Introduction

Automatic Differentiation (AD) is a technique to obtain derivatives through the application of the chain rule on source code. Viewing source code as sequences of instructions, and instructions as elementary mathematical functions, the chain rule can be applied to obtain a new source code which includes the original instructions plus new instructions for the derivatives.

AD is used in several application areas, each using complex computational models. For example we can mention: Numerical Methods [Klein96], Optimization [Talagrand91], Sensitivity Analysis [Kim01], Data Assimilation and Inverse Problems [Restrepo98].

There exists among others two fundamental modes of AD: tangent and reverse. The tangent mode computes directional derivatives, i.e. the first-order effect on the output resulting from a small modification of some inputs following a given direction. Conversely, the reverse mode computes gradients, i.e. given a linear combination of the output, it returns the direction in the input space that maximizes the increase of the combined output. In theory, the reverse mode is cheaper to compute when the number of independent variables is larger than the dependent variables.

Currently, AD models do not include any verification of the differentiability of the functions. Therefore, it may happens that AD returns some derivatives, that may not be valid because the original functions were not differentiable. We can assume that most of this non-differentiability condition is introduced by *if...then...else* statements.

Our goal in this work is to evaluate the interval around the input data inside which no non-differentiability problem arises. Practically, this requires to analyzing each test at run-time, in order to find for which data it will switch, and to propagate this information as a constraint on the input data. We propose a new method of AD which returns this validity information. Further, we discuss the complexity of this mode, and how it can be improved.

This paper is organized as follows: in Section 2 we give the basics concepts of AD. In Section 3 we review the literature about the problem we are addressing. In Section 4 we introduce our approach. Finally, we discuss the future work and the conclusions in Section 5.

2 Presentation of Automatic Differentiation

Let us present the AD framework. Programs are sequences of instructions, like:

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

, where each instruction represents an elementary function f_i, and the composition of functions returns the mathematical model:

$$F = f_p \circ f_{p-1} \circ ... \circ f_2 \circ f_1 \; with$$

$$F : X \in I\!R^n \rightarrow Y \in I\!R^m, \;\; and \, Y = F(X)$$

We present the fundamental modes of AD. First, the tangent mode which computes directional derivatives. Second, the reverse mode which computes adjoint values of gradients.

2.1 Forward Mode

When the chain rule is applied to elementary functions, the results are jacobian matrices f_i', where $x = x_0$ represent the input variables, and $x_{p-1} = f_{p-1} \circ ... \circ f_2 \circ f_1$ the intermediate variables. Using the previous notation, the derivative of a function F, F', is the multiplication of the jacobians f_i',

$$F'(X) = f_p'(x_{p-1}) \cdot f_{p-1}'(x_{p-2}) \cdot ... \cdot f_1'(x_0)$$

$$F', P' : X', X \in I\!R^n \rightarrow Y' \in I\!R^{mxn}, \;\; with \;\; Y' = F'(X)dX$$

The differentiated program P' has the following sequence of instructions:

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

From a computational point of view, the differentiated program can be easily built. The differentiated program is composed by the original instructions necessary to compute the derivatives, and by the instructions which represent the derivatives. The differentiated program maintains the original program flow control structure.

Example of Forward Mode of AD	
Original Code	**Differentiated Code**
subroutine sub1(x,y,z,o1)	subroutine sub1_d(x, xd, y, yd, z, zd, o1, o1d)
I_1 x = y * x	
I_2 r = x * x + y * y	I'_1 xd = yd * x + y * xd I_1 x = y * x I'_2 rd = 2 * x * xd + 2* y * yd I_2 r = x * x + y * y
if (r $>$ 10) then	if (r $>$ 10) then
I_3 r = 10	I'_3 rd = 0.0 I_3 r = 10
endif	endif
I_4 o1 = r * z	I'_4 o1d = rd * z + r * zd I_4 o1 = r * z
end	end

Table 2.1: Application of AD to source code of example, using TAPENADE tool [Tropics04]

The original and differentiated example programs have two valid behaviors, because the control flow structure.

$$P = I_1; I_2; [I_3; I_4 || I_4]$$

$$P' = I'_1; I_1; I'_2; I_2; [I'_3; I_3; I'_4; I_4 || I'_4; I_4]$$

The example has a branch. Consequently, it is only piecewise differentiable [Griewank00]. Unfortunately, the differentiated version does not take this into account, and returns a derivative even when the variable $r = 10$.

In the case when the conditional expression depends on a variable which has a derivative, small changes in the input values may return totally different derivatives.

2.2 Reverse Mode

When we have scalar functions, we are interested in calculating the gradient. To calculate the gradient, we build a vector $\bar{Y}$ that contains the weights of each component of the original vector Y. Further, the result of vector$\times$vector operation will be the linear combination of elements $\bar{Y} \cdot Y = \bar{Y}^t \cdot F(X)$. After transposition, the gradient is

$$\bar{X} = f_p^{'t}(x_{p-1}) \cdot f_{p-1}^{'t}(x_{p-2}) \cdot ... \cdot f_1^{'t}(x_0) = F^{'t}(X) \cdot \bar{Y} \quad (1)$$

$$F', \bar{P} : \bar{Y}, X \in I\!R^n \rightarrow \bar{X} \in I\!R^{mxn}, \ \ with \ \ \bar{X} = F^{'t}(X) \cdot \bar{Y}$$

Equation (1) is cheap to compute if it is executed in reverse order, because matrix×vector products are more efficient than matrix×matrix products. Unfortunately, this mode of AD has a difficulty because the f_i' instructions require the intermediate values x_i computed in the original order. The intermediate values may have been overwritten, and lost for future calculations. There are two main strategies to access the intermediate variables: recompute-all or store-all [Griewank92].

Reverse mode of AD generates a new program $\bar{P}$. Program $\bar{P}$ has two parts, the first is called the forward sweep $\overrightarrow{P}$, the forward sweep is basically the original program P plus some instructions to fulfill the storage/re-computation trade-off; the second part is called the reverse sweep $\overleftarrow{P}$, reverse sweep consists of the instructions that represents the fundamental functions $f_i^{'t}(x)$ from (1), and the instructions to recover the needed intermediate values.

The reverse differentiated version of the program P is as follows:

Original Code	**Differentiated Code**
subroutine sub1(x,y,z,o1)	subroutine sub1_b(x, xb, y, yb z, zb, o1b)
I_1 $x = y * x$	PUSH(x)
I_2 $r = x * x + y * y$	I_1 $x = y * x$
	I_2 $r = x * x + y * y$
if (r $>$ 10) then	if (r $>$ 10) then
I_3 $r = 10$	I_3 $r = 10$
	PUSH(1)
endif	else
	PUSH(0)
	endif
I_4 $o1 = r * z$	$\overleftarrow{I_4}$ $\begin{cases} rb = z * o1b \\ zb = zb + r * o1b \end{cases}$
end	o1b = 0.0
	POP(branch)
	if (branch == 1)
	$\overleftarrow{I_3}$ $rb = 0.0$
	endif
	$\overleftarrow{I_2}$ $\begin{cases} xb = xb + 2 * x * rb \\ yb = yb + 2 * y * rb \end{cases}$
	POP(x)
	$\overleftarrow{I_2}$ $\begin{cases} yb = yb + x * xb \\ xb = y * xb \end{cases}$
	end

Table 2.2: Application of AD to example, using TAPENADE tool

Where $PUSH(x)$ and $POP(x)$ are the functions to store and recover values of needed variables. Notice that sometimes not all original instructions are needed in the forward sweep; some of them are dead code [Hascoet04] for the derivatives (for example, in $\overleftarrow{P}$ the instruction I_4).

3 Related Work

This section is a selection of works from the literature, all these works are related to the problem of calculating derivatives, especially when the functions involved have problems of differentiability. We discuss two approaches, Interval Extension and Laurent Series.

3.1 Interval Extension

The goal of this approach is to deal with non-differentiability functions using interval extensions, in particular with conditional statements. The interval extension $F(X)$ is the interval which encloses the extreme values of the function results, $\{f(x) \mid x \in X\} \subseteq F(X)$, where X is the input domain. Functions with branches are represented as follows:

$$F(X) = \chi(x_s, x_q, x_r) = \begin{cases} x_q & \text{if } x_s < 0 \\ x_r & \text{if } x_s > 0 \\ x_q \underline{\cup} x_r & otherwise \end{cases}$$

Where, x_s is the decision expression, x_q and x_r are the interval evaluations of function of each branch, and $x \underline{\cup} y$ is the interval hull of the interval evaluation of functions x and y.

Interval extension is applicable when the functions are smooth. A new property is necessary to adapt interval extension to non-smooth functions.

$$F(X_2) - F(X_2) = A(X_2) - A(X_1)$$

Derivative extension $F'(X)$ is a jacobian matrix, $A \in F'(X)$ and $X_1, X_2 \in X$. The property yield that $F'(X)$ must be a bounded, closed and convex set as Lipschitz sets [Neumaier90].

The derivative extension (when $\chi(0^-, x_q, x_r) = \chi(0^+, x_q, x_r)$) is defined as follows:

$$\frac{\partial \chi(x_s, x_q, x_r)}{\partial x_q} = \begin{cases} 1 & \text{if } x_s < 0 \\ 0 & \text{if } x_s > 0 \\ [0,1] & otherwise \end{cases} \quad , \quad \frac{\partial \chi(x_s, x_q, x_r)}{\partial x_r} = \begin{cases} 0 & \text{if } x_s < 0 \\ 1 & \text{if } x_s > 0 \\ [0,1] & otherwise \end{cases}$$

$$, \frac{\partial \chi(x_s, x_q, x_r)}{\partial x_s} = 0$$

This approach was developed to verify solutions of non-linear systems of equations and for global optimization [Kearfott96].

3.2 Subdifferencials

The subdifferential of a function f at point x_0 is the set

$$\partial f(x_0) = \{y : f(x) - f(x_0) \geq \langle y, x - x_0 \rangle\}$$

For a function of one variable this is the collection of angular coefficients y for which the lines $f(x_0) + y(x - x_0)$ lie under the graph of f.

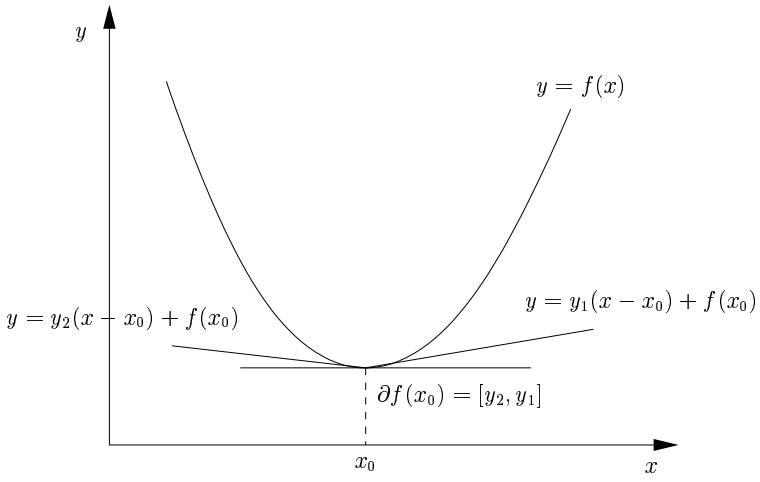

Figure 1: Subdifferential example.

The subdifferential is a concept which generalizes the derivative for a convex function, and if f is differentiable at x_0, then $\partial f(x_0) = f'(x_0)$. The subdifferential concept belongs to the so called “convex calculus", which is widely used in convex analysis, therefore in optimization research [Gamkrelidze87][Hiriart91].

The are several generalizations of the derivative [Griewank00]. But, all of them, became very complex to be useful in AD, particularly about the rules to compute the derivatives.

3.3 Laurent Series

This method handle functions with known troublesome inputs. The method is based in Laurent series.
Laurent series are one-side expansions of problematic functions, like $|x|$ and $x^2 log(x)$ at the origin. The method requires that user’s input be a point x_0 for the evaluation and differentiation and also a direction x_1, then the directional differentiation can be perform in effective way.

The problem arises when the method obtains roots with an uncertain sign, i.e. when the number of significant terms may vary from intermediate to intermediate steps. However, the Laurent series can be truncated, thus becoming a Taylor polynomial. Hence, the method is applicable as long as we deal with the latter [Griewank00].

4 Interval of Validity

We identify two main sources of non-differentiability in programs: first, the use of intrinsic functions of the language that are not differentiable for their current inputs, second, the changes in the control flow that break the continuity of the functions (*if...then...else*) sentences.

The first problem is already handled by the replacement of the *bad* functions by others *harmless* functions [Griewank95], but these *harmless* functions include control flow changes, then they degenerate in problems of the second source. The second source of problems is the only fundamental one, and is the problem that we will study, differentiability through changes in the control flow.

Our goal is to inform to the end-user when there is a risk that the returned derivatives are invalid. By invalid we mean that the returned derivatives are too close to a switch of control, therefore the derivatives may be totally different. Our idea is to evaluate the largest interval around the current input data, such that there is not differentiability problem if the input remains in this interval. In the case when this interval is notably too small, this will be a warning to the user against an invalid use of these derivatives.

4.1 Our Approach

We consider programs as straight-line blocks of code B_i, separated by conditions T_i that may change the control flow.

$$X \xrightarrow{B_1} (X_1, T_1) \cdots \xrightarrow{B_n} (X_n, T_n) \xrightarrow{B_{n+1}} Y$$

Let us consider a conditional T_i in isolation. It uses variables from X_i, which depends differentiably on the input X (at first order) by:

$$\Delta X_i = J(B_1; B_2; ...; B_i) \cdot \Delta X = J(B_i) \cdot J(B_2) \cdot ... \cdot J(B_i) \cdot \Delta X$$

Where $J(B)$ is the jacobian of the function implemented by code B. We can admit without loss of generality, that T_i is just a test on the sign of one variable x_j in X_i. Therefore, the conditional will switch if $-\Delta x_j \geq x_j$. Thus, we can state the condition on ΔX upon which the program control does not switch for this test T_i:

$$- < J(B_1) \cdot J(B_2) \cdot ... \cdot J(B_i) \cdot \Delta X | e_j > \geq < x_j | e_j > \tag{2}$$

For the entire program, the computed derivatives will be valid if the variation ΔX of the input X satisfies all the constraints (2) for each test T_i. This gives a system of constraints on ΔX. The solution of this system is the space/interval where the derivatives are valid. In general, the system is large and expensive to solve.

4.2 Differentiation Model

To implement the previous method we need to compute several jacobians, the cost to compute each jacobian in forward mode of AD is proportional to the dimension of the inputs space.

Observing equation (2), and recalling that we must solve it for ΔX, we must isolate ΔX. A powerful way to do that is to transpose the jacobians in the dot product, yielding the equivalent equation:

$$- < \Delta X \, | \, J^t(B_i) \cdot \ldots \cdot J^t(B_2) \cdot J^t(B_1) \cdot e_j / x_j > \geq \; 1 \tag{3}$$

We remark that the right side of the dot product is directly computed by the reverse mode of AD (see section 2.2).

To begin with, let us see how our approach translate into a differentiation model for a simple program (table 2.1) with just one test. If the program $P = B_1; T_1; B_2$ then the following program $\check{P}$ returns the condition on ΔX to keep T_1 from switching:

$$\check{P} \; = \overrightarrow{B_1}; \check{T_1}; \overleftarrow{B_1}$$

Things get a little more complex when program P contains two tests. If $P = B_1; T_1; B_2; T_2; B_3$ then the program which return the constraint on ΔX due to T_1 and T_2 are respectively:

$$\begin{array}{l} \check{P_1} = \overrightarrow{B_1}; \check{T_1}; \qquad\qquad\quad ; \overleftarrow{B_1} \\ \check{P_2} = \overrightarrow{B_1}; T_1; \overrightarrow{B_2}; \check{T_2}; \overleftarrow{B_2}; \overleftarrow{B_1} \end{array}$$

Instead of running $\overrightarrow{B_1}$ and $\overleftarrow{B_1}$ twice, we can use the so-called vector approach, which computes several sets of derivatives for the same original values. We shall denote $\overleftarrow{B_i}^{(n)}$ a vector execution of $\overleftarrow{B_i}$ on n distinct set of derivatives. Thus, we can build a single program $\check{P_1}$

$$\check{P} \; = \overrightarrow{B_1}; \check{T_1}; T_1; \overrightarrow{B_2}; \check{T_2}; \overleftarrow{B_2}; \overleftarrow{B_1}^{(2)}$$

For a general program P such as (4), the domain-validated program $\check{P}$ is as follows:

$$\check{P} \; = \overrightarrow{B_1}; \check{T_1}; T_1; \overrightarrow{B_2}; \check{T_2}; T_2; \ldots; \overrightarrow{B_n}; \check{T_n}; \overleftarrow{B_n}; \ldots; \overleftarrow{B_2}^{(n-1)}; \overleftarrow{B_1}^{(n)}$$

Table 4.2 shows an example with two tests. The adjoint variables xb, $o1b$, etc. hold the temporary variables needed to compute the $J^t(B_i) \cdot \ldots \cdot J^t(B_2) \cdot J^t(B_1) \cdot e_j / x_j$ (from equation 3) for each test T_i. Because of the vector execution, these adjoints are in fact arrays, where index i deals with the test T_i.

The program $\check{P}$ contains at least two times more blocks than the program P. The computational cost of program $\check{P}$ is proportional to n^2, with n number of test. In the next section, we explore strategies to down-size the cost.

Motivational Example	
Original Code	**Domain-Validated Code**
subroutine sub1(x,y,z,o1,o2)	subroutine sub1_dv(x,y,z,xb,yb,zb)
$B_1 \left\{ \begin{array}{l} x = x * y \\ r = x * x + y * y \end{array} \right.$	PUSH(x) $\overrightarrow{B_1} \left\{ \begin{array}{l} x = x * y \\ r = x * x + y * y \end{array} \right.$
C control flow change I $T_1 \left\{ \begin{array}{l} if\ (\ r > 10\ )\ then \\ \quad r = 10 \\ endif \end{array} \right.$	$\check{T}_1 \left\{ \begin{array}{l} t1 = r - 10 \\ t1b = 1/t1 \\ rb(1) = t1b \\ t1b = 0.0 \end{array} \right.$ $T_1 \left\{ \begin{array}{l} if\ (\ t1 > 0\ )\ then \\ \quad r = 10 \\ endif \end{array} \right.$
$B_2 \left\{ \begin{array}{l} o1 = r * z * z \\ o2 = r * r + 20 * o1 \end{array} \right.$	$\overrightarrow{B_2} \left\{ \begin{array}{l} o1 = r * z * z \\ o2 = r * r + 20 * o1 \end{array} \right.$
C control flow change II $T_2 \left\{ \begin{array}{l} if\ (\ o1 > o2 + 20\ )\ then \\ \quad o1 = o2 + 20 \\ endif \end{array} \right.$	$\check{T}_2 \left\{ \begin{array}{l} t2 = o1 - 20 - o2 \\ t2b = 1/t2 \\ o1b(1) = -t2b \\ o2b(1) = t2b \\ t2b = 0.0 \end{array} \right.$
$B_3 \left\{ \begin{array}{l} o1 = o2 - o1 \\ o2 = o1 * r \end{array} \right.$	$\overleftarrow{B_2} \left\{ \begin{array}{l} xb(2) = yb(2) = zb(2) = 0 \\ rb(2) = 2 * r * o2b(1) \\ o1b(1) = o1b(1) + 20 * o2b(1) \\ o2b(1) = 0.0 \\ rb(2) = rb(2) + z * z * o1b(1) \\ zb(2) = zb(2) + 2 * r * z * o1b(1) \\ o1b(1) = 0.0 \end{array} \right.$
end	$\overleftarrow{B_1}^2 \left\{ \begin{array}{l} xb(1) = yb(1) = zb(1) = 0 \\ do\ j = 1, 2, 1 \\ \quad xb(j) = xb(j) + 2 * x * rb(j) \\ \quad yb(j) = yb(j) + 2 * y * rb(j) \\ enddo \\ rb(1) = 0.0 \end{array} \right.$ POP(x) $\overleftarrow{B_1}^2 \left\{ \begin{array}{l} do\ j = 1, 2, 1 \\ \quad xb(j) = y * xb(j) \\ \quad yb(j) = yb(j) + x * xb(j) \\ enddo \end{array} \right.$ end

Table 4.1: Application of domain-validation method to motivational example

4.3 Practical Problems of the Proposed Model

We consider that the previous model in Section 4.2 is complete in the sense that it returns one constraint on ΔX for each test encountered during the execution of the program. However, in real situations, the number of constraints to manipulate is so large that this model is not practical. Indeed, there is one constraint for each run-time occurrence of a conditional, and this becomes unmanageably large.

The previous could be solve if we could somehow combine constraints as they come, in order to propagate just one at each time. But a constraint for a test T_i is actually of the form given by equation (3), which represents a half-space. Unfortunately the intersection of two half spaces is not a half-space in general.

We can think of three ways to address this problem of number of constraints:

- Choose a different representation for constraints, so that they could be merged. For example, one could approximate the half-space constraint by stronger one, such as, a rectangular area (in multidimentional-space). The advantage is that the intersection of hyper-rectangles is still hyper-rectangle. The drawback is that this constraint can become far stronger that what is required in reality.

  Different representations have different computational cost, this lead us to a trade-off between the accuracy of the representation and the computational cost of the manipulation (table 4.2).

Representation	Memory-Cpu Cost	Accuracy
spheres	low	very low
hyper rectangles	normal	low
polyhedra	high	good

Table 4.2: Representation of solutions, trade-offs

- Reduce the number of constraints that is being propagated. One way to do this is to consider only some conditionals in the source program, probably chosen by the user through directives. in any case, there still can be many of half-spaces to carry backwards, and we investigate an additional technique to remove some of them called “dropping constraints".

  Some constraints may be redundant. To detect the redundant constraints, we calculate an index of relevance of constraints. The index is calculated using a measure of distance from the constraint to the space of solution already computed. Consequently, we eliminate the useless ones. This strategy is inspired by the cutting-plane methods used in Optimization [Boyd04].

  Given the system of constraints:

$$A\vec{x} \leq B$$

or

$$P = \{x \mid a_i^t \vec{x} \leq b_i,\ i = 1, m\} \tag{4}$$

With m the number of constraints. We define the index of relevance like:

$$R_i = \frac{b_i - a_i^t x^*}{\sqrt{a_i^t H^{-1*} a_i}} \tag{5}$$

Let the matrix H^* be the Hessian of the barrier evaluated in the optimal point x^*. Developing the expression,

$$H^* = -\nabla^2 \sum log(b_i - a_i^t z)|_{z=x^*} \tag{6}$$

$$H^* = \sum \frac{a_i a_i^t}{(b_i - a_i^t x^*)^2} \tag{7}$$

finally the expression of P is,

$$P \subseteq \epsilon = \{z \mid (z - \vec{x}^*)^t H^* (z - \vec{x}^*) \leq m^2\} \tag{8}$$

The criterion to eliminate some of the constraints is:

1. if $R_i \geq m$ then constraint $a_i^t x \leq b_i$ is redundant.
2. The one that has greater relevance index is redundant (not general).

The drawback with previous method is to determine the optimal point x^*, can be very expensive. In fact, to determine that point can be as expensive as to calculate the index of relevance of each constraint of the system.

- Let the user select the direction of differentiation in the input space, like what happens already in the tangent mode of AD. The advantage is that there is only one constraint to propagate, because we only investigate the domain of validity along this direction. In other words, we only study the intersection of the domain of validity with the input differentiation direction. So, the constraint is just an interval, which is updated each time the execution reaches a conditional.

  Even if one wants to study the full domain of validity, this approach can be cheap because it suffices to run it for each element of the cartesian basis of the input space. This may prove to be cheaper than propagating a large number of constraints at one time.

5 Conclusions

The question of derivatives being valid only in a certain domain is a crucial problem of AD. If derivatives returned by AD are used outside their domain of validity, this can result in errors that are very hard to detect. AD tools must be able to detect this kind of situation.

We proposed a novel method to tackle the problem of non-differentiability in programs differentiated with Automatic Differentiation. The problem comes mainly from changes in the control flow. Our method computes approximate domains of the input data. In these domains, the returned derivatives are valid in the sense that when the input remain in this domain, the function is differentiable. This analysis must be done for each test executed at run-time by the program.

We illustrate our method on a small example.

The proposed method may prove to be very expensive on large applications. We present several strategies to cope with the computational cost. Some of the strategies involve the interaction with the end-user.

The implementation of the method is being under taken in the AD software TAPENADE.

References

[Boyd04] Boyd, S., Vandenberghe, L., "Localization and Cutting-plane Methods, Lecture topics and notes", *EE392o Optimization Projects*, Stanford University, Autumn Quarter 2003-2004, www.stanford.edu/class/ee392o (URL). 2004.

[Gamkrelidze87] Gamkrelidze, R.V., *Analysis II*, Encyclopaedia of Mathematical Sciences, Vol 14, Springer-Verlag, 1987.

[Griewank00] Griewank, A., *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers in applied mathematics, SIAM, pp. 251-302, 2000.

[Griewank92] Griewank, A., "Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation", *Optimization Methods and Software*, 1(1):35-54, 1992.

[Griewank95] Griewank, A., "Automatic Directional Differentiation of Nonsmooth Composite Functions", *Recent developments in Optimization / Seventh French-German Conference on Optimization*, Dijon 1994, pages 155-169. Springer Verlag, 1995.

[Hascoet04] Hascoët, L., Araya-Polo, M., "The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications", accepted paper at *AD 2004, The 4th International Conference on Automatic Differentiation*, Chicago, jul 19-23, 2004.

[Hiriart91] Hiriart-Urruty, J., Lemaréchal, C., *Convex Analysis and Minimization Algorithms I y II*, Springer-Verlag, 1991.

[Kearfott96] Kearfott, R. B., “Treating Non-Smooth Functions as Smooth Functions in Global Optimization and Nonlinear Systems Solvers", preprint of an article that appeared in *Scientific Computing and Validated Numerics*, ed. G. Alefeld and A. Frommer, Akademie Verlag, pp. 160-172, 1996.

[Kim01] Kim, J. G., Hovland, P. D., “Sensitivity Analysis and Parameter Tuning of a Sea-Ice Model", *Automatic Differentiation of Algorithms, from Simulation to Optimization*, Springer, Selected papers from AD2000, pp. 91-98, 2001.

[Klein96] Klein, W., “Comparisons of Automatic Differentiation tools in Circuit Simulation", *Computational Differentiation: Techniques, Applications and Tools*, SIAM, Philadel phia, Penn., pp. 297-307, 1996.

[Marsden88] Marsden, J.E., Tromba, A. J., *Vector Calculus*, Third Edition 1988, ed. W. H. Freeman and Co., 41 Madison Ave, New York, 1998.

[Neumaier90] Neumaier, A., *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, England, pp. 174-179, 1990.

[Restrepo98] Restrepo, J. M., Leaf G. K., Griewank, A., “Circumventing Storage Limitations in Variational Data Assimilation", *Journal on Scientific Computing*, SIAM, 1998.

[Talagrand91] Talagrand, O., “The use of adjoint equations in numerical modeling of the atmospheric circulation", *Automatic Differentiation of Algorithms: Theory, Implementation a nd Application*, SIAM, Philadelphia, Penn., pp. 169-180, 1991.

[Tropics04] Project Tropics, “A.D. Tool TAPENADE", *http://www-sop.inria.fr/tropics (URL)*, INRIA Sophia-Antipolis, France, 2004.

Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399