Adjoining Strategies for Multi-Layered Programs

Christèle Faure

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Adjoining strategies for Multi-layered Programs

Christèle Faure

N° 3781

Octobre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3781--FR+ENG

SOPHIA ANTIPOLIS

Adjoining strategies for Multi-layered Programs

Christèle Faure*

Thème 1 — Réseaux et systèmes
Projet TROPICS

Rapport de recherche n° 3781 — Octobre 1999 — 24 pages

Abstract: Several papers have presented the rules to apply to a straight line program to differentiate it in direct or reverse mode. In this paper, we first recall these rules, and we try to specify the different possible strategies for the differentiation of a multi-level program in direct or reverse mode. The strategy to apply in direct mode is straight forward and can be directly extended from the straight line case. For the reverse mode, the computation of derivatives in reverse order (ie. the computation of initial variables) makes the problem much more complicated. We show that, a lot of strategies can be applied, between storing recursively the variables and recomputing them all from the initial point. In order to make the comparison of those strategies possible, we show the complexities in terms of memory requirement and execution time.

The first section describes basics of automatic differentiation, the second one contains some notations and the graphical representation of a program we are using in the rest of the paper. The third section is dedicated to the description of different strategies applicable on a real code.

Key-words: Automatic Differentiation, Computational Differentiation, Complexity, Adjoint code, Reverse mode, Forward mode.

* Email : Christele.Faure@sophia.inria.fr, URL : http://www.inria.fr/tropics/Christele.Faure

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Stratégies hiérarchiques de génération de codes adjoints

Résumé : De nombreux articles ont décrit les règles de transformation à appliquer à un programme sans boucles pour le dériver automatiquement en mode direct ou inverse. Dans ce rapport, nous rappelons ces règles, mais tentons de spécifier les différentes stratégies possibles pour dériver automatiquement un code multi-niveau en mode direct ou inverse. La stratégie à utiliser en mode direct pour calculer uen dérivée directionnelle peut être directement extraite du cas des programmes sans boucles. Pour le mode inverse, le fait que les variables adjointes soient calculées dans l'ordre inverse des variables originales rend le problème plus compliqué. Nous montrons que de nombreuses stratégies peuvent être appliquées entre mémoriser récursivement toutes les variables de la trajectoires et recalculer toutes ces variables intermédiaires à partir des variables initiales. Pour pouvoir comparer ces stratégies, nous calculons la compléxité du code généré en terme d'encombrement mémoire et de temps d'exécution.

Mots-clés : Differentiation automatique, complexité, adjoint discret, mode inverse, mode direct

Chapter 1

Motivation

Automatic Differentiation ([12, 2]) is a set of techniques for computing derivatives at arbitrary points. Two modes of Automatic Differentiation have been studied: the direct (or forward) mode that computes the derivatives and the initial values simultaneously, and the reverse (or backward) mode that computes first the initial values and then the derivatives in reverse order. The reverse mode is particularly efficient for computing gradients because its cost is independent of the number of inputs. Two classes of automatic differentiation Tool exist: those that work by code generation, and those that work by operator overloading. `Odyssée`, `Adifor`, `GRESS`, `TAMC` belong to the first class of automatic differentiation tools based on code generation, whereas `Adolc` belongs to the second one.

In this paper, we describe the main approaches used to make the Reverse Mode of automatic differentiation applicable on real world codes. Some of those techniques have been implemented in automatic differentiation tools (`Tamc`, `Odyssée`), but some are only used to write discrete adjoint codes by hand. For example, the "No recomputation" approach can only be used for hand written discrete adjoint because it requires a great deal of optimization to be applied on a real code.

In the first section, we describe the basics of Automatic differentiation and how to apply those principles on straight line codes. In the second section we present the notations used in this paper and graphical representation of programs. The third section shows the different strategies one can think of to generate the cotangent code of a program.

Chapter 2

Principles of Automatic differentiation

Automatic (or Computational) Differentiation is based on two observations. First, any instruction executed on a computer can be seen as an elementary function using simple operations, a program is then a composition of those elementary functions. Second, a program can be differentiated as a composition of functions using the chain rule.

Using Computational Differentiation two kinds of derivative codes can be generated: a code for computing the product of the Jacobian matrix by one (or more) direction(s), which is called the tangent code, or a code for computing the product of the transposed Jacobian matrix with some dual directions which is called the cotangent code or adjoint code.

In this section we will explain how those principles are used to generate tangent codes or cotangent codes. First we will show how to deal with a single instruction, and then with a straight line program.

2.1 Differentiation of one instruction

The differentiation of one assignment can be deduced from the first principle given in the introduction of this section.

For example, the assignment A shown in figure 2.1(a) can be seen as a mathematical elementary function f shown in figure 2.1(b). The mathematical input and output domains of f are $\mathbb{R}^2 \rightarrow \mathbb{R}$.
To be able to built up the composition of the elementary functions corresponding to a sequence of instructions, one has to extend its input and output domains. The definition of $\mathcal{F}$ shows in figure 2.1(c) the extension of f to $\mathbb{R}^3 \rightarrow \mathbb{R}^3$. This leads to consider all the variables as potential input and output.

Using the definition $\mathcal{F}$ of the code A, the product of the Jacobian matrix of A by some direction $(dX, dY, dZ)_i$ can be easily built. The figure 2.2 shows two equivalent representations of this computation in mathematical notation: figure 2.2(a) shows the matrix computation and figure 2.2(b) shows the equivalent scalar computation (where $_{i,o}$ indicate input and output respectively):

The product of the transposed Jacobian matrix of $\mathcal{F}$ by the direction in the dual space $(dX^\star, dY^\star, dZ^\star)_i$ can also be easily built. The figure 2.3 shows two equivalent representations of this computation in mathematical notation (where $_{i,o}$ indicate input and output, respectively) :

```
Z = X * Y**2
```

(a) code A

$$\begin{array}{rcl} \mathbb{R}^2 & \rightarrow & \mathbb{R} \\ (x,y) & \rightarrow & x * y^2 \end{array}$$

(b) f

$$\begin{array}{rcl} \mathbb{R}^3 & \rightarrow & \mathbb{R}^3 \\ (x,y,z) & \rightarrow & (x, y, x * y^2) \end{array}$$

(c) $\mathcal{F}$

Figure 2.1: An assignment A seen as an elementary function $\mathcal{F}$

$$\begin{pmatrix} dX \\ dY \\ dZ \end{pmatrix}_o := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Y^2 & 2XY & 0 \end{pmatrix} \begin{pmatrix} dX \\ dY \\ dZ \end{pmatrix}_i \qquad \begin{aligned} dX_o &:= dX_i \\ dY_o &:= dY_i \\ dZ_o &:= Y^2 dX_i + 2XY dY_i \end{aligned}$$

(a) (b)

Figure 2.2: Product of the Jacobian matrix of A by the direction $(dX, dY, dZ)_i$

$$\begin{pmatrix} dX^\star \\ dY^\star \\ dZ^\star \end{pmatrix}_o := \begin{pmatrix} 1 & 0 & Y^2 \\ 0 & 1 & 2XY \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} dX^\star \\ dY^\star \\ dZ^\star \end{pmatrix}_i \qquad \begin{aligned} dX^\star_o &:= dX^\star_i + Y^2 dZ^\star_i \\ dY^\star_o &:= dY^\star_i + 2XY dZ^\star_i \\ dZ^\star_o &:= 0. \end{aligned}$$

(a) (b)

Figure 2.3: Product of the transposed Jacobian matrix of A by the dual "direction" $(dX^\star, dY^\star, dZ^\star)_i$

```
                                    dX = dX + Y**2*dZ
                                    dY = dY + 2*X*Y*dZ
dZ = Y**2 * dX + 2*X*Y*dY           dZ = 0.
```

(a) $J_{\mathcal{F}} * d$ (b) $J^\top_{\mathcal{F}} * d^\star$

Figure 2.4: Derivatives of the assignment A

From the two scalar computations shown in figure 2.2(b) and figure 2.3(b), one can simply get the corresponding tangent and cotangent straight line codes. A mathematical variable can be set but never modified which is why we have introduced the $_i$,$_o$ postfixes, but on a computer, variables are memory locations and can be modified. In order to optimize the code in term of the number of intermediate variables, the X_i and X_o variables are identified and denoted X. Therefore, the instructions $dX_o = dX_i$ are transformed in $dX = dX$ and can be discarded. The figures 2.4(a) and 2.4(b) show respectively the optimized tangent code and the optimized cotangent code of A.

One must notice that the Jacobian matrices must be evaluated on the correct input, so in the context of a real program the computation of the point performed by the initial code must be reproduced before the computation of the Jacobian.

2.2 Differentiation of a straight line code without re-assignment

We have said in the previous sections that one may consider any instruction of a straight line program as an elementary function and a straight line program as a composition of functions. In this section, we will show how to use the chain rule to generate the tangent and cotangent codes of a straight line.

First we recall that by the chain rule, if f is the composition of n elementary functions $f = f_n \circ \cdots \circ f_2 \circ f_1$, its Jacobian matrix J is the product of the n elementary Jacobian matrices $J = J_n \cdots J_2 J_1$ where J_i denotes $\mathrm{J}_{(f_{i-1} \circ \ldots \circ f_1)(x)}(f_i)$ the Jacobian matrix of f_i computed at the point $(f_{i-1} \circ \ldots \circ f_1)(x)$. The transposed Jacobian matrix of f dented by $J^\top$ is the product in reverse order of the n transposed elementary Jacobian matrices: $J^\top = J_1^\top J_2^\top \cdots J_n^\top$ where J_i denotes the $\mathrm{J}_{(f_{i-1} \circ \ldots \circ f_1)(x)}(f_i)$.

The figure 2.5(a) shows a toy straight line program G of two instructions without re-assignment of variable. The mathematical function $\mathcal{G}$ implemented in G can be seen as the composition of the two elementary

```
Z = X * Y**2
W = Z**2 * Y
```

(a) G

$$\begin{array}{lcl} \mathbb{R}^4 & \rightarrow & \mathbb{R}^4 \\ (x,y,z,w) & \rightarrow & (x,y,x*y^2,w) \end{array}$$

(b) $\mathcal{G}_1$

$$\begin{array}{lcl} \mathbb{R}^4 & \rightarrow & \mathbb{R}^4 \\ (x,y,z,w) & \rightarrow & (x,y,z,y*z^2) \end{array}$$

(c) $\mathcal{G}_2$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ Y^2 & 2XY & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(d) $J_{\mathcal{G}_1}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & Z^2 & 2YZ & 0 \end{pmatrix}$$

(e) $J_{\mathcal{G}_2}$

Figure 2.5: Applying the first principle on G

```
dZ = Y**2 * dX + 2*X*Y * dY
```

(a) $d_o := J_{\mathcal{G}_1} d_i$

```
dW = Z**2 * dY + 2*Y*Z * dZ
```

(b) $d_o := J_{\mathcal{G}_2} d_i$

```
dX = dX + Y**2 * dZ
dY = dY + 2*X*Y * dZ
dZ = 0.
```

(c) $d_o^\star := J_{\mathcal{G}_1}^\top d_i^\star$

```
dX = dX + Z**2 * dW
dY = dY + Z**2 * dW
dZ = dZ + 2*Y*Z * dW
dW = 0.
```

(d) $d_o^\star := J_{\mathcal{G}_2}^\top d_i^\star$

Figure 2.6: Derivatives of G_1, G_2

functions $\mathcal{G}_1, \mathcal{G}_2$ (see figures 2.5(b),2.5(c)) $G = \mathcal{G}_2 \circ \mathcal{G}_1$. As shown in the previous section, it is easy to get the two elementary Jacobian matrices $J_{\mathcal{G}_1}, J_{\mathcal{G}_2}$ (see figures 2.5(d),2.5(e)) of $\mathcal{G}_1, \mathcal{G}_2$.

We denote $J = J_{\mathcal{G}_2} J_{\mathcal{G}_1}$ the Jacobian matrix of $\mathcal{G}$ and d ($d^\star$) the direction (respectively dual direction). The generation of tangent and cotangent codes of G_1, G_2 can be performed as described in the previous section and lead to the code shown in figure 2.6.

Now, we want to built the product of the two Jacobian vector products using the chain rule. From the chain rule, we know that the composition of functions leads to the product of matrices. We know also that we want to generate a code that computes only Jacobian vector products, so we generate a code that splits $d_o := J_2 J_1 d_i$ within two Jacobian vector products $d_1 := J_1 d_i$ and thereafter $d_o := J_2 d_1$.

The last problem is to generate a code that computes J_1, J_2 at the correct points i_0, i_1 where i_0 is the point before the call of $\mathcal{G}_1$ and $i_1 = \mathcal{G}_1(i_0)$. In this section we show how to deal with straight line codes without re-assignment, so i_0 is not modified by the computation of G and i_1 is computed by G_1 but not modified by g_2. Therefore, one can before compute i_0, i_1 before the computation of the adjoint derivatives. The computation of $i_2 = \mathcal{G}_1(i_1)$ is not necessary but cannot lead to errors, so one can add after the straight line program G the new straight line program $G' = d_1 := J_1 d_i; d_o := J_2 d_1$ as shown in figure 2.7.

```
Z = X * Y**2
W = Z**2 * Y

dZ = Y**2 * dX + 2*X*Y * dY
dW = Z**2 * dY + 2*Y*Z * dZ
```

(a) $J_{\mathcal{G}} * d$

```
Z = X * Y**2
W = Z**2 * Y

dX = dX + Z**2 * dW
dY = dY + Z**2 * dW
dZ = dZ + 2*Y*Z * dW
dW = 0.

dX = dX + Y**2 * dZ
dY = dY + 2*X*Y * dZ
dZ = 0.
```

(b) $J_{\mathcal{G}}^{\top} * d^{\star}$

Figure 2.7: Derivatives of G

```
Y = X * Y**2
Y = Y**3 * X**2
```

(a) $\mathcal{H}$

$$\begin{array}{ccc} \mathbb{R}^2 & \rightarrow & \mathbb{R}^2 \\ (x,y) & \rightarrow & (x, x*y^2) \end{array}$$

(b) $\mathcal{H}_1$

$$\begin{array}{ccc} \mathbb{R}^2 & \rightarrow & \mathbb{R}^2 \\ (x,y) & \rightarrow & (x, y^3*x^2) \end{array}$$

(c) $\mathcal{H}_2$

$$\begin{pmatrix} 1 & 0 \\ Y^2 & 2XY \end{pmatrix}$$

(d) $J_{\mathcal{H}_1}$

$$\begin{pmatrix} 1 & 0 \\ 2XY^3 & 3X^2Y^2 \end{pmatrix}$$

(e) $J_{\mathcal{H}_2}$

Figure 2.8: Applying the first principle on $\mathcal{H}$

2.3 Differentiation of a straight line with re-assignment

In this section we will show how to differentiate a straight line code with some re-assignment of variables.

If we call $J_{\mathcal{H}}$ the Jacobian matrix of $\mathcal{H}$, $J_{\mathcal{H}} * d$ and $J_{\mathcal{H}}^{\top} * d^{\star}$ are computed using the chain rule in the same way as for the previous example (see figure 2.7). But in this example, the variable `Y` is assigned twice and that makes a big difference in the way the resulting code is written.

We call (X_0, Y_0) the initial value of the point `X,Y`, and (X_1, Y_1) its value after the first assignment $(X_1, Y_1) = (X_0, X_0 * Y_0 * *2)$. In this example, $J_{\mathcal{H}_1}$ must be evaluated on the point (X_0, Y_0) and $J_{\mathcal{H}_2}$ must be evaluated on (X_1, Y_1).

In the tangent code, we insert the instruction that computes (X_1, Y_1) before the computation of $J_{\mathcal{H}_2} * d$ but after the computation of $J_{\mathcal{H}_1} * d$. In order to get a general method for getting the tangent code, we chose to insert the initial instruction after the derivative instruction. The tangent code of $\mathcal{H}$ using this rule is shown in figure 2.9(a). One can notice that for this example the last instruction `Y=Y**3 * X**2` is useless for the computation of $J_{\mathcal{H}} * d$.

```
dY = Y**2 * dX + 2*X*Y * dY
Y = X*Y**2

dY = 2XY**3 * dX + 3Y**2X**2 * dY
Y = Y**3*X**2
```

(a) $J_{\mathcal{H}} * d$

```
S0 = Y
Y = X*Y**2
S1 = Y
Y = Y**3*X**2

Y = S1
dX = dX + 2XY**3 * dY
dY =   3Y**2X**2 * dY

Y = S0
dX = dX + Y**2 * dY
dY =       2YX * dY
```

(b) $J_{\mathcal{H}}^{\top} * d^{\star}$

```
YY = Y
XX = X

Y = X*Y**2
Y = Y**3*X**2

Y = YY
X = XX
Y = X*Y**2
dX = dX + 2XY**3 * dY
dY =   3Y**2X**2 * dY

Y = YY
X = XX
dX = dX + Y**2 * dY
dY =       2YX * dY
```

(c) $J_{\mathcal{H}}^{\top} * d^{\star}$

Figure 2.9: Differentiation of $\mathcal{H}$

In the cotangent code, the first Jacobian matrix to be evaluated is $J_{\mathcal{H}_2}^{\top} * d^{\star}$ and the second is $J_{\mathcal{H}_1}^{\top} * d^{\star}$. The code to be written has then to compute (X_1, Y_1) before $J_{\mathcal{H}_2}^{\top} * d^{\star}$. But `(X,Y)` has also to be reset to its initial value (X_0, Y_0) before evaluating $J_{\mathcal{H}_1}^{\top} * d^{\star}$.

To do so, one can chose to store the value[1] of `Y` before the first assignment in the intermediate variable `S` and to restore this value before the computation of $J_{\mathcal{H}_1}^{\top} * d^{\star}$. In order to get a general method for getting the cotangent code, we chose to store any computed variable in the code of $\mathcal{H}$ and to restore it before each Jacobian matrix evaluation. The cotangent code of $\mathcal{H}$ using this store/restore rule is shown in figure 2.9(b). One can notice that for this example the three instructions `Y=Y**3 * X**2` and the `S1=Y,Y=S1` are useless for the computation of $J_{\mathcal{H}} * d^{\star}$.

The other method to compute $J_{\mathcal{H}_2}^{\top} * d^{\star}$ and $J_{\mathcal{H}_1}^{\top} * d^{\star}$ at the correct points would be to recompute the correct value of `X,Y` from their initial value (X_0, Y_0) instead of storing/restoring the intermediate results. In order to show the general method for getting the cotangent code, we have chosen to save the values of the point before the computation of $\mathcal{H}$ even if some components are not modified. The cotangent code of $\mathcal{H}$ using this recomputation rule is shown in figure 2.9(c). One can notice that for this example the four instructions `Y=Y**3*X**2,Y=XX*YY**2` and the `XX=X,X=XX` are useless for the computation of $J_{\mathcal{H}} * d^{\star}$.

On this toy example, we have shown that getting an efficient tangent code is easily possible even though it is error prone to write it by hand. But we have also shown clearly the difficulties of getting an efficient cotangent code even on this small example. In this case, using general rules without optimization lead to a lot of useless computations and/or extra memory requirement.

2.4 Conclusion

In the previous sections, the basic ideas of automatic differentiation have been recalled. We have shown how to differentiate straight line programs, but in a "real world" code there is a lot of complex instructions `do loops, branches, subroutine calls ...`

One can convince himself that even for complex instructions, the tangent code is easy to derive from the previous section. The cotangent code seems clearly much more difficult to apply on complex instructions and there is a lot of possible choices between the two basic solutions: all the variables modified by the initial

[1] One can also think of storing the partial derivatives (see [10]), but this strategy is not described in this paper.

function are stored (shown in figure 2.9(b)), and all the necessary variables are recomputed from their input values (shown in figure 2.9(c)).

Different strategies for dealing with the real world original functions have been used when writing adjoint codes by hand, or generating the derivative with Automatic Differentiation Tools. In the next sections we try to clarify and compare those different approaches.

Chapter 3

Description of a program

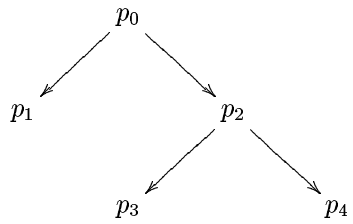
Let us look at the differentiation of a program where all the routines are straight line programs with calls of other routines. One can think of a straight line code where common sequence of instructions are represented by routine calls. In order to describe clearly the different possible combinations (storage/recomputation) on a whole program we need to describe a program at runtime, but we also need some properties to evaluate the complexity. In this section we introduce the different terms we will use all along this paper.

3.1 Representations of a program

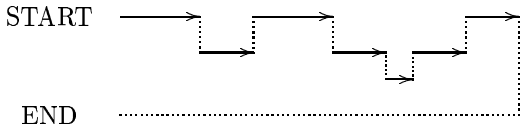
In this paper we will use two descriptions of a program: one is the **call tree** and the second is the **execution path**. Both representations of a program are known at run time which means that several couples call tree/execution path may be associated with each program depending on its inputs. In the call tree, each node represents a sub-program of the source and each arrow links a routine with a routine it actually calls. If a routine is called twice in the program we replicate the corresponding nodes. The execution path shows the different pieces of code really used during one execution of a program.

For example, the figure 3.1(a) shows the call tree of one execution E of a program P made of five sub-programs $p_0, \cdots p_3$. To read this call tree one must start from p_0 and see that it "may call" first p_1 then p_2 and that p_2 calls p_3. The figure 3.1(b) the execution path of E. To read the execution path, one must follow the arrow that goes from the label $START$ to the label END. The doted line stands for calling and returning from the sub-program.

In the call tree shown figure 3.1(a), one can see that each sub-program is represented by it's name whereas in the execution path (shown in figure 3.1(b)) it is split into line segments representing pieces of code. For example, the code of p_0 is split into three parts, one before the call to p_1, one between the call to p_1 and p_2



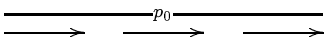
(a) Call tree of P



(b) Execution path E of P

Figure 3.1: Descriptions of P

and one after the call to p_2. The code of p_2 is also split into two pieces due to the call to p_3. In the execution path those parts appear clearly as:



3.2 Complexity of a program

Let P be a program, and $\{p_i, i = 0, N\}$ the set of its sub-programs. As we have seen before, each program P is associated to several execution paths depending on its inputs. We denote E the trace of one execution of P (sequence of sub-programs executed) and B the list of branches of the call tree associated to the same execution. We denote d_i the depth of p_i in the call tree of one execution of the program P. We denote t_i the execution time of p_i and m_i the memory (dynamic or static) required for the execution of p_i without its children in the call tree. Those data (d_i, m_i, t_i) depend on the execution of P because they can be dynamically adjusted.

In our example described in figure 3.1, the depth of all the subprograms of P $\{d_i, i = 0, 3\}$ are $d_0 = 0, d_1 = 1, d_2 = 1, d_3 = 2$. The execution path $E = p_0; p_1; p_2; p_3$ and the list of branches of the corresponding call tree is $B = p_0; p_1; p_0; p_2; p_3$. For the execution path E of P, the associated local memory is:

$$M = \max(m_0 + m_1, m_0 + m_2 + m_3) = \max_{b \in B} \sum_{i \in b} m_i$$

and the execution time is:

$$T = t_0 + t_1 + t_2 + t_3 = \sum_{i \in E} t_i.$$

To get the complexity of P instead of the complexity of one execution E_j of P, one must maximize those results over all the possible executions (let say K). But then if an infinite execution of P is possible, execution time and memory requirement could be infinite. For each execution j of P the associated execution time is $T = \max_{j \in K} \sum_{i \in E_j} t_i$ and the local memory required is $M = \max_{j \in K} \max_{b \in B_j} \sum_{i \in b} m_i$.

3.3 Components of the derivative code

In the previous section, we have shown various strategies for the generation of the cotangent code of a straight line code. In order to describe those strategies on a whole program in a coherent manner, we have chosen to introduce some concepts with the corresponding notations.

First, we separate the two different components of the cotangent code of a straight line code. Figure 3.2 shows the two components of the cotangent code of $\mathcal{H}$ (see source code in 2.8(a)) using two strategies; $\mathcal{H}_1^s$ and ${\mathcal{H}_1'}^r$ are the two components of the cotangent code of $\mathcal{H}$ using the "all storage strategy", $\mathcal{H}_2^s$ and ${\mathcal{H}_2'}^r$ are the two components of the cotangent code using the "all recomputation strategy". You can see from this toy example, that using the "all storage strategy" leads to the storage of 3 variables whereas the "all recomputation strategy" leads to the storage of only 2 variables and could be optimized to store only one variable Y as X is not modified.

The forward component computes the initial function and stores (some or all) the initial values, and the backward component restores (some or all) the initial values and computes the derivatives. Those two components can be written using several storage/recomputation strategies, the only constraint is that if the context of execution of the initial function is restored, then the execution of a forward component followed by the execution of the backward component of a function computes the cotangent values of this function. Those components must share only the tape where the values of the variables are stored (`push`) and retrieved (`get, pop`).

Now we generalize this notion of component to a subprogram in order to explain the different strategies that can be applied at the call tree level to get the cotangent code of a program. We decide to treat a routine as a whole which means that each routine is associated with one forward component and one reverse component. If p_i is the name of a sub-program, we denote p_i^s the forward component of its cotangent code and ${p_i'}^r$ the reverse component. We denote m_i' the memory necessary for the storage of the derivatives. For each variable in p_i there is at most one derivative variable associated, so

$$\forall i \quad m_i' \leq m_i.$$

```
push (Y)
Y = X*Y**2
push (Y)
Y = Y**3*X**2
push (Y)
Y = XY
```

(a) $\mathcal{H}_1^s$

```
pop (Y)
dX = dX + Y * dY
dY =      X * dY

pop (Y)
dX = dX + 2XY**3 * dY
dY =   3Y**2X**2 * dY

pop (Y)
dX = dX + Y**2 * dY
dY =      2YX * dY
```

(b) $\mathcal{H}_1'^r$

```
push (Y)
push (X)
Y = X*Y**2
Y = Y**3*X**2
```

(c) $\mathcal{H}_2^s$

```
get (Y)
pop (X)
Y  = X*Y**2
Y  = Y**3*X**2
dX = dX + Y * dY
dY =      X * dY

get (Y)
Y  = X*Y**2
dX = dX + 2XY**3 * dY
dY =    3Y**2X**2 * dY

pop (Y)
dX = dX + Y**2 * dY
dY =      2YX * dY
```

(d) $\mathcal{H}_2'^r$

Figure 3.2: $\mathcal{H}^s$ and $\mathcal{H}'^r$

We denote t_i^s the execution time of p_i^s and $t_i'^r$ the execution time of $p_i'^r$. In this section, we consider the execution time due to the storage or the retrieve of the variables computed by p_i to be zero. Under this hypotheses, we get that ${t_i}^s = t_i$ and ${t_i'}^r = t_i'$, on an other hand we know that theoretically [15, 1] the ratio between the execution time of the function and one gradient with respect to the execution time of the function is lower than 5, we get then:

$$\forall i \quad {t_i}^s + {t_i'}^r \leq 5 * t_i.$$

In the example P the cotangent code of the execution $E = p_0, p_1, p_2, p_3$ should execute the reverse parts in reverse order: ${t_3'}^r; {t_2'}^r; {t_1'}^r; {t_0'}^r$. The only constraint on the sequence of execution of the forward parts wrt. reverse part is that each forward part ${t_i}^s$ must be executed before the corresponding reverse part ${t_i'}^r$. In the following section we will show different strategies to generate the cotangent code and the corresponding complexities for computing the derivatives.

Chapter 4

Cotangent code of a program

In this section, we compute the complexity in execution time and memory requirement of the cotangent code of one execution path E of a program P. In order to extrapolate the complexities of one execution path of P to all execution paths of a program P one has to compute the maximum over all the possible executions of P.

4.1 Extreme strategies

The first idea for getting a cotangent code from a program is to consider this program globally and to apply the "all storage" or the "all recomputation" strategy. We call those strategies "extreme" because they do not do any compromise between storage and recomputation.

Using the "all recomputation" strategy, the computation of the point where to run each sequence $p_i^s; {p_i}'^r$ is performed by running again the path from p_0 to the call of p_i in the initial execution path.

We call $P'r$ the adjoint code generated from P using the "all recomputation" strategy. The call tree and execution path of $P'r$ are shown in figures 4.1(a) and 4.1(b) respectively. The figure 4.1(b) shows the execution path of the derivative where $\bullet$ means storage of the context of the initial data and $\circ$ means the retrieve of this context.

Using "all recomputation" strategy, the storage of the modified variables is performed recursively on each branch, and the execution of ${p_i}'^r$ follows directly the one of p_i^s. If we denote c_0 the input context of p_0, l_i the storage of all the variables computed by p_i, the storage necessary for the computation of the derivative is:

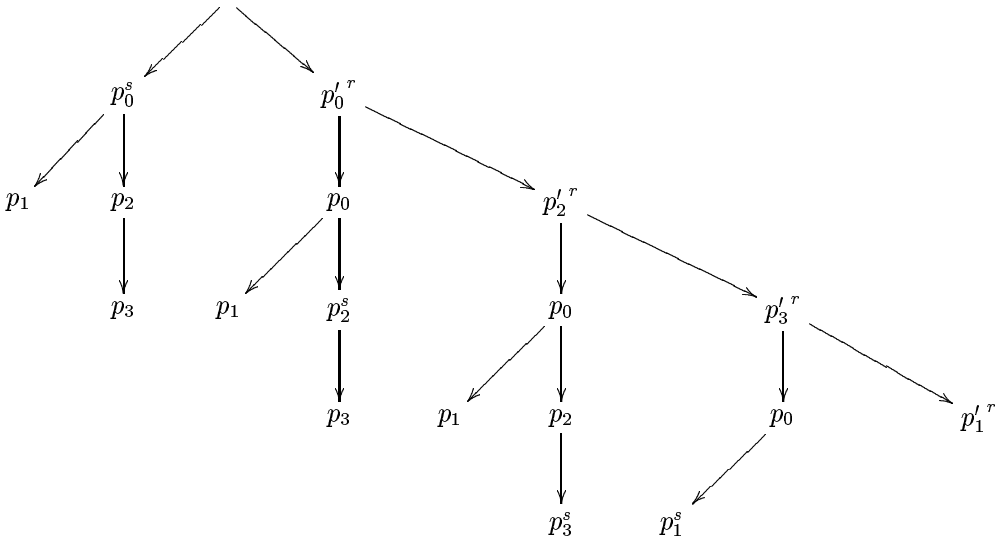
$$\begin{array}{rcl} M' & = & c_0 + \max_{b\in B}(\sum_{i\in b} m_i + l_i + m_i') \\ & \leq & 2 * M + c_0 + \max_{b\in B} \sum_{i\in b} l_i \end{array} .$$

Even if the memory required is minimal, it is easy to figure out that the growth in terms of execution time is quadratic in the number of routines executed in E. The exact cost in execution time is uneasy to formulate because in this strategy the recomputation of each routine p_i is split. For example p_0 is recalled three times but two times only the two first parts are necessary and the third one only the first part is necessary. We chose to over approximate this cost in terms of total recomputation of routines, in the example the extra cost in terms of recomputation of p_0 is $3 * t_0$.

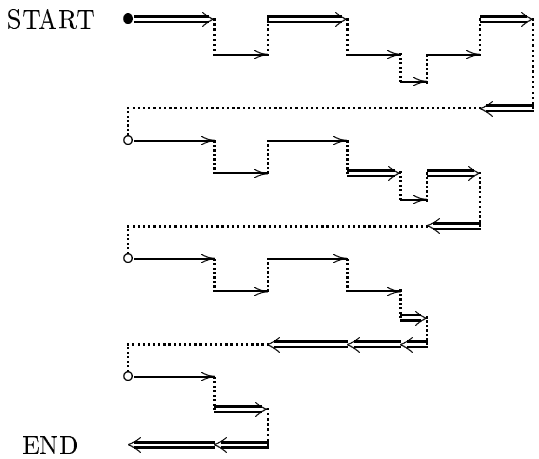
$$\begin{array}{rcl} T' & \leq & \sum_{i\in E} {t_i}^s + {t_i'}^r + \sum_{j<i} t_j \\ & \leq & 5 * T + \sum_{i\in E} \sum_{j<i} t_j \end{array} .$$

This strategy has not been implemented in any automatic tool, nor applied in hand written discrete adjoints.

Using the "all storage" strategy, the execution of the cotangent code $P's$ of P is then made of the execution of the forward component of $P's$ followed by the reverse component of $P's$. This means storing globally all the values modified by the initial program P all along the execution path of the forward component recursively (through `calls`) and then to restore those values all along the execution path of the reverse component of $P's$. The cotangent code generated using this strategy is similar to what is called adjoint code

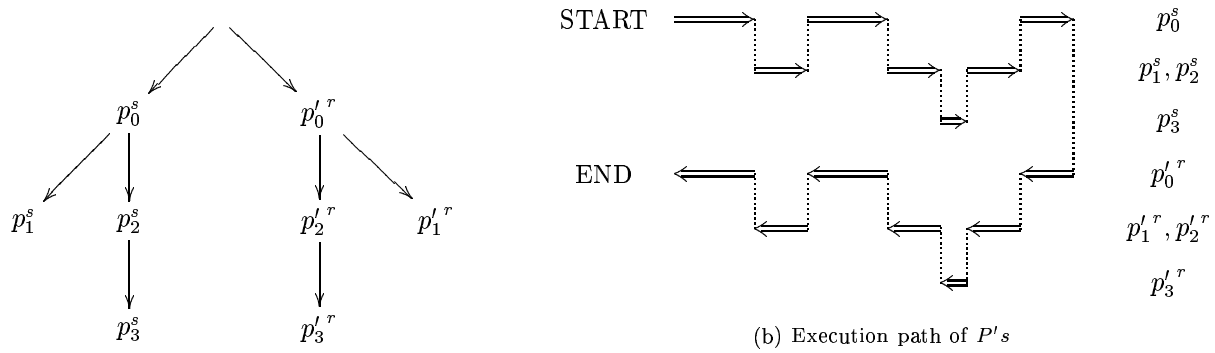


(a) Call tree of $P'r$



(b) Execution path of $P'r$

Figure 4.1: All recomputation strategy



(a) Call tree of $P's$

(b) Execution path of $P's$

Figure 4.2: All storage strategy

in the meteorological community. This strategy has not been implemented in AD-tools because of the large memory required to store the initial trajectory.

This approach leads to the creation of two routines $p_i^s, p_i'^r$ for each routine p_i in the initial code. The routine p_i^s computes the same output as p_i and stores globally all the computed variables including the intermediate variables. The routine $p_i'^r$ restores all the correct values of the initial variable (from p_i) and computes the derivatives.

The execution path of $P's$ consists of two parts: the first one is the same path as E where each call of p_i is replaced by a call of p_i^s, the second one is the reverse path of E where each call of p_i is replaced by a call of $p_i'^r$.

The execution path of $P's$ is shown in the figure 4.2(b). The symbol $\Rightarrow$ means execution of the function with storage, and $\Leftarrow$ means retrieve of the stored variables followed by execution of the derivative.

The execution time of $P's$ the cotangent code corresponding to the execution E of P is then:

$$\begin{array}{rcl} T' & = & \sum_{i\in E} {t_i}^s + {t_i'}^r \\ & \leq & 5 * T \end{array}$$

We denote g_i the added memory necessary to store all the variables computed by p_i. The global memory G necessary to store recursively all the modified variable through the execution path E is the sum over all the executed routines of the memory necessary for one routine: $G = \sum_{i\in E} g_i$.

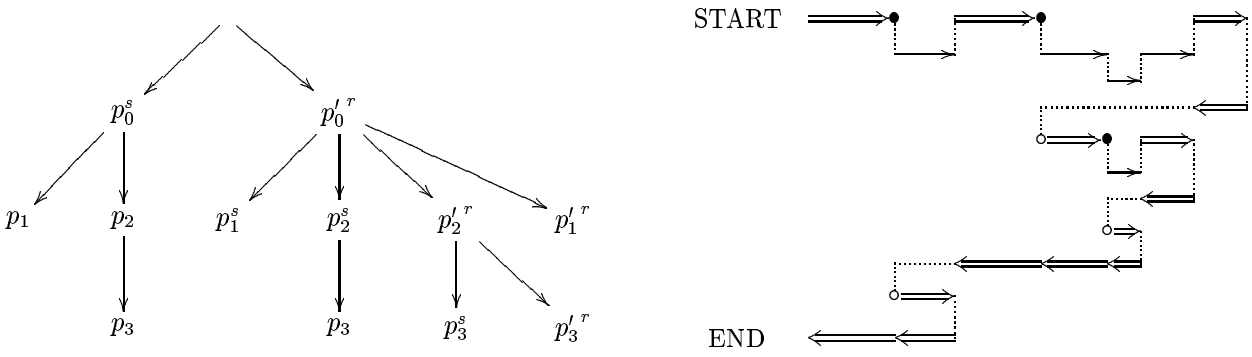
As one can understand from the execution path of $P's$, the memory necessary for the computation of the function followed by the computation of the derivative is:

$$\begin{array}{rcl} M' & = & \max(\max_{b\in B} \sum_{i\in b} m_i, \max_{b\in B} \sum_{i\in b} m_i') + \sum_{i\in E} g_i \\ & \leq & M + \sum_{i\in E} g_i \end{array}$$

This is the maximum of the memory necessary for the computation of each component of P' because the memory necessary for the computation of the forward component can be released at the end of its computation. But $\forall i\ m_i' \leq m_i$ then the maximum between $\max_{b\in B} \sum_{i\in b} m_i$ and $\max_{b\in B} \sum_{i\in b} m_i'$ is M, the memory required by the computation of the initial program P. One must notice that $G >> M$.

4.2 Intermediate strategies

In order to diminish the execution time (wrt. all recomputation strategy) without augmenting to much the memory requirement (wrt. all storage strategy), one can think of using checkpoints. One checkpoint c_i stores some state of the computation at time t_i and is used to compute the adjoint of the computation from t_{i+1} to t_i as the initial point of the forward execution from t_i to t_{i+1}.



(a) Call tree of P'

(b) Execution path of P'

Figure 4.3: Checkpointing at the routine level

Two applications of this idea have been implemented in AD-tools in order to generate code that could be run even for real world initial code; the "optimal checkpointing" and the "checkpointing at the routine level". In this section, we describe those two strategies, but it is clear that a lot of combination of the two extreme strategies could be applied depending on the initial code.

The first one called "optimal checkpointing" is based on theoretical results on the reversion of a sequence of instructions. If you consider any execution of P as a straight line, and if you can divide this execution into n blocks of instructions of the same execution time t, and the same size of context of call m (where the context of call of a block is the set of input variables necessary to run this block). If you have N checkpoints, it is possible to compute their optimal repartition all along the execution of the sequence of blocks that minimizes the recomputation of those blocks. This optimal schedule has been described in different papers [14, 11]. This strategy has been implemented in AD-tools on some patterns of code such as a loop with a fix number of iterations. Then the sequence of instructions is syntactically split into block which are the body of the loop of (nearly) the same execution time and context size. We have to say the complexities of the body of the loop are nearly constant because in practice, you may have differences if some branches appears in the body of the loop. In `Tamc` (see [9]) and `Odyssée` (see [7]) the user can ask the system to apply this strategy using some differentiation options. A general package called `treeverse` has been also developed and is described in [13].

The second one is called "checkpointing at the routine level" and has been implemented in `Tamc` and `Odyssée`. This is a structural checkpointing (described in [17]) as the place where the checkpoints are set depend on the structure of the call tree but is not based on any "optimal trade-off" between storage and recomputation. Using this strategy, the checkpoints are used to store once the input context of each routine, in order to run the sequence $p_i^s; p_i'^r$ at the correct point. Thus, instead of recomputing the initial point from scratch, the generated code restores the initial context of each routine to get the correct computation of the derivative.

This is the strategy used in `Odyssée` as well as in `Tamc` because it is easy to automatize and gives a good compromise between storage and recomputation to the generated code. If the program is made of one routine there is no difference between the "all storage strategy" the "checkpointing at the routine level" strategy, but those codes are really different on a general call tree.

The figures 4.3(a) and 4.3(b) show the call tree and execution path of the derivative where $\bullet$ means storage of the context of the next routine and $\circ$ means the retrieve of this context.

Using this approach, the forward and reverse components of each routine are executed successively, they can thus be implemented in the same routine; in this case for each routine p_i in the initial code one can only generate one routine that executes $p_i^s; p_i'^r$. This routine computes the same output as p_i and stores (locally or globally) all the computed variables including the intermediate variables and after restoring all the correct values of the initial variables computes the derivatives.

We call c_i the memory necessary to store the input context of p_i, l_i the memory necessary to store the modified variables within the p_i routine. Using this strategy, the time execution can be computed by adding for each routine in E the execution of the forward and reverse components plus an extra cost due to the recomputation of the initial routine. The number of recomputations of each routine p_i is exactly the depth of the routine in the call tree.

$$\begin{array}{rcl} T' & = & \sum_{i\in E}(t_i^s + t_i'^r) + \sum_{i\in E} d_i * t_i \\ & \leq & 5 * T + \sum_{i\in E} d_i * t_i \\ M' & = & \max_{b\in B} \sum_{i\in b}(m_i + m_i' + c_i + l_i) \\ & \leq & 2 * M + \max_{b\in B} \sum_{i\in b}(c_i + l_i) \end{array}$$

4.3 Conclusion

We denote c_t ($c_t \leq 5$) the ratio in execution time of the adjoint instructions with respect to the initial function and c_m ($c_m \leq 2$) the ratio in memory requirement of the derivative variables. Those coefficients characterize the derivative code and do not depend on the strategy: they depend only on the sequence of instructions executed in the initial function and on the variable with respect to which those instructions are differentiated. Thus, we define an optimal cotangent code as a derivative code which execution time is $c_t * T$ and which memory requirement is $c_m * M$ where T is the execution time of the initial code and M the memory required be the computation of the initial variables. The figure 4.3 recalls all the complexity results we have shown for each strategy presented as extra cost with respect to those optimal values.

Strategy	Execution time	Memory requirement
Optimal code	$c_t * T$	$c_m * M$
All recomputation	$\sum_{i\in E} \sum_{j<i} t_j$	$c_0 + \max_{b\in B} \sum_{i\in b} l_i$
All storage	0	$\sum_{b\in B} \sum_{i\in b} g_i$
Checkpointing at the routine level	$\sum_{i\in E}(d_i + 1) * t_i$	$\max_{b\in B} \sum_{i\in b}(c_i + l_i)$

The time complexities can be easily compared because the extra cost is due to initial function evaluation. As for the memory requirement, the complexities shown above are uneasy to compare due to the fact that they are evaluated in terms of l_i, c_i, g_i depending on the chosen strategy. Ideally if g_i is defined as the set of variables modified during the execution of p_i at one level, one wants g_i to be equal to $c_i + l_i$. The optimal c_i to be stored is not the whole context of call of p_i but only the sub-set of its input (read) variables which are modified (written) by it's call. This optimization applies for the checkpointing at the routine level but not for the general checkpointing.

In practice, those measures l_i, c_i, g_i depend on the efficiency of the analysis of the code (automatic or manual). For example, it is really difficult to detect automatically which components of an array have be computed through some instructions (see [16]). The hand coded adjoint codes are therefore really more efficient that automatically generated ones.

In order to get those complexities we have made the hypothesis that; the extra memory requirement depends only on the modified variables, the memory management does not cost any execution time, a call to a sub-program does not cost any time either. Those hypothesis are not verified on any machine using any compiler. We know for sure that a call to a sub-program does cost some time but this extra cost is impossible to evaluate even in order of magnitude. On some large codes, this can be 1/3 of the total execution time if the storage is completely dynamic (see [6]). This is a general problem of the reverse mode.

Chapter 5

Conclusion and future work

In this paper, we have described the main approaches used to make the Reverse Mode of automatic differentiation applicable on real world codes. Some of those techniques have been implemented in automatic differentiation tools (`Tamc`, `Odyssée`), but some are only used to write discrete adjoint codes by hand. For example, the "all stored" approach can only be used for hand written discrete adjoint because it requires a great deal of storage optimization to be applied on a real code. We want to implement this strategy within `Odyssée`, but in order to diminish the storage, we have to refine the code analysis strategies used to compute all the variable to be stored. A semi-automatic application of `Odyssée` to generate a discrete adjoint for a large code `Meso-nh` has been successful. This works is presented in [5, 4].

In this paper, we did not take into account the execution time necessary to manage the extra memory due to the storage of the intermediate computations. We want to study a general way of storing/retrieving a large amount of values at a minimal cost. We will compare static storage, dynamic storage and a mixture of those techniques and try to implement a general package that could be used in any AD-tool.

We will also study and implement new hybrid strategies between storage and recomputation to enable the user to chose the approach suitable for his code. For example, one idea used in hand coded adjoint is to store the state vector which is computed at each time iterations or each spatial iteration. Then instead of storing all the modified variables at all the step of the loop, only the most important variables are stored, and the other "intermediate" variables are recomputed.

An other interesting problem is the aliasing problem in the reverse mode of automatic differentiation. For example, if you replace `Y` by `Z` in the adjoint code of $\mathcal{G}_1$ shown in figure 2.6 you get a wrong answer. Moreover, if you replace `Y` by `X` in the adjoint code of $\mathcal{G}_2$ shown in figure 2.6 you get a non valid code as the result depends on the compiler if `dX,dY` are arguments of the routine. In fact aliasing by call of a routine leads to great trouble using the reverse mode, because the status (read, written) of the dual variables is the inverse of the status of the corresponding initial variable.

Some other directions for making the reverse mode automatically applicable on real world code is being studied. For example, the hierarchical approach ([3]) and the cross country elimination (in [8] `pp 47-51`) could be used to mix direct and reverse mode and then diminish the memory cost. The use of parallelism for adjoint code generation is also investigated (in [8] `pp 23-29`) as well as parallel checkpointing.

Bibliography

[1] W. Baur and V. Strassen, *The complexity of partial derivatives*, Theoretical Comp. Sci. **22** (1983), 317–330.

[2] M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank (eds.), *Computational differentiation: Applications, techniques, and tools*, SIAM, Philadelphia, 1996.

[3] C. H. Bischof and M. R. Haghighat, *Hierarchical approaches to automatic differentiation*, Computational Differentiation: Applications, Techniques, and Tools (M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds.), SIAM, 1996, pp. 83–94.

[4] I. Charpentier, *Génération de codes adjoints : Traitement de la trajectoire du modèle direct.*, Rapport de recherche 3405, INRIA, April 1998.

[5] I. Charpentier and M. Ghemires, *Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée. Application au code météorologique Meso-NH*, Rapport de recherche 3251, INRIA, September 1997.

[6] C. Faure, *Le gradient de THYC3D par Odyssée*, Rapport de recherche 3519, INRIA, October 1998.

[7] C. Faure and Y. Papegay, *Odyssée User's Guide. Version 1.7*, Rapport technique 0224, INRIA, September 1998.

[8] C. (Ed.) Faure, *Automatic differentiation for adjoint code generation*, Rapport de recherche, INRIA, November 1998.

[9] R. Giering, *Tangent linear and Adjoint Model Compiler , Users manual*, 1997, Unpublished, available from `http://puddle.mit.edu/~ralf/tamc`.

[10] J.C. Gilbert, G. Le Vey, and J. Masse, *La Différentiation Automatique de fonctions représentées par des programmes*, Rapport de recherche 1557, INRIA, November 1991.

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

[12] A. Griewank and G.F. Corliss (eds.), *Automatic differentiation of algorithms: Theory, implementation, and applications*, SIAM, Philadelphia, 1991.

[13] A. Griewank and A. Walther, *Treeverse : An implementation of the checkpointing for the reverse or ajoint mode of differentiation*, Tech. report, TU Dresden, 1997.

[14] J. Grimm, L. Pottier, and N. Rostaing-Schmidt, *Optimal time and minimum space-time product for reversing a certain class of programs*, Computational Differentiation: Applications, Techniques, and Tools (M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds.), SIAM, 1996, pp. 95–106.

[15] J. Morgenstern, *How to compute fast a function and all its derivatives, a variation on the theorem of baur-strassen*, Sigact News **16** (1985), 60–62.

[16] M. Tadjouddine, C. Faure, and F. Eyssette, *Sparse jacobian computation in automatic differentiation by static program analysis*, Static Analysis (G. Levi, ed.), Lecture Notes in Computer Science, vol. 1503, Springer-Verlag, September 1998, pp. 311–326.

[17] Yu.M. Volin and G.M. Ostrovskii, *Automatic computation of derivatives with the use of the multilevel differentiating technique - 1. algorithmic basis*, Computers & Mathematics with Applications **11** (1985), no. 11, 1099–1114.

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

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 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 - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

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