Comparing Strategies for Coding Adjoints

Christèle Faure, Isabelle Charpentier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Comparing strategies for coding adjoints

Christèle Faure and Isabelle Charpentier

N° 3811

Novembre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3811--FR+ENG

SOPHIA ANTIPOLIS

Comparing strategies for coding adjoints

Christèle Faure* and Isabelle Charpentier†

Thème 1 — Réseaux et systèmes

Projet TROPICS

Rapport de recherche n° 3811 — Novembre 1999 — 17 pages

Abstract: This document presents and compares methodologies to generate discrete adjoint codes. These methods can be implemented when hand writing adjoints, or within Automatic Differentiation tools. Automatic differentiation has been applied successfully to industrial codes that are large and general enough to fully validate this new technology. Future AD tools will make use of some general methods presented in this paper but still not implemented within present tools. In this paper, we try to compare these strategies in terms of accuracy, execution time and memory requirement on a one dimensional thermal-hydraulic module for two-phase flow modeling.

Key-words: discrete adjoint, automatic differentiation, reverse mode, storage/recomputation strategy

This work was supported by the INRIA cooperative research action for Operative Inverse Mode (MIO).

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

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 comparées pour coder des adjoints

Résumé : Ce document présente et compare des méthodologies pour écrire des adjoints discrets. Ces méthodes peuvent être utilisées lors de l'écriture à la main de codes adjoints ou implémentées dans un outils de différentiation automatique. La différentiation automatique est une technologie qui a maintenant fait ses preuves à travers des applications à des codes industriels de généraux et de grande taille. Les outils futurs de différentiation automatique prendront certainement en compte les méthodes décrites dans ce document pour offrir des codes adjoints spécialisés. Nous avons comparé les différentes stratégies en terme de temps de calcul et de taille mémoire nécessaire à l'exécution sur une maqette mono-dimensionnel d'un code de thermohydraulique industriel.

Mots-clés : adjoint discret, différentiation automatique, mode inverse, stockage/recalcul

1 Introduction

Optimal control techniques [Lio71] commonly used for solving inverse problems require adjoints. The two methods for getting adjoint codes one can think of, are: writing the code from the derivatives of the continuous mathematical equations, or differentiating the code that discretizes the continuous mathematical equations. Note that the derivatives produced by the two methods may differ depending on the equations and their discretization, and a theoretical analysis is necessary to choose between the two methods.

If one wishes to obtain a discrete adjoint from an existing computational code, one may choose either to write it by hand or to generate it using Automatic Differentiation (AD), or to mix both techniques. In the communities where adjoint codes are well known, methodologies have been developed to help the hand coder write his discrete adjoint "easily". People who have done it once know that the terminology "easy" is never really appropriate for such a task. But hand-coding is certainly possible, and the methodology is clear enough even if the testing and debugging of the resulting code are tedious. Basically, for each statement in the source code, some adjoint statements have to be coded, then the sequence of derivative statements has to be reversed and the call tree of the source code has to be transposed. All the intermediate values computed have to be stored to be able to apply the derivative statements at the point where the initial statement was applied.

Automatic differentiation (see [GC91] and [BBCG96]) is a set of techniques for computing derivatives at arbitrary points. Automatic differentiation is based on two main observations: first a program execution can be seen as a composition of functions, and second it can thus be differentiated using the chain rule. The derivatives of elementary statements are computed using standard rules for differentiating expressions such as: “the derivative of a sum is the sum of the derivatives” ...

Two modes of automatic differentiation have been studied and implemented by various authors: the direct (or forward) mode that propagates directional derivatives, and the reverse (or backward) mode that propagates adjoint values. The reverse mode is particularly efficient for computing gradients because its cost is independent of the number of input variables [Mor85, BS83]. Two classes of automatic differentiation tool exist: those that work by code generation, and those that work by operator overloading. `Odyssée`, Adifor, GRESS and TAMC belong to the first class.

We call **adjoint code** a code that computes the product $J^T d$ where J is the Jacobian matrix of a function F encoded within a code C and d is an adjoint direction (initial values of the adjoint variables). Such an adjoint is generated from the original code C by: differentiating each statement as an elementary function, and propagating backward the adjoint direction. The code generated using the reverse mode of AD by source transformation is in this sense equivalent to a hand written discrete adjoint. Both codes compute $J^T d$ and if one looks inside the code they are likely to be the same at the local level, but tend to differ significantly on a global level. At the statement (local) level, the derivative statements are composed using the chain rule, the control structure is reproduced and the flow is reverted. At the global level, the composition of the derivative sub-program differ as well as the way intermediate computations are stored and restored. The local and global strategies are coupled to generate an adjoint code, either by hand or automatically. This coupling is fixed for each method for generating adjoint codes, but we want to show that these initial choices could be revised depending of the orginal code as well as the constraint in generation time, execution time and memory requirement.

In this paper, we describe the differentiation of one statement (local strategy), and the global strategies and we compare several possible global strategies as well as storage methods and evaluate their efficiency on a given code. In the second section, we describe classes of problems for which adjoint codes are required and we motivate the choice of the method of differentiation (from equations, or from discretized equations). The third section is devoted to the description of strategies local to the differentiating of one statement. In the fourth section, strategies global to the differentiation of a whole program are described. Section five shows

on a trial code how the codes generated by coupling the local and global strategies as well as the storage methods may compare in terms of execution time and memory requirement.

2 Discrete gradient (DG) or discretized continuous gradient (DCG)

Inverse problems solved by optimal control methods [Lio71] and [CGM73] include shape design problems, forecasting experiments and pollutants sources identifications.

Let P be a partial differential equation problem and J be a cost function depending on either some shape parameters (x) and the state variable (s) of P, the computational goal is to minimize J over all couples (x, s). According to this formulation, the inverse problem constructed to identify approximation of, for example, the parameter values is solved as long as one can evaluate the gradient of J with respect to those parameters. There are two manners for computing the gradient of J. On one hand one may discretize the continuous gradient, in the sequel this method is denoted by DCG for discretized continuous gradient. A second approach consists in first discretizing the state equations and the cost function through, for example, a finite differences scheme. The subsequent differentiation of the discrete schemes allows the computation of the gradient of the discrete cost function, this discrete gradient method is called DG. Three examples are given below to describe benefits and drawbacks of either approaches.

For optimal shape design problems, DCG and DG methods have been proven equivalent (see [CL94] and [Lod92]) for elliptic problem solved within an interior approximation of the geometry. In that case, the DCG method is applied to the continuous equations P and the cost function so as to obtain the continuous adjoint equations P^a and the continuous gradient equation ∇J that are then discretized. The advantages of the DCG method are a low computational cost of each gradient computation since P and P^a are often solved at the same cost. Furthermore the coding work is rather easy: same solvers, no differentiation of the discrete equations. However this method may induce a lack of convergence in an optimization process (especially line-search failure): the cost function J and its gradient ∇J may not be exactly consistent. Since a given optimization algorithm deals with a discrete scheme representing J, it expects the DG gradient of J: the optimization algorithm may be confused by the use of the DCG gradient. In [Roc97], the author compares the two methods for a shape optimization problem where the DG and the DCG results differ slightly. The difference in the resulting optimal shapes has a size proportional to the discrepancy in the gradients. In that case, the author chose the DCG method to compute optimal shapes for its lower execution time.

Several examples showing a difference between DCG and DG methods are presented in [Cha79], the discrete gradient being the more appropriate one for an optimization process. The second example is devoted to variational data assimilation technics [LDT86]. This optimal control method is commonly applied to geophysical problems such as weather forecast, identification of pollutants sources, ... In that case, inverse problems are composed of a set P of evolutionary state equations governing the physics coupled to a cost function J that measures the discrepancy between observations and the solution of P. They are both discretized and implemented as a computer code. A theoretical computation of the continuous optimality system (DCG method) of a model containing a large set of equations and variables is a tedious task that may be of no use. The adjoint code is then determined by the choice of the computational code used to discretize the direct model P: this ensures the consistency of the computed discrete gradient with the computed discrete cost function that will be given to any optimization code.

The special issue of evolutionary problems discretized through a Leap-Frog scheme is treated in [ST97]. It is well known that the adjoint of a discrete model coded by a Leap-frog scheme differs from the Leap-Frog scheme discretizing the adjoint equations. For that particular case the authors show that there is no obvious answer. For instance time-dependent sensitivity analysis experiments realized with the DG method include strong computational modes: the sensitivity of the numerical scheme overwhelms the model sensitivity, the DCG differentiation has to be used. On the other hand a data assimilation process requires the simultaneous computation of the gradient: the DG method is then preferred since it "ensures the convergence" of the

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

(a) Code `A`

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

(b) $\mathcal{F}$

Fig. 1: An assignment `A` seen as an elementary function $\mathcal{F}$

optimization method. As a conclusion, the two methods (DCG and DG) for adjoining equations have to be used carefully, especially if the resulting gradients are different. A theoretical analysis is then necessary to determine the domain of validity of either approach.

In this paper we only study the DG method that requires differentiating the discrete set of equations encoded within a computational code. One notices that the hand coding of an adjoint code is a tedious and error prone task that may be avoided thanks to automatic differentiation tools such as `Odyssée`.

3 Local strategies

In this section, we describe three strategies for the differentiation of a sequence of assignments. A general code does contain statements other than assignments such as branches and loops; the derivative of a branch is the same branch; the derivative of a loop `Do (i,1,n,1)` is a new loop `Do (i,n,1,-1)` with the same number of iterations, but executed in reverse order. The way automatic tools and hand coder generate the derivatives of such statement is the same, we do not describe them here. The first part presents the mathematical point of view whereas effective source generation methods are described in a second part. These source generation strategies can either be used when hand-coding adjoints or developing AD tools.

3.1 Local differentiation

Each statement must be differentiated so that the corresponding piece of code computes in reverse mode, the product of its local transposed Jacobian matrix by an "adjoint" direction.

For example, the assignment `A` shown in figure 1(a) can be seen as an elementary mathematical function f where mathematical input and output domains of f are $\mathbb{R}^2$ and $\mathbb{R}$. However, one has to extend its input and output domains to be able to built up the composition of the elementary functions corresponding to a sequence of statements. As we will see in this section, this leads to compute the product of the Jacobian matrices of each elementary functions. The extension $\mathcal{F}$ of f from $\mathbb{R}$ to $\mathbb{R}^2$ is presented in figure 1(b), and shows that all the variables must be interpreted as potential input and output variables.

The product of the transposed Jacobian matrix of $\mathcal{F}$ by the "adjoint" direction $(dX^\star, dY^\star)_i$ can be easily built. Figure 2 shows two equivalent representations of this computation in mathematical notation (where $_{i,o}$ indicate input and output values of a variable), from figure 2(a) one simply deduces the corresponding adjoint straight line code `A'` written in figure 2(b). One notices that a mathematical variable can be set but never modified, this is why we have introduced the $_{i,o}$ suffixes. On the contrary, on a computer, variables are memory locations and can be modified. In order to optimize the code in terms of number of intermediate variables, the dX_i and dX_o variables are identified and denoted `dX`. In the same manner, one can derive the adjoint code `B'` (see 3(c)) of a second statement `B` (see 3(a)) as shown in Figure 3. From these examples, on can see that the adjoint code of one statement is a sequence of statements where the number of statements is equal to the number of variables of the right hand side of the original assignment.

$$\begin{pmatrix} dX^\star \\ dY^\star \end{pmatrix}_o := \begin{pmatrix} 1 & Y^2 \\ 0 & 2XY \end{pmatrix} \begin{pmatrix} dX^\star \\ dY^\star \end{pmatrix}_i$$

(a) ${J_A}^T$

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

(b) Code `A'`

Fig. 2: Adjoint code of `A` in the "adjoint" direction `(dX,dY)`

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

(a) Code `B`

$$\begin{pmatrix} dX^\star \\ dY^\star \end{pmatrix}_o := \begin{pmatrix} 1 & 2XY^3 \\ 0 & 3X^2Y^2 \end{pmatrix} \begin{pmatrix} dX^\star \\ dY^\star \end{pmatrix}_i$$

(b) ${J_B}^T$

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

(c) Code `B'`

Fig. 3: Adjoint code of `B` in the "adjoint" direction `(dX,dY)`

To get the adjoint code of the sequence `[A;B]`, one combines pieces of code `A,B,A',B'`. From a mathematical point of view, this combination is the transposition of the product of the two Jacobian matrices J_A and J_B. From this, one derives that if the original statement `A` is executed before `B`, its adjoint derivative `A'` must be executed after `B'`. One gets from the initial sequence `[A;B]` the derivative sequence `[B';A']`, where `A'` and `B'` must be evaluate on the correct values of `X` and `Y`.

The main difficulty of the adjoint generation appears: the original values of all the intermediate variables have to be restored before the evaluation of the corresponding Jacobian matrix. The next section shows different strategies to solve this problem.

3.2 Local storage/recomputation strategies

As we show in the previous section, the context of evaluation of the local (transposed) Jacobian matrices must be reproduced. The first manner is to recompute the values from some initial data whereas the second one is to store some values during a direct run and to restore them before the evaluation of the (transposed) Jacobian matrix.
We call **direct part** the code that computes the original function (if necessary) and stores the values of some intermediate variables, and **reverse part** the code that computes the derivatives after restoration of those intermediate values.

Let `E` be the execution trace of an original sequence of statements where each statement is labeled by its index `(i)`. We represent this trace as a sequence `[n;...;m]` where `n,m` are statement indexes. The next section describes three general strategies that can be applied at a local level:

Recomputation from initial values The first strategy (illustrated in Figures 4(a), 4(d)) consists in storing the initial values (at some level) of the variables overwritten by the program. More precisely, for each statement `(i)` in reverse order in the original execution `E` of the program, the value of the input variables are restored and the statements `[0;i-1]` are executed to get the original context of the statement `(i)`, then `(i')` (generated as shown in the previous section) is executed. One notices that when using this strategy, the direct part consists in storing the initial values of the input variables, and the reverse part restores these values to recompute the original function and to compute the derivatives.

Storage of modified intermediate variables The second strategy (illustrated in Figures 4(b), 4(e)) consists in storing the values of all variables modified by the original program. The values of overwritten variables are stored before each statement `(i)` in the original execution `E` of the program and these values are restored before each derivative statement `(i')` (in reverse order in `E`). In this case, the direct part of the generated code executes the function and stores the values of the modified variables, whereas the reverse part restores the values and computes the derivatives.

Storage of partial derivatives The third strategy (illustrated in Figures 4(c), 4(f)) consists in storing the partial derivative of each statement in the original program. All the partial derivatives are stored before each statement `(i)` in `E`, and each derivative `(i')` in reverse order in `E` is written as a sum of products of these partials by the corresponding direction. Using this strategy, the direct part computes the original values as well as the partial derivatives, whereas the reverse part only computes the product of those partials with the direction.

Figure 4 shows these strategies applied on a simple code `C` made of the two statements `A,B` examined in the previous section:

```
(A)   Y = X * Y**2
(B)   Y = Y**3 * X**2
```

For each of the three strategies described above, we show in column: the direct part at the top, and the reverse part at the bottom. If one labels the two statements `(A)` and `(B)` as shown in 4, the execution of the code `C` can be described by the sequence of statements `[A;B]` and the adjoint code `C'` must run the sequence `[B';A']`, where `(B')` is the sequence of statements that corresponds to the adjoint of `(B)` and `(A')` is the derivative of `(A)`. As we said before the two derivatives `A',B'` must be evaluated on the correct values of `X,Y`. In this example the variable `Y` is used and overwritten in both statements, but `X` is never modified: the problem will be to get the correct value of `Y`.

```
      S = Y
(A)
(B)
```

(a) D1

```
      V0 = Y
       Y = X * Y**2

      V1 = Y
       Y = Y**3 * X**2
```

(b) D2

```
      P0 = Y**2
      P1 = 2*Y*X
       Y = X * Y**2

      P2 = 2*X*Y**3
      P3 = 3*Y**2*X**2
       Y = Y**3 * X**2
```

(c) D3

```
      Y = S
(A)   Y = X * Y**2
(B') dX = dX + 2*X*Y**3 *dY
     dY =   3*Y**2*X**2 *dY
```

```
      Y = S
(A') dX = dX + Y**2 *dY
     dY =     2*Y*X *dY
```

(d) R1

```
       Y = V1
      dX = dX + 2*X*Y**3 *dY
      dY =   3*Y**2*X**2 *dY

       Y = V0
      dX = dX + Y**2 *dY
      dY =       2*Y*X *dY
```

(e) R2

```
      dX = dX + P2 *dY
      dY =      P3 *dY

      dX = dX + P0 *dY
      dY =      P1 *dY
```

(f) R3

Fig. 4: Adjoint codes of C in direction `(dX,dY)` using various strategies

We call Y_0 the initial value of Y, and Y_1 (Y_2) its value after execution of A (B). In order to compute the correct derivatives Y must be set to Y_1 before computing the partial derivatives necessary to B' and Y must be set to Y_0 before computing the partials of A'.

The first column (figures 4(a), 4(d)) shows the generated code using the first strategy. In this example, the only variable to be stored is Y and its initial value Y_0 is written in the new variable S. The value Y_1 of the variable Y is restored by executing [Y=S;A] before B', and Y_0 is restored to Y_0 by executing [Y=S] before A'.
The second column (figures 4(b), 4(e)) shows the generated code using the second strategy. Using this strategy, the two necessary values Y_0,Y_1 of Y are stored in V_0,V_1 respectively. Before the computation of B', the value of the variable Y is restored to Y_1 by [Y=V_1] and before A' its is restored to Y_0 by the assignment [Y=V_0].
The third column (figures 4(c), 4(f)) shows the generated code using the third strategy. The two partials necessary to the computation of A' are set to the variables P0,P1, and computed before A to be correct. The same is done for the partials necessary to the computation of B', they are computed before B and stored in P2,P3. In the reverse part, the only thing to be done is the product of the Jacobian matrix (stored into P0,P1 and P2,P3) by the direction dX,dY.
One can notice that the statement (B) is not necessary to the computation of the derivatives [B',A']. We keep it here to be general: if one adds a new statement which uses Y, the statement (B) is necessary.

3.3 Conclusion

The local strategies described above are the main ones one can think of, they can be easily mixed by hand, but this possibility has not been completely exploited in existing AD tools. The first method cannot be applied as a general strategy as its cost in recomputation of the original statement is quadratic in the number of statement. In contrast, the second and third method can really be applied. Those three strategies are generally mixed when hand coding adjoint and could be implemented in AD tools.

In addition to those general methods, complementary optimizations are also applied. An interesting improvement is to share common sub-expressions. This idea can be applied on constant sub-expressions at the statement level between one statement and its derivative statements. In the example developed in this section this would reduce the number of products necessay for the computation of (B) and (B') from 19 to 9 using two supplementary variables I1,I2 assigned respectively to I1=X*Y**2 before (B) and I2=I1*dY before (B'). An extension of this is to consider not only one statement, but a sequence of statements. If statements are grouped into blocks in such a way that no input of a block is modified, the Jacobian matrix can be computed at the block level to exploit the implicit sharing of sub-expressions between the block function and its derivative block. This kind of optimization is one of the basic optimizations implemented in compilers, and could then be avoided. But, in adjoint codes the original statement can be "far" from the derivatives, and compilers are sometimes unable to apply such optimizations. Some experiments on the influence on execution time of sub-expressions sharing within AD generated code are described in [Fau96]. In this paper we have shown that introducing temporary variables can cancel the benefits of the optimization phase performed by the compiler. But in some case (reverse mode) the use of temporaries can also improve the efficiency of the generated code. It is really difficult to no exactly how those two level of optimization combines when the execution time is the criterion.

The management of the values to be stored and restored is one of the main problem of adjoint codes: one can chose to use statically allocated memory, dynamically allocated memory or external memory (files). The characteristics of the runtime will change a lot depending on the strategy of implementation chosen and on the machine where the code is run. We show that in the fourth section of this paper.

4 Global strategies

In this section we will use a description of a program, named **call tree**. In the call tree, each node represents a sub-program of the source and each arrow links a routine with a routine it may call. If a routine appears twice in the source code of the program the corresponding node is duplicated. The call tree is a representation of the program at compile time, thus it contains branches that are not followed in all executions, and does not show the number of time each branch is walked through. When executing the program, the call tree is walked through from the top to the bottom and from the left to the right. We define the **depth** of a sub-program to be 0 for the root of the tree, and to be $\text{depth}(q) = 1+\text{depth}(p)$ for q where p is the father of q. The depth of a tree is the maximum depth of all the leaves of the tree.

For example, figure 5 shows the call tree of a program `P` made of four sub-programs $p_0, \cdots, p_3$. This program `P` is executed from p_0 and one can see that it first calls p_1 then p_2 and that p_2 calls p_3. From this call tree one does not see whether p_3 is really called during execution E, nor does one understand how may times the branch p_2, p_3 is called during E.

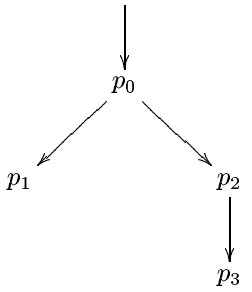


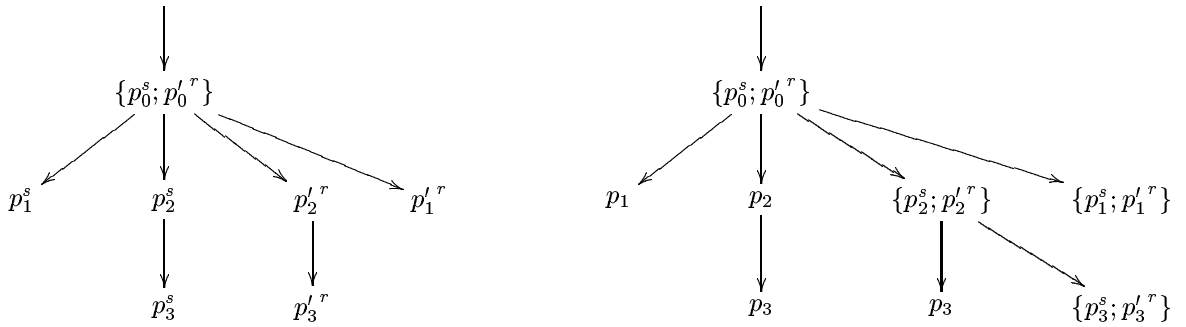
Fig. 5: Call tree of `P`

We call `P'` the derivative program of `P` with respect to some active inputs chosen on the root sub-program. If only some output derivatives are required, active outputs must be specified. Differentiating `P` in `P'` as a whole function requires the knowledge, at each point of the original code, of all the **active** variables i.e. variable that depend on the active inputs chosen on the root sub-program and that impact the output given by the user. Getting this information correct is the first difficulty encountered when coding an adjoint, and doing it by hand is really tedious. AD tools know how to propagate the activity information from the root to all the nodes in the call tree. When the active input variables are known for each sub-program, the derivative of `P` is completly defined and the sub-programs can be differentiated independently. In this paper, we assume that any original routine is differentiated in a unique routine that computes the maximum number of adjoint variables. This means that if an original routine has to be differentiated with respect to two sets of parameters depending on the place where it is called in the program, the system will generate only the maximal derivative considering the union of the two sets.

In the next section, we consider the strategies that are applicable at the call tree level to built up the derivative program. These global strategies can be coupled with any local strategy previously described.

4.1 No recomputation strategy (NR)

The strategy often used for hand coding adjoints is to store intermediate calculations all along the execution path (referred to as **trajectory**). The corresponding derivatives are computed from those initial values taken in reverse order. If one carries this choice to its limit, one stores all the trajectory. We describe this strategy even if it has to be mixed with some recomputation to be effectively applied.



(a) Using the "no recomputation" strategy

(b) Using the "subroutine recomputation" strategy

Fig. 6: Call trees of P'

Using this strategy, each initial routine p_i is associated with two routines $p_i^s, p_i'^r$ where p_i^s computes and stores a trajectory recursively on the call tree, then $p_i'^r$ restores the trajectory and propagates the derivatives. These two routines must access the same intermediate values of the trajectory.

The effectiveness of such a strategy really depends on the ability of the code writer to control the storage. If the trajectory cannot be stored in memory, a file is then necessary and that changes the performances. To reduce the storage some sub-strategies are used: only costly and representative variables are stored (for example the state vector in an iterative scheme) whereas some local intermediate values are recomputed. Such a strategy is often used, but the efficiency of the resulting code really depends on the knowledge of the original code.

Using this global strategy, if access to the trajectory cost no execution time, the theoretical execution time of the derivative is in the worst case 5 times the initial execution time. We approximate the memory required to store the trajectory by the number of lines of the original code turned into a straight line program. This is equivalent to say that each statement (or line) in the program modifies one variable. The complexity in memory of this strategy can then be approximated by the sum of the length of all the routines within the call tree.

4.2 Subroutine recomputation strategy (SR)

AD tools like `TAMC`, `Odyssée` take the problem the other way round and choose to store only the minimum quantity of intermediate values. To do so the recomputation of parts of the program is necessary. The choice has been to do the recomputation at the routine level and not at the block (or the statement) level, in order to avoid the quadratic growth in terms of number of blocks (or statements) of the program.

Using this strategy, each initial routine p_i is associated to one routine p_i' which is the sequence $[p_i^s; p_i'^r]$ where p_i^s computes and stores the trajectory only at one level and $p_i'^r$ retrieves this trajectory and computes the derivatives. One must notice that to be able to call p_i and its copy p_i^s with the same input variables is essential. For this purpose, p_i^s (respctively $p_i'^r$) not only stores (respctively restores) the variables modified locally by p_i, but also the contexts of call of all the sub programs it calls directly. As, the two parts p_i^s and $p_i'^r$ of a derivative subroutine share only the trajectory local to p_i, this trajectory may be stored within local variables.

If one looks at the call tree of the execution P (using the "subroutine recomputation strategy") in figure 6(b), one observes that each original routine is executed a number of times equal to its depth in the call tree.

This strategy is standard in AD tools and is easy to implement when hand-coding adjoint codes. For each sub-program in the program, three derived sub-programs are necessary: the original routine p_i, the direct part p_i^s and the reverse part p_i^{lr}.

Using this global strategy, if access to the trajectory cost no execution time, the theoretical execution time of the derivative is in the worst case $5 + d$ times the initial execution time, where d is the depth of the call tree. Using the same approximation of the memory requirement as in the previous section, the memory required to store the trajectory is the maximum, over all the branches of the call tree, of the sum of the length of all the routines in a branch, plus the sum of the size of all the contexts of call of the sub programs encountered on the branch. Those contexts have to be stored to insure that the sub program p_i and its copy p_i^s are computed with the same input variables. Using this strategy, the problem of storage is less critical since only at most one branch of the trajectory is stored.

4.3 Conclusion

On can conclude from the previous sections, that if the NR strategy is minimal in terms of execution time, the SR strategy is minimal in terms of memory locations necessary to store the trajectory.

Both methods can be applied and mixed depending on the knowledge of the source code of the adjoint developper. If one knows nothing on the source code, it is better to apply the second strategy generally used in AD tools to be sure to get a correct program. Furthermore, it is easy to modify a code generated using the second strategy to get a code using the first one as shown in [CG99] for the Meso-NH code. A more precise complexity analysis of these strategies can be found in [Fau99].

One must notice that another global strategy called "checkpointing" [Gri92, GPRS96] can be applied to optimize the recomputations. This strategy is applied on just a few routines (one in general) that match a general pattern described in [GPRS96] and the other routines are differentiated using the standard strategies. Some trials of various checkpointing schedules added to the "no recomputation strategy" are presented in [Cha98] whereas the optimal checkpointing schedule combined to the "subroutine recomputation strategy" is used in [Fau98].

5 Comparison of the strategies on one example

In this section, we will mainly describe the application of the global strategies described above on `Thyc-1D` (developed at EDF-DER) as well as the storage method. We choose this code because it is large enough to get some general observations, and small enough to be validated by hand. This code was not been written in prevision of any adjoint construction and can then be taken as a general example.

All the adjoint codes compared in this section have been generated using the second local strategy: all the modified variables have been stored. In other words we have not introduced recomputation at the statement level. We use the AD tool `Odyssée` to generate fast the adjoint statements as well as the storage statements. We have modified this initial code to generate all the variations necessary for the comparison we intend to do.

5.1 Target code

`Thyc-1D` is a one dimensional thermal-hydraulic module for two-phase flow modeling that simulates the evolution of some parameters in heat exchangers during a certain period of time. It consists of five partial differential equations: three conservation equations for the two-phase mixture (mass, momentum and energy),

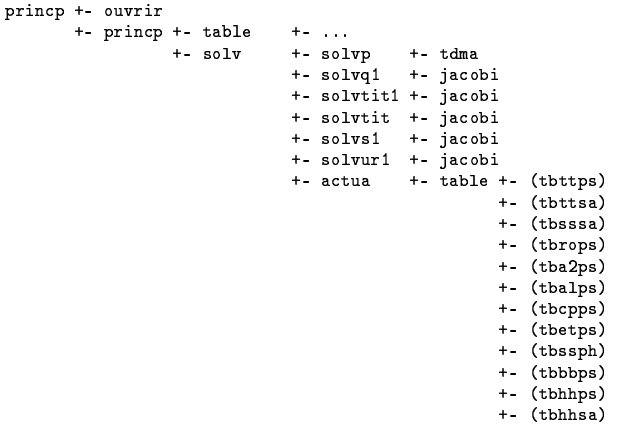


Fig. 7: Call tree of the program `Thyc-1D`

one conservation equation for the vapor mass and one conservation equation for the relative liquid-vapor velocity between the two phases. To compare the different strategies, we choose to evaluate the sensitivity of the relative velocity between phases y with respect to four parameters: x_1 and x_2 included in the inter-facial drag coefficient between vapor and liquid phases, x_3 the relaxation time in boiling modeling and x_4 the thermal power generated in the bundle.

The target code `Thyc-1D` (presented in figure 7) consists of two different kinds of routines: Fortran-77 routines, and also routines from EDF's libraries as fluid thermodynamic properties indicated by brackets in figure 7. These routines exist only as a compiled library and could not be treated as Fortran-77 routines by `Odyssée`. To differentiate it correctly, we have used an information base as described in the User's Guide [FP98], and we have implemented the derivatives using finite differences. Using another AD tool but `Odyssée`, we would have been obliged to write down a Fortran-77 routine for each routine in the library that simulates the dependencies between input and output variables.

`Thyc-1D` can be described as a main loop that simulates an evolutionary process. We use this property to apply the checkpointing strategy to store some steps of this loop, and recompute all the other time steps from those snap-shots.

5.2 Generation of the different codes

The original code `Thyc-1D` is used as the reference for the original execution time and memory requirement (static memory). We use the tangent code TG (automatically generated by `Odyssée`) to verify the value of the gradient computed by the adjoint codes.

We compare adjoint codes using the "Subroutine recomputation (SR)" strategy to adjoint codes using the "No recomputation (NR)" strategy. As for the local strategy, we only consider the "Storage of intermediate values" as it is the strategy which is generally used. We also compare the three storage techniques on both general strategies, let say: statically allocated memory, dynamically allocated memory, external memory (file). And finally, we compare the behavior of the checkpointing applied to both general strategies using static storage for the trajectory as well as for the snap-shots.

Using `Odyssée`, we have automatically generated three adjoint codes of `Thyc-1D` that use the "subroutine recomputation (`SR`)" strategy: `SR-S` uses static storage, `SR-D` uses dynamic storage, and `SR-C` uses checkpoints on the main loop (see [FD98]). We have obtained a fourth adjoint code `SR-E` using "subroutine recomputation"

and storage of the trajectory into a file, by manual modification of SR-S. One must notice that in SR-C, the number of actually used checkpoints can be modified at runtime.

We have also hand coded four adjoint codes of Thyc-1D using the the "No recomputation" strategy NR-S, NR-D, NR-E, NR-C. For this purpose, we have modified SR-S: we have split each routine in SR-S into its direct and reverse parts, and generated the adjoint by calling all the direct parts first, and then all the reverse parts. In SR-S, the storage is local to each derivative routine, we have changed it to be global to the direct part and the reverse part (now separated). We have generated NR-S that uses static global storage, then by modifying the storage we have obtained NR-D and NR-E that use respectively dynamic and external memory. We have also modified NR-S into NR-C to use the checkpointing package called treeverse [GW97].

Name	Global strategy	Storage technique	Generation	
			Method	Time
SR-S	SR	*static*	*Auto*	1 *hour*
SR-D	SR	*dynamic*	*Auto*	1 *hour*
SR-E	SR	*external*	*Semi*	3 *hours*
SR-C	SR, CHK	*static*	*Auto*	1 *hour*
NR-S	NR	*static*	*Semi*	1 *week*
NR-D	NR	*dynamic*	*Semi*	1 *week*
NR-E	NR	*external*	*Semi*	1 *week*
NR-C	NR, CHK	*static*	*Semi*	1 *week*

Tab. 1: Table of adjoint codes characteristics

Table 1 summarizes the characteristics of the different codes. We recall that these eight different codes use the same local strategy which is "intermediate values storage", and store the derivatives and the checkpoints into statically allocated memory. The storage techniques are labeled: *static* for statically allocated storage, *dynamic* for dynamically allocated storage, and *external* for external storage (into files). The global strategies are labeled: NR for no recomputation strategy, SR for subroutine recomputation strategy, CHK for checkpointing of the main routine solv. The column labeled Generation/Method indicates the way the code has been obtained: Auto means fully automatically generated using Odyssée and Semi means hand modified from SR-S (automatically generated). The column labeled Generation/time proposes an evaluation of the time for us to generate a correct adjoint code from SR-S, and not from the original source of Thyc-1D. That means generating, compiling, running and debugging the adjoints.

5.3 Comparisons of runtimes

In this section we try to compare the eight different adjoint codes described in Table 1 in terms of execution time and memory requirement as well as accuracy of the computed gradient. We have run those codes on a SPARC Station with 512M memory. The original simulation stops after 500 time steps but we have forced the program to run for a varying number of time steps: from 50 to 500. In this way, we are able to see the evolution of the execution time and memory requirement in function of the number of time steps.

Table 2 shows the values of the gradient at time step 500 computed by four calls to the tangent code and one call of any adjoint code. We have not shown the gradients obtained by the eight adjoint codes because they do not differ at all. One can see from the fit between the two gradients computed that the accuracy is really good, at least 11 digits are the same in both gradients. In a previous study presented in [DEFG96] we have shown by comparison to centered finite differences that the gradient obtained using AD is really accurate.

Figures 8(a) and 8(b) show the evolution of the execution time with respect to the number of time steps, whereas Figure 8(c) shows the evolution of the storage amount with respect to the number of time steps.

Name	$\partial y/\partial x_1$	$\partial y/\partial x_2$	$\partial y/\partial x_3$	$\partial y/\partial x_4$
TG	−2.6877707476399	−1.1283833581534D-02	0.23953473365123	8.4189731620359D-09
Adj	−2.6877707476403	−1.1283833581538D-02	0.23953473365118	8.4189731620564D-09

Tab. 2: Gradient of `Thyc` in double precision

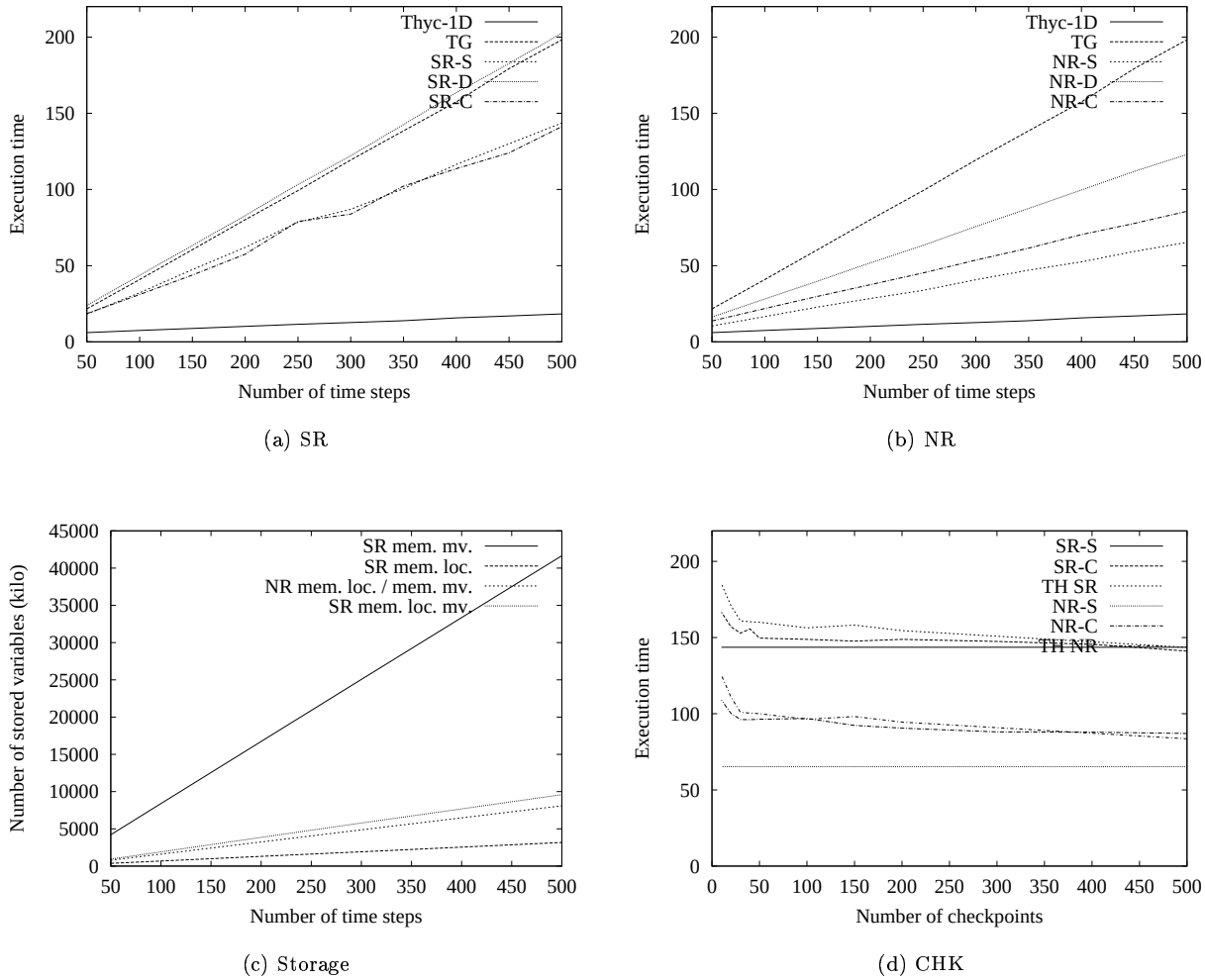
An immediate comment on figures 8(a) and 8(b) is that the execution time of the adjoint codes are lower than the computation time of the tangent code TG run four times to get the whole gradient. On this example, the adjoint codes are faster than the tangent code for a gradient of size four, but this result really depends on the differentiated code and on the machine on which it is run.

If we compare the NR and SR strategy, whatever storage technique is used, the execution time using the NR is lower that using SR (see Figures 8(a) and 8(b)): the ratio is two in favor of the NR technique when the static storage is used. As we said before, this is due first to the recomputation of parts of the original function when the SR strategy is used, but also to the management of the values to be stored (as we will show afterwards). On the contrary, the number of memory locations (mem. loc.) necessary to store the trajectory (see Figure 8(c)) all along the execution using NR is nearly twice as large as using the SR strategy. Those results are consistent with the cost evaluations given in the previous sections.

If we look at those results more in detail, we get some relations between the storage techniques and the execution time (whatever global strategy is used). The static storage `-S` technique as well as the use of static checkpoints `-C500` is a lot faster than the dynamic storage technique `-D` using both global strategies. This result seems independent of the code as well as the target machine. In the figures, we do not present the external storage technique because the performances depend on the way the values are written on the file. Indeed, the results are a lot improved if the values are put into an intermediate buffe, and the buffer is written on the file when it is full. The size of this intermediate buffer and the characteristics of the machine also influences the performances.

Figure 8(c) shows two measures of the memory necessary to store the trajectory: the first measure (mem. loc.) is the number of memory locations used all along the execution, the second measure is the total number of memory moves (mem. mov.) during the execution. As we said before, the number of memory locations (mem. loc.) is lower using SR than using NR. In contrast, the number of memory moves is a lot greater (10 times) using SR than using NR. In order to explain this gap, we have drawn a new measure which is the maximal number of memory moves local to sub programs using SR (mem. loc. mov.) at each time in the execution. This means that the triangle between SR mem. loc. mov. and SR mem. mov. represents the number of moves necessary to store the contexts. This triangle is large, this can be explained by the difficulty to know at compile time what components of an array are modified by a sub program. The system has then to be conservative, and stores the whole array even though only a few components will be modified. We have shown in [Fau98] that depending on the implementation of those memory moves (push and pop statements), the supplementary cost in terms of execution time can be dreadful (1/3 of the total execution time). On can remark that even thought the number of memory moves is 10 times more using the SR strategy than the NR strategy, the execution time is not that larger using the static storage of the trajectory.

If one combines checkpoints to the SR or NR strategy, the evolution of the execution time for 500 time steps with respect to the number of checkpoints used (from 10 to 500) is presented in Figure 8(d). One can notice that for 500 time steps and 500 checkpoints (static), the SR strategy (with static storage) `SR-S` and SR strategy plus checkpoint `SR-C` are really equivalent, whereas for the NR strategy checkpointing `NR-C` costs exactly one run of the initial function. The theoretical curves `TH NR-C` and `TH SR-C` fit perfectly with the practical ones we got. Checkpoints are used to reduce the intermediate storage, on our example we go from 500 checkpoints to 10 which means dividing the memory requirement by 1 to 50. The NR strategy is still faster than the SR strategy, and for 50 checkpoints which means dividing the total storage by 10, the increase in terms of execution time is negligible.



(a) SR

(b) NR

(c) Storage

(d) CHK

Fig. 8: Comparison of the adjoint codes

6 Conclusion

Computational methods using derivatives are classical for a large number of applications involving optimal control theory such as shape optimization or data assimilation in geophysics. The main advantage of using adjoint codes relies in the fact that the execution time as well as the trajectory storage do not depend on the number of input variables (10^6 for data assimilation). For all applications where the code computes a few output variables for a large number of inputs, adjoint code is the only practical way to get derivatives. Until now, adjoint codes used for industrial purposes such as operational weather forecasts were hand coded, but it is now possible to generate them automatically. Hand coded adjoint codes are fast, but their generation generally takes a long time (1 or 2 years). On the contrary, developping an adjoint using and AD tool is fast, but the resulting code has to be improved. What is almost certainly obtained with an AD tool is a correct code: consistent because the propagation of active variables is automatically performed, and locally correct because the derivative of each statement is correct.

In this paper, we describe several methods for the differentiation of sub program (SR and NR strategies). Some of these methodologies are natural when hand coding adjoints (NR), and the others are standard

when automatically generating adjoints (SR). We describe them in some uniform way to be able to compare them, but also to point out what is missing in AD tools until know to compare to hand coding. In the next versions of AD Tools (source-to-source), the choice of the global strategy (SR or NR) will certainly be given. We are working on this subject within `Odyssée`, but it is also studied by other tool developers. When generating adjoint codes, the most fundamental challenge is the knowledge of the trajectory for the evaluation of the local Jacobian matrices. We know that not all the optimizations possible when hand coding can not be automomatized, but some could be. For example, detecting the linear parts of the code are difficult automatically but could really improve the efficiency in memory requirement. On the contrary, changing storage by recomputation is really not an easy task for AD tools. In `Odyssée` we are implementing some new data analysis to guarantee minimal storage (using either SR or NR strategy). In particular, we are working on the storage of the contexts (SR strategy) that seems to be a kee point. But even then human beings will be able to optimize it: detecting dependencies between array components is not automatically possible at compile time (see [Tad99]). As a conclusion, we recommend to adjoint developers the use of AD tools, even if the result has to be modified. After this automatic phase, the restructuration of the storage (if necessary) is easy to do by hand. At least it is a lot easier than testing the derivative statements one by one in the hand coded adjoint. One can easily translate the generated code from SR to NR strategy, and optimize the storage by adding a semi-automatic post-processing step. This has been done for generating the adjoint code of Meso-NH [CG99] with trajectory storage on a file. As for AD tool developers, the aim will be to generate adjoint codes as efficient as hand coded ones. We are working on this within `Odyssée` and the first results we got are really impressive. `Odyssée` will offer all the possible combinations of the strategies described in this paper. Moreover algorithms are being studied to optimize the storage by refined analysis (for NR and SR), or to avoid the storage of the contexts (SR strategy).

Acknowledgment: The authors want to thank Mohammed Ghemires for his work on building by hand some of the adjoint codes used in this paper.

References

[BBCG96] M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors. *Computational Differentiation: Applications, Techniques, and Tools.* SIAM, Philadelphia, 1996.

[BS83] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Comp. Sci.*, 22:317–330, 1983.

[CG99] I. Charpentier and M. Ghemires. Efficient adjoint derivatives: Application to the atmospheric model meso-nh. *Optimization Methods and Software*, 1999. To appear.

[CGM73] J. Cea, A. Gioan, and J. Michel. Some results on domain identification. *Calcolo*, 3/4, 1973.

[Cha79] G. Chavent. Identification of distributed parameter systems: about the output least square method, its implementations, and identifiability. In R. Isermann, editor, *5^{th} IFAC symposium, in Identification and system parameter identification*, volume 1, pages 85–97. Pergamon press, 1979.

[Cha98] I. Charpentier. Génération de codes adjoints : Traitement de la trajectoire du modèle direct. Rapport de recherche 3405, INRIA, April 1998.

[CL94] D. Chenais and V. Lods. Comparaison du gradient discret et du gradient continu discrétisé pour des méthodes éléments finis non conformes. *C. R. Acad. Sci.Paris*, 1994.

[DEFG96] C. Duval, P. Erhard, C. Faure, and J.C. Gilbert. Application of the automatic differentiation tool Odyssée to a system of thermohydraulic equations. In J.-A. Désidéri, P. Le Tallec, E. Oñate, J. Périaux, and E. Stein, editors, *Proc. of ECCOMAS'96*, volume Numerical Methods in Engineering'96, pages 795–802. John Wiley & Sons, September 1996.

[Fau96] C. Faure. Splitting of algebraic expressions for automatic differentiation. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*, pages 117–127. SIAM, Philadelphia, Penn., 1996.

[Fau98] C. Faure. Le gradient de THYC3D par Odyssée. Rapport de recherche 3519, INRIA, October 1998.

[Fau99] C. Faure. Adjoining strategies for multi-layered programs. 1999. Submitted to Optimisation Methods and Software.

[FD98] C. Faure and C. Duval. Automatic differentiation for sensitivity analysis. A test case. In K. Chan, S. Tarantola, and F. Campolongo, editors, *Proceedings of Second International Symposium on Sensitivity Analysis of Model Output (SAMO'98)*, volume 17758. EN, Luxembourg, 1998.

[FP98] C. Faure and Y. Papegay. Odyssée User's Guide. Version 1.7. Rapport technique 0224, INRIA, September 1998.

[GC91] A. Griewank and G.F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications.* SIAM, Philadelphia, 1991.

[GPRS96] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 95–106. SIAM, 1996.

[Gri92] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[GW97] A. Griewank and A. Walther. Treeverse : An implementation of the checkpointing for the reverse or ajoint mode of differentiation. Technical report, TU Dresden, 1997.

[LDT86] F.-X. Le Dimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects. *Tellus*, 38A:97–110, 1986.

[Lio71] J.-L. Lions. *Optimal control of systems governed by partial differential equations.* Springer-Verlag, Berlin, 1971.

[Lod92] V. Lods. *Gradient discret et gradient continu discretisé en contrôle optimal à paramètres distribués.* PhD thesis, Université de Nice-Sophia Antipolis, 1992.

[Mor85] J. Morgenstern. How to compute fast a function and all its derivatives, a variation on the theorem of baur-strassen. *Sigact News*, 16:60–62, 1985.

[Roc97] J.-R. Roche. Gradient of the discretized energy method and discretized. *Appl. Math. Comput. Sci.*, 7(3):545–565, 1997.

[ST97] Z. Sirkes and E. Tziperman. Finite difference of adjoint or adjoint of finite difference ? *Mon. Wea. Rev.*, 125:3373–3378, 1997.

[Tad99] Mohamed Tadjouddine. *Analyse de Dépendances de Jacobiennes Creuses pour la Différentiation Automatique.* PhD thesis, Université de Nice-Sophia Antipolis, 1999.

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