# Certification of Directional Derivatives Computed by Automatic Differentiation

MAURICIO ARAYA-POLO LAURENT HASCOËT

Project TROPICS
INRIA Sophia-Antipolis
2004 Route des Lucioles - BP 93, FR-06902 Sophia Antipolis.
FRANCE

mauricio.araya@sophia.inria.fr http://www-sop.inria.fr/tropics

*Abstract:* Automatic Differentiation (AD) tools assume differentiability of the function implemented by the given program. However, due to switches in the control flow, most programs are only piecewise differentiable. Thereby, sometimes the derivatives are wrong, unfortunately this fact is overlook by everyday use of AD. There exist extended models of AD that return useful generalized derivatives for some classes of piecewise differentiable functions, but there is little hope of doing so for all cases. In contrast, our goal is to evaluate, along with the derivative, the size of the differentiable neighborhood around the current input. This "safe neighborhood" is essential to use the derivatives consistently. We investigate several models to compute this neighborhood and study their complexity. We propose one model of acceptable cost. We present a first implementation and experiments made with our AD tool TAPENADE.

*Key-Words:* directional, derivative, validity, automatic, differentiation, control-flow

## 1 Introduction

The context of this work is Automatic Differentiation [1,2,3]. AD is an innovative strategy which, given a program that evaluates a function $F$, builds a new program that evaluates the derivatives of $F$. To compute the derivatives AD takes advantage of the fact that programs are a sequential set of instructions, and the instructions represents elementary functions, hence the application of the calculus rules is suitable.

There exists two fundamental modes of AD: *forward* and *reverse*. The *forward* 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. 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.

Currently, AD models do not include verification of the differentiability of the functions. Mainly the non-differentiability problem is introduced by conditional statements (test). The tests are part of that the control flow structure of the original program, this structure is preserved in the differentiated version of the original program. If the derivatives depends on the test, the resulting derivatives can be totally different even if the variation of the input values is very small, because a switch in the test can lead us to a totally different derivative instruction. Therefore, it may happens that AD returns some derivatives, that may not be valid because the original functions were not continuous differentiable in their complete dominion.

We propose a new method which validate the derivatives obtained using AD. To validate the derivatives we evaluate the interval around the input

data where no non-differentiability problem arises. Practically, this requires to analyzing each conditional statement at run-time, in order to find for which data it will switch, and propagate this information as a constraint on the input data. We also discuss the complexity of this mode and some alternatives. Finally, we develop a mode that is focused in the validity of the directional derivatives.

This paper is organized as follows: in Section 2 we give the basics concepts of AD. In Section 3 we state the problem. In Section 4 we proposed a general approach and we develop the directional validity method. In Section 5 we present the numerical result of experiments with the proposed method. Finally, we discuss the future work and the conclusions in Section 6.

# 2 A Brief Introduction to Automatic Differentiation

In this section, we present the framework of this work and then we introduce the forward mode of AD, which computes directional derivatives.
A program $P$ is a set of concatenated sequences of instructions $I_i$, when the control flow is fixed (in execution time) the program runs only one sequence of instructions. In particular (no control flow), program $P$ has the following form:

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

Each instruction $I_i$ represents an elementary function $f_i$ from the mathematical model or function $F$. In the other hand, the mathematical model $F$ is the composition of elementary functions $f_i$:

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

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

where $n$ is the dimension of the input space and $m$ is the dimension of the output space, $X$ input variables and $Y$ output variables.

Programs also includes special instructions to manage the control flow. Mainly there are two kinds of such instructions: loop cycles and conditional statements. The conditional statements or *tests* ($T_1$ in Table 2.1), are the key point in our work, because this kind of instruction represents piecewise functions, which are the source of the problem.

| | |
|---|---|
| | subroutine SUB1(x, y, o1) |
| $I_1$ | $x = y * x$ |
| $I_2$ | $o1 = x * x + y * y$ |
| $T_1$ | if ( $o1 > 190$) then |
| $I_3$ | $o1 = -o1 * o1/2$ |
| | else |
| $I_4$ | $o1 = o1 * o1 * 20$ |
| | endif |
| | end |

Table 2.1: Sample code

It is important to mention that there are some *intrinsic* instructions (instructions provided by the programming languages) underlays conditional statements (example: min, max, log, etc.), hence, introducing more tests.

## 2.1 Forward Mode of Automatic Differentiation (FMAD)

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

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

$$F' : X' \in I\!R^n \rightarrow Y' \in I\!R^{mxn}$$

$$with \ \ dY = F'(X)dX$$

where $dX$ represents the directional variation of the input values.

Every jacobian $f_i'$ times vector has a corresponding $I_i'$ instruction. Thus, differentiated program $P'$ has the following sequence of instructions:

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

From a computational point of view, the differentiated program is composed by the original instructions

necessary to compute the derivatives, plus the instructions which represent the derivatives. Also, as we can see in the Table 2.2, the differentiated program maintains the flow control structure of the given program.

| | |
|---|---|
| | subroutine SUB1_D(x, xd, y, yd, o1, o1d) |
| $I'_1$ | $xd = yd * x + y * xd$ |
| $I_1$ | $x = y * x$ |
| $I'_2$ | $o1d = 2 * x * xd + 2 * y * yd$ |
| $I_2$ | $o1 = x * x + y * y$ |
| $T_1$ | if $(o1 > 190)$ then |
| $I'_3$ | $o1d = -(o1d * o1)$ |
| $I_3$ | $o1 = -(o1 * o1/2)$ |
| | else |
| $I'_4$ | $o1d = 40 * o1d * o1$ |
| $I_4$ | $o1 = o1 * o1 * 20$ |
| | endif |
| | end |

Table 2.2: Direct differentiated code of example

Notice that in Table 2.2 the derivatives are expressed like $xd, yd, o1d$, being $xd, yd \in dX$ inputs and $o1d \in dY$ output.

## 3 The Problem, No Validated Derivatives

Sometimes the derivatives depend on the tests, then we have different derivatives depending on the switch of the test. If the test has the form "$if \ldots then \ldots else$" then we have two sets of the derivatives, each one corresponding to the branches of the test, in our program example the control flow will follow instructions $I'_3; I_3$ or $I'_4; I_4$ depending sign on the test.

The problem arise when for some input the program evaluates the sequence of instructions $I_1; I_2; T_1; I_3$, and for other slightly different input value the program evaluates $I_1; I_2; T_1; I_4$. The difference between the first and second input value may be very small, but small enough to switch the test $T_1$. The derivative instruction $I'_3$ and $I'_4$ may be totally different. So, small changes in the input values may switch the test returning completely different derivatives.

Note that more complicated forms of conditional statements just introduced more sets of derivatives but not more complexity to the problem.

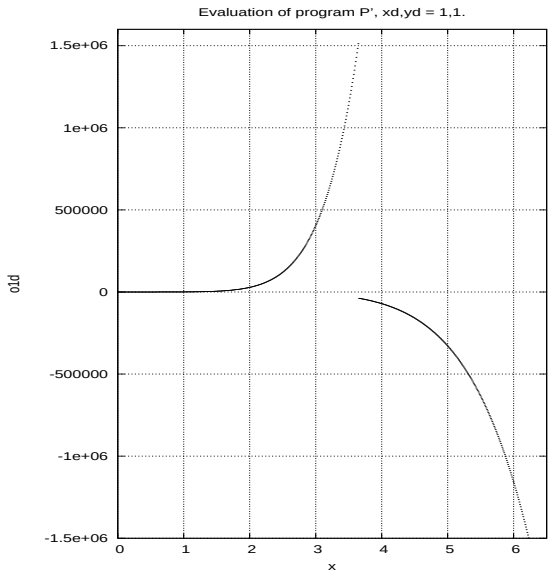


**Figure 1**: Derivatives at discontinuity on $o1 = 190$.

On the Figure 1, we can observe that the test $T_1$ introduces a discontinuity on the graph, consequently the derivatives computed by program $P'$ return totally different figures around the discontinuity.

## 4 Methods

We device a method that returns a certain interval of solutions where the derivatives are not compromise by conditional switching. To do that, we develop a formalization that relates the tests, the inputs values and the variations of the inputs variables. 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.

In this Section, we present our strategies to solve the problem through the computation of the interval of validity. In Section 4.1, we define the neighborhood of validity. In Section 4.2, we show how the pre-

vious strategy is implemented in computation model. In Section 4.3, we discuss about the neighborhood of validity. In Section 4.4, the directional strategy is presented. Finally in Section 4.5, we present the computational model for the directional strategy.

### 4.1 Neighborhood of Validity

Programs can be seen like a composition of blocks $B_i$ and tests, these blocks are sequences of elementary instructions, the tests are build up from the intermediates variables within the block before the test.

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

where $X_i$ in vector $(X_i\ T_i)$ represents the intermediates variables before the test $T_i$.

Once AD is applied to a program, the new augmented program includes the derivatives and the original control structure of the program.

$$X', X \xrightarrow{B'_1} (X_1\ T_1) \cdots \xrightarrow{B'_n} (X_n\ T_n) \xrightarrow{B'_{n+1}} Y, Y'$$

where $B'_1 = I'_2; ...; I'_i$. Let us consider a conditional $T_i$ in isolation. It uses variables from $X_i$, which depends differentially on the input $X$ (at first order) by:

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

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 $-\Delta x_j \leq x_j$, because the small change $\Delta x_j$ makes the sign of the test change. Thus, we can state the condition on $\Delta X$ upon which the program control does not switch for this test $T_i$:

$$< J(B_i) \cdot ... \cdot J(B_1) \cdot \Delta X | e_j > \geq - < X_i | e_j > \quad (1)$$

For the entire program, the computed derivatives will be valid if the variation $\Delta X$ of the input $X$ satisfies all the constraints (1) for each test $T_i$. This gives a system of constraints on $\Delta X$. The solution of this system is the neighborhood around the inputs values that returns valid derivatives.

### 4.2 Computing the Neighborhood of Validity

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 (1), and recalling that we must solve it for $\Delta X$, we must isolate $\Delta X$. A powerful way to do that is to transpose the jacobians in the dot product, yielding the equivalent equation:

$$< \Delta X \cdot J(B_1)^t \cdot ... \cdot J(B_i)^t \cdot e_j > \geq - < X_i | e_j > \quad (2)$$

The right side of the dot product is directly computed by the reverse mode of AD, which computes gradients. Unfortunately, the reverse mode is very expensive in terms of memory consumption, because requires store large number of intermediate variables.

### 4.3 Discussion on Neighborhood of Validity

We consider the model of Section 4.1 complete in the sense that it returns one constraint on $\Delta X$ for each test encountered during the execution of the program. However, in real situations, the number of tests is so large that this complete model is not practical. This section investigates strategies to reduce the cost of this model.

A first idea would be to 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 (1), which represents a half-space. Unfortunately the intersection of two half spaces is not a half-space in general.

Second idea is to reduce the size of the problem. The size of the system of equations comes from the number of constraints and inputs. The two alternatives are: to select certain constraints or/and to select certain directions of derivation.

The user can identify which tests must be analyzed. Another possibility would be to drop some constraints automatically, because some them 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 [6].

Alternately, the user can identify certain directions of the input; This allow us to simplify the constraints and focus on the relevant domain of validity. Also, we can use the forward mode of AD because the needed derivatives will be directional ones. This idea is the inspiration for the strategy in the next section.

### 4.4 Directional Validity

The model of Section 4.1 is expensive in memory consumption and run-time. We propose a new strategy which is focused in the directional derivatives. The goal is give to the user information about the validity of the derivatives in the input space regarding specific directions in the input space. Note that the repetition of the procedure for all directions returns the same information of model of Section 4.1.

The idea behind the following strategy is evaluate how much we can change the input $X$ without switch the particular test $T_i$. Thus, the size of this change defines the "safe neighborhood" where derivatives may lay.

Let us consider just one test $T_0$, this test is build upon the intermediates variables computated by $B_0$, also we have the test sign variable $t_0$ from a particular input $X$. If the test is $t_0 \geq 0$ then we have a constraint $t_i \geq 0$. Small changes in $X$ produce the variation on test ($\Delta t_0$), like $\Delta t_0 = t_i - t_0$, then because the sign of the constraint we obtained:

$$\Delta t_0 + t_0 \geq 0 \quad or \quad \Delta t_0 \geq -t_0 \tag{3}$$

To build $\Delta t_0$ we use the following expression,

$$\Delta t_0 = J(T_0) \cdot \Delta B_0 \tag{4}$$

where $\Delta B_0$ represents the variation of the intermediates variables of block $B_0$ due the input variables $X$, and $J(T_0)$ is the jacobian matrix of the test $T_0$. Then we need $\Delta B_0$, which has the following form:

$$\Delta B_0 = J(B_0) \cdot \beta \cdot \dot{X} \tag{5}$$

where $\dot{X}$ represents the normalized directional variation of the input and $\beta$ is the scalar that hold the magnitude of the this variation. Replacing expressions (4) and (5) on expression (3) we obtain:

$$J(T_0) \cdot J(B_0) \cdot \beta \cdot \dot{X} \geq -t_0$$

$$\beta \geq \frac{-t_0}{J(T_0) \cdot J(B_0) \cdot \dot{X}} \tag{6}$$

Expression (6) satisfy the constraint over the test, and $\beta$ store the information about how much the input variation can increase following the given direction $\dot{X}$ of the input space.

To compute the directional dominion of validity around the input $X$, we repeat the computation of $\beta$ before every test in the program, thus, updating the value of $\beta$.

### 4.5 Computing the Directional Validity

To implement the directional method we insert an instruction before every test of the program $P'$, this instruction computes the value $\beta$ for the test and update the global value $\beta$ of the program.

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

$$\check{P} = B'_1; V_1; T_1; \ldots; B'_n; V_n; T_n; B'_{n+1}$$

where $V_i$ is the instruction that computes the value of $\beta$. The values of $\beta$ are propagated forward and updated for every instruction $V_i$. To the end, we obtained the value of $\beta$ which holds the information of the whole program $\check{P}$; The interval of validity ("safe neighborhood") is build up from the $\beta$ value.

The next section presents some experiments using our new model.

## 5 Experimental Results

In this Section, we show how this new mode works and how the results are expressed. In Section 5.1, we presents the numerical results for the example already given in Section 2.2. In Section 5.2, we applied the new mode over two real-life programs.

### 5.1 Basic Example

The interval of validity has the following form: $[gmin, gmax]$, where $gmin$ represents the maximum size of the derivatives in opposition to the given direction, conversely, the $gmax$ is the maximum size of the derivative following the given direction.

In Table 5.1, we can see how the interval change around the critical point (where the test switch). Before the switch, the interval is $[n.p., 0.005]$, which means that derivate is close to a discontinuity in the given direction, but in the opposite direction is n.p., where n.p. means that there is no differential problem. Conversely, after the test, the interval is $[0.004, n.p.]$, because the critical point lay behind the evaluated point in the given direction.

| **x** | **y** | **o1d** | **gmin** | **gmax** |
|---|---|---|---|---|
| 3.62 | 3.62 | 1456628.2 | n.p. | 0.026 |
| 3.63 | 3.63 | 1484149.2 | n.p. | 0.016 |
| 3.64 | 3.64 | 1512117.1 | n.p. | 0.005 |
| 3.65 | 3.65 | -38513.4 | 0.004 | n.p. |
| 3.66 | 3.66 | -39235.4 | 0.014 | n.p. |
| 3.67 | 3.67 | -39969.0 | 0.023 | n.p. |

Table 5.1: Results from validated code of the example with direction (xd,yd) = (1,1).

### 5.2 Real-life Examples

We conduct a large number of tests on real-life programs, with satisfactory results, even when the mathematical model was not completely suitable.

| program | lines code | # tests | # validated tests |
|---|---|---|---|
| STICS | 21.163 | 2.682 | 542 |
| CEA | 19.789 | 1.864 | 189 |

Table 5.2: Real-life programs settings.

The results require close analysis from the end-user in order to be useful, but from our point view, the results are promising in the sense that they are consistent with the predicted behavior.

## 6 Conclusion

We proposed a novel method to tackle the problem of non-differentiability in programs differentiated with Automatic Differentiation. The method computes intervals following a given direction of input data. In these intervals, the returned derivatives have no problem of differentiability.

The computational cost of the new mode is marginal (3%) with respect the computational cost of the forward mode.

This 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. The method we proposed is one possible way to warn the user from abusive use of the derivatives.

The future work is expand the model of validation to the reverse mode of AD.

*Acknowledgement* The present work has been partially supported by CONICYT-INRIA Sophia-Antipolis cooperation agreement.

*References:*

[1] Berz, M., Bischof, G., Corliss, G., and Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, PA, 1996.

[2] Corliss, G., Faure, Ch., Griewank, A., Hascoët, L., and Naumann, U. *Automatic Differentiation of Algorithms, from Simulation to Optimization*, Springer, Selected papers from AD2000, 2001.

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

[4] Hascoët, L., Pascual, V., *TAPENADE 2.1 User's guide*. Technical report #224. INRIA, 2004.

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

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