Comparing global strategies for coding adjoints

Christèle Faure[a] and Isabelle Charpentier[b]
[a]*INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France*
[b]*Projet IDOPT (CNRS, UJF, INRIA, INPG), 51 rue des mathématiques, BP 53, F-38041 Grenoble Cedex 9, France*

From a computational point of view, sensitivity analysis, calibration of a model, or variational data assimilation may be tackled after the differentiation of the numerical code representing the model into an adjoint code.

This paper presents and compares methodologies to generate discrete adjoint codes. These methods can be implemented when hand writing adjoint codes, or within Automatic Differentiation (AD) tools. AD has been successfully applied to industrial codes that were large and general enough to fully validate this new technology. We compare these methodologies in terms of execution time and memory requirement on a one dimensional thermal-hydraulic module for two-phase flow modeling. With regard to this experiment, some development axes for AD tools are extracted as well as methods for AD tool users to get efficient adjoint codes semi-automatically. The next objective is to generate automatically adjoint codes as efficient as hand written ones.

1. Introduction

During the last three decades the interest for optimal control [15] techniques grows up in various research communities. From a meteorological point of view, sensitivity analysis, calibration of a model, or variational data assimilation [13,14] are for example performed after the differentiation of the numerical code representing the model. Optimal shape design problems [17] can also be solved by such methods. There is a considerable body of literature on the subject, and the reader is referred to [3] for getting examples and references.

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

Inverse problems require adjoints. The two methods for getting adjoint codes are: writing the code from the derivatives of the continuous mathematical equations, or differentiating the code that discretizes the continuous mathematical equations. Note that depending on the equations and their discretization, the derivatives may differ from one another and a theoretical analysis [18,19] 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, for example in meteorology [20], methodologies have been developed to help the hand coder write his discrete adjoint. Nevertheless the testing and debugging of the resulting codes are tedious tasks.

Automatic differentiation [1,8,11] is a set of techniques for computing derivatives at arbitrary points. AD is mainly based on the following observation: a program execution can be seen as a composition of functions, thus it can be differentiated using the chain rule. 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 AD 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 [4,16]. One notices that an adjoint code requires particular statements that store values for its evaluation. Two classes of AD tools exist: those that work by code generation and those that work by operator overloading. `Odyssée` [7], `TAMC` [9], and the version 3.0 of `Adifor` [2] belong to the first one, whereas `Adolc` [10] belongs to the second class. A theoretical presentation of the strategies used to generate adjoint codes (by hand or automatically) can be found in [6]. In that respect, adjoining strategies are applied at three different levels: local strategies at the

Scientific Programming 9 (2001) 1–10
ISSN 1058-9244 / $8.00

statement level, loop strategies at the loop level and global strategies at the sub-program level. This paper concentrates on a brief presentation of the global strategies, as well as their comparison on a sample thermal-hydraulic code.

The layout of the paper is as follows. Section 2 proposes definitions and explains how to write the adjoint code of a statement whereas Section 3 describes different strategies for generating the adjoint of a whole code. These strategies are used in Section 4 to generate adjoints codes of a trial program. These codes are compared in terms of execution time and memory requirement. Section 5 discusses advantage and drawbacks of the use of AD tools and gives some research axes for AD.

2. Basics of automatic differentiation

An *adjoint code* computes the product $J^T d$ where J is the Jacobian matrix of a function f encoded within a code `P`, and d is an adjoint direction (initial values of the adjoint variables). Such a code is generated from the original code `P` by differentiating each statement as an elementary function in order to propagate the adjoint direction backward as explained in Section 2.1 and Section 2.2. The code generated using the reverse mode of AD by source transformation is in this sense equivalent to a hand written discrete adjoint. Differences between both codes essentially come from global strategies (see Section 3).

2.1. Chain rule application

When a statement `S` is differentiated with respect to a variable `X`, the corresponding piece of code `S'` computes an output direction $\mathtt{dX}_o$ as the product of its local transposed Jacobian matrix by an adjoint input direction $\mathtt{dX}_i$.

For example, assignment `A` shown in Fig. 1(a) may be viewed as an elementary mathematical function f which input and output domains are $\mathbf{R}^2$ and $\mathbf{R}$, that computes $f(x, y) = z$. To build up the composition of the elementary functions corresponding to a sequence of statements, the input and output domains of each elementary function are extended. On this example the extension leads to consider f as a function `F` from input domain $\mathbf{R}^3$ to output domain $\mathbf{R}^3$ such that $\mathtt{F}(x, y, z) = (x, y, f(x, y))$. The vector $d_o = (\mathtt{dX}, \mathtt{dY}, \mathtt{dZ})_o^T$ is the product of the transposed Jacobian matrix of `F` denoted by $J_{\mathtt{A}}^T$ (see Fig. 1(b) by the adjoint input vector d_i. The adjoint code of `A` denoted by `A'` Fig. 1(c) directly derives from the mathematical computation $d_o = J_{\mathtt{A}}^T * d_i$. In order to optimise the code in terms of number of intermediate variables, both $\mathtt{dX}_i$ and $\mathtt{dX}_o$ are denoted by `dX` in the actual piece of code `A'`. From this example, one observes that the adjoint code of one statement is a sequence of statements which length is the number of adjoint variables of the original assignment. The adjoint code `B'` (Fig. 2(c)) is derived from statement `B` (Fig. 2(a)) using the same methods.

In order to get the adjoint code of sequence `[A;B]`, one combines pieces of code `A,A',B,B'` using the chain rule. The transposed Jacobian matrix $(J_{\mathtt{A}} * J_{\mathtt{B}})^T$ of `[A;B]` is the product ${J_{\mathtt{B}}}^T * {J_{\mathtt{A}}}^T$ of the elementary Jacobian matrices. One deduces that if the original statement `A` is executed before `B`, its adjoint derivative statements `A'` are computed after `B'`. The adjoint of the sequence `[A;B]` is then `[B';A']` where `A'` and `B'` must be evaluated on correct values of `X`, `Y` and `Z`. The remaining problem is to get the correct value of `Y` since variable `Y` is used and overwritten. The difficulty of adjoint generation appears: original values of required intermediate variables must be restored before the evaluation of the corresponding Jacobian matrix. The organisation of the final adjoint code of `[A;B]` is described in Section 3. The fundamental constraints on this adjoint are: `A` must be executed before `B'` and `B'` must be executed before `A'`.

Getting the adjoint code of a sequence of assignments is quite easy. But general codes contain complex statements as branches, loops and calls to sub-programs that define control structures. This structure is only determined at runtime and must be reproduced: for example the values of tests or the number of steps of a loop need to be stored or recomputed. We name *trajectory* the list of values required for evaluating the Jacobian matrices or for reverting the computation. As described in Section3.1, the size of the trajectory depends on the storage-recomputation trade-off chosen for the adjoint generation. This problem arises when using AD tools as well as when hand writing adjoint codes.

2.2. Activity propagation

We use a description of a program `P` named *call tree* which is known at compile time. In the call tree, each node represents a sub-program of the source code and each arrow links a sub-program with a sub-program it may call. If a sub-program appears twice in the source code of `P` the corresponding node is duplicated

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

(a) Code `A`

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

(b) $J_{\mathtt{A}}^T$

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

(c) Code `A'`

Fig. 1. Adjoint code of `A` in the "adjoint" direction $(\mathtt{dX}, \mathtt{dY}, \mathtt{dZ})^T$.

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

(a) Code `B`

$$\begin{pmatrix} 1 & 2XZ^3 & 0 \\ 0 & 0 & 0 \\ 0 & 3X^2Z^2 & 1 \end{pmatrix}$$

(b) ${J_{\mathtt{B}}}^T$

```
dX = dX +    2*X*Z**3 * dY
dZ = dZ + 3*X**2*Z**2 * dY
dY = 0
```

(c) Code `B'`

Fig. 2. Adjoint code of `B` in the "adjoint" direction $(\mathtt{dX}, \mathtt{dY}, \mathtt{dZ})^T$.

in the call tree representation of `P`. 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 depth $(q) = 1+$depth(p) for node q if node p calls node q. The depth of a call tree is the maximum depth of all the leaves of the tree. As an example, Fig. 3 shows the call tree of a program `P` of depth 3 made of four sub-programs $p_0, \cdots, p_3$. Program `P` is executed from p_0 that calls p_1 then p_2, and p_2 calls p_3.

We denote by `P'` the derivative program of `P` with respect to some *active inputs* chosen at the root level of the program. In addition, a sub-set of *active outputs* can be specified: the generated code computes these outputs only. Differentiating `P` as a whole function requires the knowledge of all the *active* variables at each point of the original code. More precisely variables are active when they both depend on at least one active input and impact at least one active output.

The activity propagation is a key point to obtain a minimal adjoint code. If no activity propagation is performed, any variable is associated to a derivative and each line of the code is differentiated. On the contrary, if the active/passive variables are known at each point of the program, derivatives are only associated to active ones and lines of code are only differentiated if they involve active computations. The activity propagation is performed through the call tree: from the root to all nodes. This tedious task is the second problem when coding an adjoint. It can be left to AD tools that automatically follow active variables all along the code. From the call tree and the activity information at each node, an adjoint code can be generated using one (or a mixture) of the adjoining strategies described in the next section.

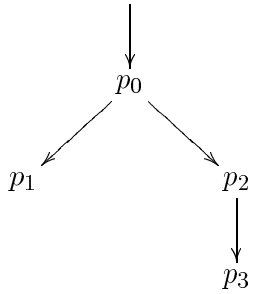


Fig. 3. Call tree of `P`.

3. Adjoining strategies

The generation of an adjoint code is based on the reversal of the original computation. As described in the previous section, this requires to store or recompute some intermediate values. In this section, two levels of storage/recomputation strategy are described: local level in Section 3.1 and global level in Section 3.2.

3.1. Local strategies

As shown in the previous section, the context of evaluation of the local (transposed) Jacobian matrices must be reproduced. Sequence `[A;B]` is chosen as an example again. The adjoint code of `[A;B]` runs the sequence `[B';A']`, where `B'` and `A'` are displayed in Figs 1 and 2. The problem is to get the correct value of variable `Y` since variable `Y` is used and overwritten. We call $\mathtt{Y}_0$ the initial value of `Y` and $\mathtt{Y}_1$ its value after execution of `B`. In order to compute correct derivatives, `Y` must be set to $\mathtt{Y}_0$ before computing the partials of `A'`. The same method applies for variable `Z`.

The local strategy generally used (via AD or hand writing) consists in memorising the values of all the original variables before modification. These values are restored at the right time to be used within the computation of the partials. The piece of code that computes the original function and stores values of modified variables is named *direct part*. The piece of code that restores values and computes adjoint variables is named *reverse part*. The direct part is necessarily run before the reverse part.

Figure 4 shows the adjoint code of `[A;B]`. The two values $\mathtt{Z}_0$ and $\mathtt{Y}_0$ are respectively stored in $\mathtt{V}_0$,$\mathtt{V}_1$ in the direct part (see Fig. 4(a)). In the reverse part (see Fig. 4(b)), the value of the variable `Y` is restored to $\mathtt{Y}_0$ by $[\mathtt{Y} = \mathtt{V}_1]$ before the computation of `B'` and the value of `Z` is restored to $\mathtt{Z}_0$ by $[\mathtt{Z} = \mathtt{V}_0]$ before `A'`. The number of values to be stored is at most one value per statement. This is the reason why this method is the standard one used by adjoint developers as well as within AD tools.

Two other strategies can be applied at the local level. The first leads to the recomputation of all the modified variables from some initial ones. This strategy yields to a quadratic growth of the execution time in terms of number of statements actually executed. A second strategy consists in computing and memorising partial derivatives [12] along the direct part, and restoring them along the reverse part. This strategy leads to a growth of required memory: for each statement the number of values to be stored is exactly equal to the number of active variables involved in the local computation. Even though these non-standard strategies are not in general the best choice, they can be applied on specific pieces of code.

3.2. Global strategies

Strategies that are applicable at the call tree level to built up the derivative program are named global strategies. Two methods are used to generate adjoint codes: the *No recomputation* strategy is used when hand coding adjoints, and the *Sub-program recomputation* strategy which is implemented in AD tools. We denote by p_i^s (where ^{s} stands for "store") the direct part of the sub-program p_i and by $p_i'^r$ (where ^{r} stands for "restore") its reverse part. Parts p_i^s and $p_i'^r$ share the trajectory local to p_i. As explained before, p_i^s computes and stores the trajectory whereas $p_i'^r$ restores this trajectory and propagates adjoint values.

The two global strategies under consideration combine these direct and reverse parts in a different way. Let consider the program `P` whose call tree is shown in Fig. 3 as an example. Figure 5 shows the call tree of the two adjoint codes of `P'` generated using the two global strategies described in this section.

No recomputation – This strategy is used for hand coding adjoints: it consists in storing intermediate calculations recursively on the call tree.
The resulting adjoint code $\mathtt{P'}_{\mathrm{NR}}$ is shown in Fig. 5(a). Each original sub-program p_i is associated to the two sub-programs p_i^s and $p_i'^r$. Since the two parts may be executed far from one another (it depends on the call tree), the trajectory is stored within global variables. The trajectory then consists of the union of the trajectories of all the sub-programs walked through during the execution of the program. It can be enormous. The adjoint variables are computed from these values taken in reverse order.
The effectiveness of such a strategy depends on the ability of the hand-coder to control the storage. Each sub-program called in $\mathtt{P'}_{\mathrm{NR}}$ is either a direct part p_i^s or a reverse part $p_i'^r$. If p_i calls p_j in `P`, then p_i^s calls p_j^s and $p_i'^r$ calls $p_j'^r$. Each original sub-program is executed once to store the trajectory and is never called again: no recomputation of the original sub-programs is necessary in this strategy.

Sub-program recomputation – The problem can be taken the other way round: the minimum quantity of intermediate values is stored. This is possible when recomputing sub-trees of the original call tree: we name it the "Sub-program recomputation strategy".
The call tree of the corresponding adjoint code $\mathtt{P'}_{\mathrm{SR}}$ is shown in Fig. 5(b). Each original sub-program p_i is associated to one sub-program $p_i' = \{p_i^s; p_i'^r\}$. Since the two parts p_i^s and $p_i'^r$ are executed consecutively, the trajectory can be stored within local variables.
Using this strategy, the trajectory consists of the union of the trajectories of all the sub-programs walked through within one branch of the call tree. Each sub-program called in $\mathtt{P'}_{\mathrm{SR}}$ is either an original sub-program p_i from `P` or a generated one $\{p_i^s; p_i'^r\}$. If p_i calls p_j in `P`, then p_i^s calls p_j and $p_i'^r$ calls $\{p_j^s; p_j'^r\}$. As a result, recomputations of the original functions are necessary: each original sub-program is executed a number of times equal to its depth in the call tree. The total supplementary execution time is in the worst case equal to δ times the original execution time where δ is the

```
   V0 = Z            !Z0
A: Z = X * Y**2      !Z1

   V1 = Y            !Y0
B: Y = X**2 * Z**3   !Y1
```

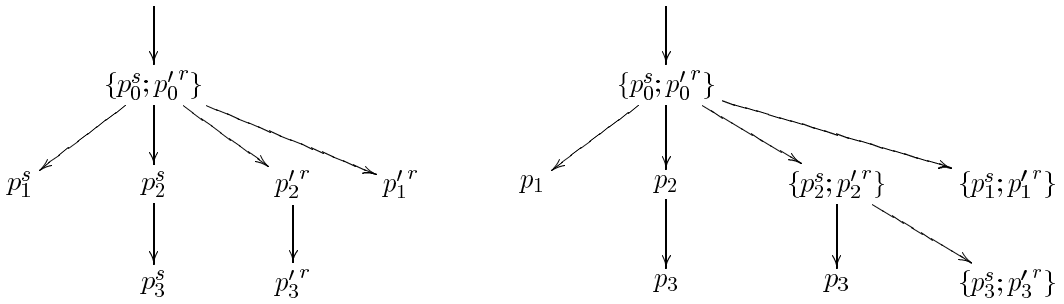
(a) direct part

```
     Y = V1                      !Y0
B': dX = dX +    2*X*Z**3 *dY
    dZ = dZ + 3*X**2*Z**2 *dY
    dY = 0

     Z = V0                      !Z0
A': dX = dX +  Y**2 *dZ
    dY = dY + 2*X*Y *dZ
    dZ = 0
```

(b) reverse part

Fig. 4. Adjoint code of `[A;B]` in direction $(\mathtt{dX}, \mathtt{dY}, \mathtt{dZ})^T$: standard strategy.



(a) No recomputation strategy (P'$_{\mathrm{NR}}$) (b) Sub-program recomputation strategy (P'$_{\mathrm{SR}}$)

Fig. 5. Call trees of `P'`.

depth of the program. Moreover, any sub-program p'_j must be run with the same input values as for p_j. As a consequence, if sub-program p_i calls n sub-programs $\{p_j, j = 1, n\}$, the direct part p_i^s (respectively $p_i'^r$) of p'_i must store (respectively restore) the context of call of all the sub-programs $\{p_j, j = 1, n\}$.

3.3. From one strategy to the other

An adjoint generated using the Sub-program recomputation strategy may be transformed into a new adjoint code that implements the No recomputation strategy and vice versa. This section discusses such transformations. The adjoint code of any program `P` created using the Sub-program recomputation strategy `P'` $_{\mathrm{SR}}$ (see Fig. 5(a)) is rewritten as a second code `P'` $_{\mathrm{NR}}$ implementing the No recomputation strategy (see Fig. 5(b)).

To generate the sub-programs necessary for `P'` $_{\mathrm{NR}}$ from `P'` $_{\mathrm{SR}}$, each sub-program $\{p^s; p'^r\}$ is split into two sub-programs p^s and p'^r. This transformation is performed in three steps:

1. $\{p^s; p'^r\}$ is split into two sub-programs: p^s and p'^r,
2. each call to an original sub-program q in p^s is replaced by a call to q^s,
3. each call to $\{p^s; p'^r\}$ in p'^r is replaced by a call to q'^r.

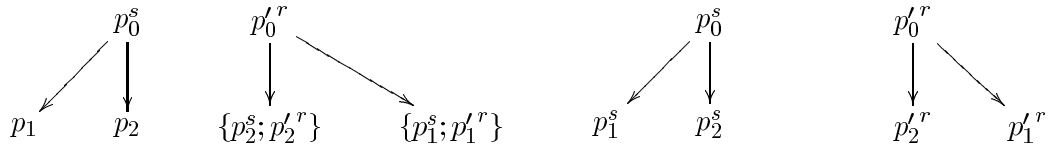
Figure 6 shows the (one level) transformation of sub-program $\{q_0^s; q'^r_0\}$ into p_0^s and p'^r_0. The result of each step of the transformation is illustrated by one sub-figure.

It is clear that if the transformation applied to $\{p_0^s; p'^r_0\}$ is applied to $\{p_1^s; p'^r_1\}$, $\{p_2^s; p'^r_2\}$, $\{p_3^s; p'^r_3\}$, the program `P'` $_{\mathrm{SR}}$ in Fig. 5(b) is changed to `P'` $_{\mathrm{NR}}$ (see Fig. 5(a)).

`Tamc` and `Odyssée` implement the Sub-program recomputation strategy whereas `Adolc` implements the No recomputation strategy.

4. Comparison of the global strategies

This section mainly describes the application of the global strategies to the thermo-hydraulic code `Thyc-1D` (developed at Électricité de France/Direction des Etudes et Recherches). The local strategy chosen is the standard one: the value of each modified variable is



(a) First step (b) Second step (c) Third step

Fig. 6. Transformation of $\{p_0^s; p'^r_0\}$ (given in Fig. 5(b)) into p_0^s and p'^s_0.

stored. `Thyc-1D` is chosen because it is large enough to allow for general observations and small enough to be modified by hand. This trial code was not written in prevision of any adjoint construction and can thus be taken as a true example.

In order to compare the influence of the trajectory management on the efficiency of the resulting adjoints, we implement and apply the common trajectory management methods. Execution time and static memory requirement of the original code `Thyc-1D` are used as references.

4.1. 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. It consists of five partial differential equations: three conservation equations for the two-phase mixture (mass, momentum and energy), one conservation equation for the vapour mass and one conservation equation for the relative liquid-vapour velocity between the two phases. To compare the different strategies, we choose to evaluate the sensitivity of the relative velocity between phases with respect to four parameters. The latter are, in particular, the relaxation time in boiling modeling and the thermal power generated in the bundle.

`Thyc-1D` consists of two different kinds of sub-programs: 13 Fortran-77 sub-programs and 12 sub-programs from EDF's compiled libraries that are differentiated using a first order finite difference scheme. It is clear that the use of finite differences certainly impacts the execution time of the adjoint code with respect to a true adjoint. This influence is the same for all global strategy and allows for fair comparisons.

4.2. Generation of the different adjoint codes

Using `Odyssée`, we have automatically generated a first adjoint code of `Thyc-1D` named `Recomp-` that uses the Sub-program recomputation strategy. Then, the `Recomp-` code is modified in order to obtain the `NoRecomp-` code using the method described in Section 3.3.

Table 1
Adjoint code generation

Name	Generation			
	Standard		Optimised	
	Method	Time	Method	Time
`Recomp-`	Auto	1 hour	Semi	2 months
`NoRecomp-`	Semi	+ 1 day	Semi	+ 1 day

The version of `Odyssée` used here applies no optimisation to diminish the storage: it stores the value of all variables before modification. It does not check for example if the value is used afterwards in the original code, or if it is used linearly or not in the derivative computation. Such optimisations have been applied semi-automatically on Meso-NH [5] to turn the adjoint `Recomp-` code to an adjoint `NoRecomp-` code. The same trajectory improvements are applied to `Thyc-1D` to obtain optimised versions of `Recomp-` and `NoRecomp-`.

Table 1 summarises the way the four codes are generated: the generation method and the generation time. We consider generation time to be: generation, compilation, run and debug of the adjoint runtimes. As all the codes are derived from `Recomp-`, the first column of Table 1 indicates the additional time with respect to this basic generation time using the symbol $+$. All the optimised code are generated from the optimised version of `Recomp-`, itself generated in two months from the code of `Recomp-`. The second column indicates (using $+$) the generation time of all the versions of the code with respect to the optimised version of `Recomp-`. Columns labelled *Method* indicate the generation method: *Auto* means fully automatically generated using `Odyssée` and *Semi* means hand modified from another code.

The functionalities of a library dedicated to the management of the trajectory are: store and restore, scalar values as well as arrays, various types (integer, logical, real, double ...). We develop three implementations of the trajectory management library: the first one labelled *Stat* is implemented in Fortran and uses global

static arrays, the second one labelled *Dyn* implemented in C uses a dynamic list of buffers, the third one labelled *File* implemented in Fortran uses direct access files. The trajectory management libraries are independent from the adjoint code.

Linking the three trajectory management libraries with one adjoint code, one gets three different adjoint versions: for example `Recomp-`*Stat*, `Recomp-`*Dyn*, `Recomp-`*File* are obtained from `Recomp-`. From the four adjoint codes, we obtain twelve versions.

4.3. Adjoint runtimes comparisons

In this section, the twelve adjoint versions described above are compared in terms of execution time and memory requirement. We have verified the correctness of the values of the computed gradients: they all agree to the finite difference results.

Table 2 (respectively Table 3) presents the execution time and trajectory characteristics of the standard (resp. optimised) adjoint codes. These values are obtained on a SPARC Station (SunOS 5.6) with `512 MegaBytes` memory and `1 GigaByte` swap.

Execution time is measured in seconds and averaged on 10 executions. The *Total* execution time is divided into two components: the *Traj*ectory management time and the *Base* execution time which is equal to *Total* minus *Traj.*. The *Ratio* of the *Total* execution time with respect to the execution time of `Thyc-1D` is presented. The *Total* execution time is directly measured from the adjoint codes. The Trajectory management execution time is difficult to measure from the adjoint codes because each move costs less than the measurable minimum execution time. To get credible measures, a code that reproduces the trajectory characteristics is used.

The trajectory management libraries compute the *Trajectory characteristics*: the *Size* of the trajectory (in `KiloByte`) and the number of trajectory *Moves* (divided by 10^3). We show two components of the *Total* number of moves: the number of *Scalar* or *Block* moves of length 100 (this length is averaged on all the blocks moves). A block move of length n is functionally equivalent to n scalar moves and is used for example to move arrays in one shot. The trajectory characteristics only depend on the global strategy chosen. Using the No recomputation strategy, each component of the trajectory memory contains the same value all along the execution. On the contrary, the Sub-program recomputation strategy implies several moves of the same value.

At a first glance to Table 2 and Table 3, one deserves that the static trajectory management *-Stat* technique is a lot faster than the dynamic technique *-Dyn* using both global strategies. The external trajectory management labelled *-File* is a bad choice in comparison to static *-Stat* and dynamic *-Dyn* management of the trajectory. Therefor, in the rest of the paper, we only consider the *-Stat* and *-Dyn* executions.

We first investigate the influence of the global strategy on the execution time and memory requirement of the adjoint codes (see Table 2).

As expected from the theory, one observes that the No recomputation strategy is minimal in terms of execution time whereas the Sub-program recomputation strategy is minimal in terms of trajectory size. The trajectory size of `NoRecomp-` is 13 times the trajectory size of `Recomp-` and the execution time is 0.87 times the `Recomp-` execution time. The theoretical *Ratio* between the adjoint execution time and the original execution time is in the worst case 5 for a straight line program. The practical ratio of 6 (or 7) obtained on the standard adjoint can then be considered really good as `Thyc-1D` is far from a straight-line code. The large benefit in trajectory size of Sub-program recomputation is obtained to the detriment of the execution time since each part of the initial code is run at least a number of time equal to its depth in the original call tree. From the theory, we know that the total extra recomputation is in the worst case $D * T$ where D and T are respectively the depth of the call tree and the execution time of the original program. The depth of `Thyc-1D` is $D = 6$ and and its execution time is $T = 7$ seconds. Therefor, the extra recomputation of the original function could be 42 seconds, but is only 9 seconds (the difference between the *Base* components of `Recomp-` and `NoRecomp-`). It is $1.3 * T$ which is a little more than one execution of the original program `Thyc-1D`. From these results, we conclude that the standard adjoints obtained semi-automatically are really efficient.

The choice between the two global strategy depends on the user key resource: execution time for real time computations or memory requirement. On some examples a ratio of 13 in trajectory size could lead to a dead lock, whereas on some others a gain of 16% in terms of execution time can be of prime necessity.

In the later, the influence of the optimisation of the trajectory is studied by comparing Tables 2, 3. The optimisations performed on the standard codes to get the optimised versions are eliminations of unnecessary trajectory components. This implies a gain in trajectory

Table 2
Execution time and trajectory characteristics of the standard adjoint versions

Code name	Execution time				Trajectory characteristics			
	Total	Traj.	Base	Ratio	Size	Moves		
					Total	Scalar	Block	
`Recomp-`*Stat*	51	5	46	7	25380	9905	9584	321
`Recomp-`*Dyn*	60	13	47	8	25380	9905	9584	321
`Recomp-`*File*	2760	2711	49	378	25380	9905	9584	321
`NoRecomp-`*Stat*	44	6	38	6	345883	10051	9708	343
`NoRecomp-`*Dyn*	53	14	38	7	345883	10051	9708	343
`NoRecomp-`*File*	2998	2959	39	410	345883	10051	9708	343

Table 3
Execution time and trajectory characteristics of the optimised adjoint versions

Code name	Execution time				Trajectory characteristics			
	Total	Traj.	Base	Ratio	Size	Moves		
						Total	Scalar	Block
`Recomp-`*Stat*	42	2	40	6	21	8063.5	8063	0.5
`Recomp-`*Dyn*	48	7	41	7	21	8063.5	8063	0.5
`Recomp-`*File*	557	517	40	76	21	8063.5	8063	0.5
`NoRecomp-`*Stat*	39	2	37	5	59128	8089	8089	0
`NoRecomp-`*Dyn*	45	7	38	6	59128	8089	8089	0
`NoRecomp-`*File*	576	538	38	79	59128	8089	8089	0

size and a reduction of the number of trajectory moves. On this example, the manual optimisation of the trajectory reduces (*a*) the number of trajectory moves by a factor of 1.3 for both strategies and (*b*) the size of the trajectory by a factor of 1200 using the Sub-program recomputation strategy and 6 using the No recomputation strategy. At the same time, the total execution time is only reduced by an averaged factor of 1.2. From these practical results, on deduces that the execution time depends more on the number of trajectory moves than on the trajectory size. This is not specific to the adjoints of `Thyc-1D` but is a general rule. This result is confirmed by the observation of the trajectory management time for `NoRecomp` and `Recomp`. Comparing the standard and the optimized adjoints, the number of trajectory moves appear to be nearly the same as well as the trajectory management time, whereas the trajectory size is absolutely different.

On our example, all the adjoint codes could be run whatever strategy is used. But as we said before, reducing the trajectory size may be the only way to run the adjoint. We conclude that (a) it is important to diminish the trajectory size to be able to run adjoint codes that could not be run otherwise and (b) it is important to diminish the number of trajectory moves to reduce the execution time. One can reduce the trajectory size by changing the global strategy, or by suppressing for example the components of the trajectory that appear linearly as we did on `Thyc-1D`. A first strategy to be applied on adjoint codes to diminish the number of trajectory moves is to group scalar or block moves into block moves of greater length. Moreover the execution time of n scalar moves costs a lot more than the execution time of one block move of length n. The combination of both effects leads to a really powerful optimisation.

5. Conclusion

Computational methods using derivatives are classical for a large number of applications involving optimal control theory such as shape optimisation or data assimilation in geophysics. The main advantage of using adjoint codes relies on the fact that the execution time does not depend on the number of input variables. 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, developing an adjoint using an 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 discuss two methods for the differentiation of sub-program: `Recomp-` and `NoRecomp-` strategies. We briefly describe trajectory management techniques. Some of these methodologies are natural when hand coding adjoints (`NoRecomp-`), whereas the others are standard when automatically generating adjoints (`Recomp-`). They are applied on a trial code large enough to allow for generalisations but small enough to be handled by hand. We show that the `NoRecomp-` strategy gives the most efficient code in terms of execution time, whereas the `Recomp-` strategy is the most efficient in terms of memory requirement. In further versions of AD Tools (source-to-source), the user should be given the choice of the global strategy (`Recomp-` or `NoRecomp-`). However, `Recomp-` codes easily translate into `NoRecomp-` codes and vice versa.

When generating adjoint codes, the fundamental challenge is the knowledge of the trajectory for the evaluation of the local Jacobian matrices. Amongst the optimisations applied by hand to limit the size of the trajectory some can be automated. Detecting linear computations as well as replacing storage by recomputation is possible using static analysis. Static analysis of Scalar values is efficient and can bit hand coding. As far as array component analysis is concerned, inter-procedural array region analysis is to be performed. Such analyses are conservative and therefore allow for further manual optimisations. On the contrary, the mathematical knowledge is difficult to extract from a source code. A lot of manual optimisations induced by this knowledge cannot be automated. Even if the gain in execution time is not proportional to the gain in trajectory size, optimising the trajectory is fundamental. The first reason is that the execution time of an elementary operation diminishes a lot quicker than the memory or disk time accesses. The second reason is that the practical efficiency of an adjoint code is proportional to the ratio between the number of elementary operations and the number of memory or disk accesses.

As a conclusion, we recommend the use of AD tools to adjoint developers, even if the resulting code has to be modified. After this automatic phase, the restructuring of the storage (if necessary) is easy to perform by hand. At least it is easier than testing the derivative statements one by one in an hand coded adjoint. The user can also optimise the storage by adding a semi-automatic post-processing step. This has been done for generating the adjoint code of Meso-NH [5] with trajectory storage in 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 very impressive.

Acknowledgement

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

References

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

[2] C. Bischof, A. Carle, P. Khademi, A. Mauer and P. Hovland, ADIFOR2.0 User's Guide, Technical Report ANL/MCS-TM-192/CRPC-TR95516-S, Argonne National Laboratory Technical Memorandum and CRPC Technical Report, 1998.

[3] J.F. Bonnans, C. Lemaréchal and C.A. Sagastizábal, Optimisation Numérique: aspects théoriques et pratiques, *Mathématiques et Applications* **27** (1997).

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

[5] I. Charpentier and M. Ghémirès, Efficient adjoint derivatives: Application to the atmospheric model Meso-NH, *Optimization Methods and Software* **13**(1) (2000), 35–63.

[6] C. Faure, Adjoining strategies for multi-layered programs, *Optimisation Methods and Software* (2000), (To appear).

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

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

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

[10] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel and A. Walther, ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++, *ACM TOMS* **22** (1996), 131–167.

[11] A. Griewank, *Principles and Techniques of Algorithmic Differentiation*, SIAM, 2000.

[12] 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.

[13] J.M. Lewis and J.C. Derber, The use of adjoint equations to solve a variational adjustment problem with advective constraints, *Tellus* **37A** (1985), 309–322.

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

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

[16] 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.

[17] O. Pironneau, *Optimal Shape Design for Elliptic Systems*, Springer-Verlag, 1984.
[18] J.-R. Roche, Gradient of the discretized energy method and discretized, *Appl. Math. Comput. Sci.* **7**(3) (1997), 545–565.
[19] Z, Sirkes and E. Tziperman, Finite difference of adjoint or adjoint of finite difference? *Mon. Wea. Rev.* **125** (1997), 3373–3378.
[20] O. Talagrand and P. Courtier, Variational assimilation of meteorological observations with the adjoint vorticity equations. Part I. Theory, *Quarterly Journal of the Royal Meteorological Society* **113** (1987), 1311–1328.