Automatic- versus Manual- differentiation for non-linear inverse modeling

David Elizondo, Christèle Faure, Bernard Cappelaere

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Automatic- versus Manual- differentiation for non-linear inverse modeling

David Elizondo — Christèle Faure — Bernard Cappelaere

N° 3981

Juillet 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3981--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Automatic- versus Manual- differentiation for non-linear inverse modeling

David Elizondo* , Christèle Faure † , Bernard Cappelaere‡

Thème 1 — Réseaux et systèmes
Projet TROPICS

Rapport de recherche n° 3981 — Juillet 2000 — 41 pages

Abstract: Emerging tools for automatic differentiation (AD) of computer programs should be of great benefit for the implementation of many derivative-based numerical methods such as those used for inverse modeling. The Odyssée software, one such tool for Fortran 77 codes, has been tested on a sample model that solves a 2D non-linear diffusion-type equation. Odyssée offers both the forward and the reverse differentiation modes, that produce the tangent and the cotangent models, respectively. The two modes have been implemented on the sample application. A comparison is made with a manually-produced differentiated code for this model (MD), obtained by solving the adjoint equations associated with the model's discrete state equations. Following a presentation of the methods and tools and of their relative advantages and drawbacks, the performances of the codes produced by the manual and automatic methods are compared, in terms of accuracy and of computing efficiency (CPU and memory needs). The perturbation method (finite-difference approximation of derivatives) is also used as a reference. Based on the test of Taylor, the accuracy of the two AD modes proves to be excellent and as high as machine precision permits, a strong indication of Odyssée's capability to produce error-free codes. Comparatively, the manually-produced derivatives (MD) sometimes appear to be slightly biased, which is likely due to the fact that a theoretical model (state equations) and a practical model (computer program) do not exactly coincide, while the accuracy of the perturbation method is very uncertain. The MD code largely outperforms all other methods in computing efficiency, a matter of current research for the improvement of AD tools. It is reckoned though that such tools can already be of considerable help for the computer implementation of many numerical methods, avoiding the tedious task of hand-coding the differentiation of complex algorithms.

Key-words: Code differentiation, optimization, adjoint state, data assimilation, Odyssée.

* IRD - UMR Hydrologie, 911 av. Agropolis, BP 5045 - 34032 Montpellier cedex 1
† Email : Christele.Faure@sophia.inria.fr, URL: http://www.inria.fr/tropics/Christele.Faure
‡ Email : Bernard.Cappelaere@mpl.ird.fr, IRD - UMR Hydrologie, 911 av. Agropolis, BP 5045 - 34032 Montpellier cedex 1

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

Différentiation automatique ou manuelle pour l'inversion de modèles non-linéaires

Résumé : Les outils nouveaux pour la différentiation automatique (DA) de programmes informatiques devraient être d'un grand intérêt pour la mise en oeuvre des nombreuses méthodes numériques qui font appel aux dérivées, telles que celles utilisées en modélisation inverse. Le logiciel `Odyssée`, un outil de DA pour les codes Fortran 77, a été testé sur un modèle résolvant une équation de type diffusion non-linéaire 2D. `Odyssée` offre les modes de différentiation direct et inverse, qui produisent respectivement les modèles tangent et cotangent. Les deux modes ont été mis en oeuvre sur notre application. Une comparaison est faite avec un code dérivé produit par différentiation manuelle (MD) de cette application. Il est obtenu par la résolution des équations adjointes associées aux équations d'état discrétisées du modèle. Après une présentation des méthodes et outils et de leurs avantages et inconvénients respectifs, les performances des codes produits par DA et MD sont comparées en terme de précision et d'efficacité de calcul (besoins CPU et mémoire). La méthode de perturbation (approximation des dérivées par différences finies) est également utilisée, à titre de référence. Sur la base du test de Taylor, la précision des deux modes de DA s'avère excellente, aussi élevée que le permet la précision machine, ce qui constitue une indication forte de la capacité d'`Odyssée` à produire des codes sans erreur. Par comparaison, les dérivées produites manuellement (MD) apparaissent parfois légèrement biaisées, ce qui provient vraisemblablement du fait qu'un modèle théorique (équations d'état) et un modèle pratique (programme informatique) ne coïncident pas parfaitement. La précision de la méthode de perturbation est, elle, très aléatoire. Le code MD est largement plus performant que toutes les autres méthodes en termes d'efficacité informatique, laquelle fait l'objet de recherches actives pour l'amélioration des outils de DA. Nous pensons cependant que ces outils peuvent d'ores et déjà être d'une très grande utilité pour la mise en oeuvre de nombreuses méthodes numériques, en évitant la tâche fastidieuse que constitue la différentiation à la main d'algorithmes complexes.

Mots-clés : Différentiation de code, optimisation, état adjoint, assimilation de données, `Odyssée`.

Contents

Chapter 1

Introduction

The progresses made in the modeling of natural systems call for an increased ability to condition these models with observed data that are both in growing amounts and of diverse nature. This conditioning, whether it consists in parameter estimation, in model initialization, in the identification of some forcing condition, or more-generally speaking, in model tuning and data assimilation, implies being able to solve inverse modeling problems: what is known is some model output and what is looked for is the part of the model's parameters and inputs that makes up its control factors. Similar questions may need to be tackled when dealing with the dynamic coupling of models for interacting physical systems. Given that models can almost never be inverted analytically, inverse modeling necessarily consists in numerically-intensive, time-consuming tasks that currently make them rather impractical in many situations. Hence, the development of methods for efficient inverse modeling appears as a key issue for many geoscience fields.

Numerical inverse modeling is generally formulated as attempting to minimize some objective diagnostic function in a multi-dimensional space (representing the control factors that are looked for), and therefore calls for powerful optimization techniques, that are capable of finding a satisfactory solution after a finite and preferably small number of function estimations. The diagnostic function reflects the agreement between observations and model outputs, as well as possible constraints on the unknowns of the inverse modeling problem. Among the many available optimization methods, some of the most efficient algorithms belong to the family of so-called quasi-Newton methods [DS96]. These require the estimate, at each trial location in the inverse-problem space, of the function's gradient vector in addition to the function evaluation. Starting from some initial guess point, an iterative procedure constructs a series of trial points until some criterion is satisfied that ensures close proximity to a function minimum. This minimum may be global or only local, depending on the function surface (i.e. on the inverse problem formulation) and on the starting point. A quasi-Newton algorithm may also be used as a local search method coupled to a more global search method in a multi-stage approach.

Producing the algorithm that computes the gradient vector from the expressions of the model and of the diagnostic function formulation, generally is an extremely heavy task. This is particularly so as the model outputs often are obtained as implicit solutions of complex linear or non-linear systems of equations, that do not easily lend themselves to the production of explicit, closed-form derivatives of model outputs with respect to parameters or state variables. Gradient- (or, all the more so, higher derivative-) -based optimization methods are often disregarded in favour of less efficient direct-search methods, because of the difficulty of evaluating the required derivatives. Since the target function is not available in closed form, it is often thought that analytical derivatives are not available. A methodology exists from optimal control theory, namely the adjoint technique [LT86, MAS96] hereafter denoted the "adjoint state method", that allows to produce the gradient vector from the solution of a linear system of adjoint equations derived from the direct simulation model. However the code for this algorithm is not built very easily, hence it is very demanding in development time, and, like any other numerical program, it is prone to development errors that are not easily detected.

Beside its use for inverse modeling and, more generally speaking, for optimization and optimal control problems, computation of derivatives to linearize complex expressions or systems of equations is useful for many efficient numerical methods, such as Newton's solution method for non-linear systems of algebraic equations. Beyond the use for assimilation of data into a model, differentiation is a very powerful approach to the analysis and understanding of model behavior: it yields the sensitivity of outputs to input perturbations or uncertainties, and allows to qualify the model's reliability (e.g.: predictability of each output), to identify crucial observations ("observation targeting"), to backtrack in the data the source of some peculiar model behavior, or even [GLVM91] to analyze the propagation of rounding errors in the model. All these methods require repeated computations of large vectors or matrices of first-, and possibly second- order derivatives. Hence, practical and accurate techniques are needed to produce these derivative objects.

Generally speaking, the computation of derivatives can be performed through various approaches, that can broadly be classified into three categories:

perturbation methods: consist in running the program twice with different values of the parameter. Dividing the difference in the outputs by the difference in the parameter gives a finite-difference approximation of the derivative;

equation-based methods: consist in differentiating the equations that make up the physical model and implementing these derivatives into a code (e.g.: adjoint state method, see Chapter 3);

code-based methods: differentiate the original code itself, instruction by instruction, to compute the derivatives of code outputs relatively to code inputs. Automatic differentiation (`AD`) belongs to this category, and is being developed to replace the dreadful task of doing it by hand.

Perturbation methods are a rather straightforward way of obtaining derivative estimates, since they make use directly and solely of the original model (i.e. the existing direct model)

and therefore do not require the development of a specific, differentiated code. This easy implementation makes them by far the most widely used technique in practical gradient-based optimization. However, aside from the problem of precision due to the numerical approximation inherent to the method, this advantage of simplicity is at the expense of considerable computational costs in case of long-running models, since at least two program runs are needed to calculate each sensitivity. For m parameters, this leads to at least $m + 1$ runs of the program or even $2m$ runs for the more accurate, central difference approximation. On the opposite, equation-based methods entail rather heavy manual work, both on the mathematical and computer programming sides, but allow for the development of very efficient, computationally optimized codes. Code-based methods are an attractive alternative since their closeness to the original model's computational algorithm ensures consistency and accuracy of the associated derivatives, and since they open the way for automation of the differentiation process.

Because of the widespread need for differentiation of large numerical algorithms, efforts have been made in recent years towards the development of software tools that could automate the production of required derivatives for any given numerical application, through code-based methods. Examples of such tools are Adifor [BCK+], Tamc [Gie97] and `Odyssée` [FP98] for Fortran 77 source codes. Tools also exist for other languages such as Fortran 90, C or C++ [GC91, BBCG96]. The challenge of any differentiation methodology is to obtain the derivatives with as few manual interventions as possible, in such a way that the resulting code will be efficient with respect to CPU time, sufficiently modest on its memory needs, and will provide derivatives that are accurate enough with respect to the needs of the numerical method for which they are required. The work presented here is a comparison of a code-based `AD` tool, with a manual equation-based method (`MD`), as well as with the finite-difference approximation. INRIA's `Odyssée` (v1.6) automatic differentiator was chosen for that purpose, and a time-independent, non-linear, surface-modeling program is used as the sample case for that comparison. The report is organized as follows: Chapter 2 describes the direct and inverse problem used as a sample case to compare the three differentiation methods. The `MD` method ("adjoint-state" method) is presented in Chapter 3. Chapter 4 briefly reviews the principia `AD` is based on, and the main characteristics of `Odyssée`. Chapter 5 compares the performing of the methods, followed by conclusions in Chapter 6.

Chapter 2

The sample case: surface modeling and inverse problem description

The sample model used for this comparison is a surface modeling method based on the numerical solution of a 2D non-linear elliptic partial-differential equation. This surface modeling method is currently under development with the purpose of combining precise interpolation capabilities with a high degree of flexibility in the form of spatially distributed control over the surface. The description and discussion of the modeling method itself are beyond the scope of this report, and will be presented in a separate paper prepared by the last two authors. A leading idea behind the method is to be able to merge information of different natures, such as actual quantitative point observations of the variable being modeled and indirect spatial information on surrogate quantitative or qualitative variables that are related with the target variable. For instance, in geographical areas where topographic data are scarce, such as in developing countries, it should be possible to produce digital elevation models, that are more reliable and more meaningful for various applications (e.g.: hydrology) than those currently available, if these data were combined with geomorphologic information that reflect the characteristics of dominant landform patterns that prevail in the area. Mapping of geomorphologic features can be translated into parameter zonation for the surfacing model. Applying the model to a particular problem then becomes a question of optimum parameter identification and data assimilation into the modeling structure.

The basic framework used for this model is a non-linear diffusion type of equation, which writes:

$$div(D(p_i, Z)\|\nabla Z\|^{n-1}\nabla Z) + S(p_i, Z) = 0 \tag{2.1}$$

where Z is the altitude, $n \geq 1$ is a real number and D and S are two functions of m parameters $\{p_i, i = 1, m\}$ which are space-dependent through the user-defined geomorphologic zonation. Inner and outer boundary conditions are applied, either as Dirichlet or Neumann conditions. Equation 2.1 models the Z surface as if it were mapping a 2D potential field that

is producing and submitted to a conservative, anisotropic, non-linear diffusive flux. It may be very highly non-linear since there is no upper bound on n. The equation's non-linearity is also due to the coefficient D being a function of Z.

Discretization of the PDE Equation 2.1, using a 9-point finite-difference scheme, produces a non-linear system of equations, noted $E_m(Z,p) = 0$ where p designates the parameter set. This non-linear discrete system makes up our direct model M, implemented into a Fortran 77 program through a Newton-Raphson iterative solution procedure. The system is linearized by a first-order Taylor expansion around any given state Z_k:

$$E_m(Z_k + dZ, p) = E_m(Z_k, p) + \frac{\partial E_m}{\partial Z}(Z_k, p) \bullet dZ \tag{2.2}$$

where $R_k = E_m(Z_k, p)$ is the vector of model residuals at $Z = Z_k$; $\frac{\partial E_m}{\partial Z}(Z_k, p)$ is the system's Jacobian matrix with respect to the state variable, $J^z_{E_m}$, evaluated at state Z_k; dZ is a vector of Z increments, and the symbol $\bullet$ represents the matrix-vector or matrix-matrix product. At each Newton-Raphson iteration the tangential linear model:

$$\frac{\partial E_m}{\partial Z}(Z_k, p) \bullet Z = -E_m(Z_k, p) \tag{2.3}$$

is solved using the LAPACK library's large sparse system solver [ABB+95]. The final solution is noted $Z_p = M(p)$. Because the LAPACK routines raise some problems for `AD`, they were substituted with less efficient subroutines from the Numerical Recipes library [PTVF92] for the purpose of the present exercise. All results presented here were therefore obtained with the latter library.

Diffusion-like mathematical techniques are sometimes used for surface modeling, such as smoothing by the Laplacian operator. The much more general, parameterized mathematical formulation used here aims at taking much further the modeling capability of a diffusive flux analogy. Its application largely depends on the possibility of efficient data assimilation into the model. Generally speaking, the inverse problem can be described as the determination of all or part of the model's controls, i.e. its parameters (p_i) (the exponent parameter n and even some of the boundary conditions could also be included, although not considered here), given Z_{obs} a set of observations of the state variable. The inverse problem is thus expressed through the introduction of a suitable diagnostic criterion C that is an explicit function of the parameters p and of the associated discrete 2D distribution Z_p, such as:

$$C = F(Z_p, p) = F_z(Z_p, Z_{obs}) + F_p(p)$$

where the F_z component is a measure of the agreement between modeled and observed values of Z (for instance the commonly used sum of squared differences), and F_p is an optional regularization term introduced to enforce some a priori knowledge on the parameter set p and account for the likelihood of a given p value. The overall application from p to C is called A: $C = A(p)$. Figure 2.1 summarizes the variables and functions making up this direct problem (note that the F_p component of the function F is only represented as a

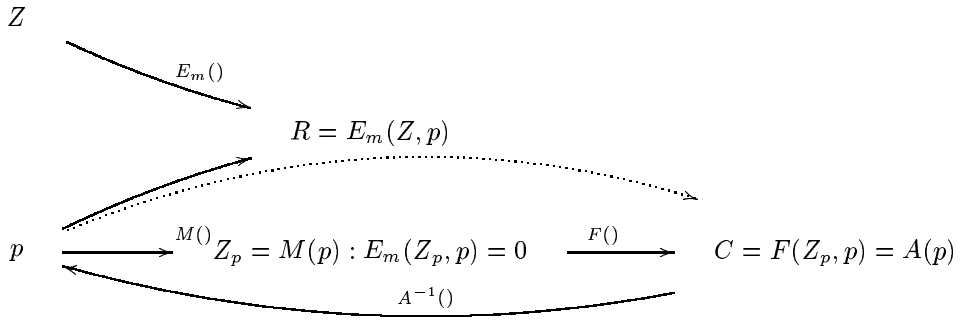

Figure 2.1: Sample direct and inverse modeling problem

dotted line in the figure). Inverse modeling here consists in finding p such that a condition of optimality be satisfied on C.

To make up a sound and viable enterprise, the formulation of this inverse problem, in terms of actual parameterization, observation distribution, diagnostic function expression, etc., must of course satisfy certain conditions for well-posedness, that translate into the response function $A(p)$ surface's shape, which cannot be addressed in the context of this paper. For the sake of testing the model differentiation approaches, a simple, uniform parameterization of D and S as a linear function of the state variable Z is applied, leading to a vector p of 8 model parameters. The inverse problem solution is obtained by finding a minimum of the diagnostic variable C using a gradient-based quasi-Newton optimizer that iteratively calls for estimation of the C value and of its gradient vector $\nabla_p C \equiv (\frac{{}^t\partial C}{dp}) \equiv {}^tJ_A$ in the inverse problem space (i.e. with respect to p). INRIA's M1QN3 library [GL95] is used for this purpose. Written as a set of Fortran subroutines, it is a very efficient module to solve high-dimensional unconstrained minimization problems. The algorithm is based on a variable-storage BFGS method [DS96], where the descent direction is determined from the supplied gradient vector together with an updated approximation of the Hessian matrix, and a new point is chosen along this direction through a line-search procedure. Fast convergence is achieved by the method while at each iteration only the cost of estimating the function value and gradient vector is incurred.

Chapter 3

Manual differentiation (MD)

The equation-based, adjoint-state method was used for the "manual" production of the problem's gradient vector. Given that the inverse problem consists in minimizing the diagnostic variable $C = F(Z,p)$ under the constraint of a simulation model M (i.e. $E_m(Z,p) = 0$), the adjoint-state method can be defined simply using the Lagrangian of that constrained optimization problem:

$$L_\gamma(Z,p) = F(Z,p) + < {}^t\gamma, E_m(Z,p) >$$

where γ is a vector of Lagrange multipliers in the space of the model M's residuals, $R = E_m(Z,p)$; and $< .,. >$ is the dot product. For any perturbation directions δp and δZ, the differential of L_γ is:

$$\delta L_\gamma = \frac{\partial L_\gamma}{\partial p} \bullet \delta p + \frac{\partial L_\gamma}{\partial Z} \bullet \delta Z.$$

If the constraint linking Z and p, i.e. $E_m(Z,p) = 0$, is satisfied (i.e. $Z = Z_p$), then for all γ: $C = L_\gamma$ and $\delta C = \delta L_\gamma$. If γ is chosen such that

$$\frac{\partial L_\gamma}{\partial Z} = 0, \tag{3.1}$$

then $\delta C = \frac{\partial L_\gamma}{\partial p} \bullet \delta p$. Hence the gradient vector $\nabla_p C$ is obtained as $\nabla_p C = \frac{{}^t\partial L_\gamma}{\partial p}$, provided that γ satisfies Equation 3.1. Therefore, the gradient can be calculated as:

$$\nabla_p C = \frac{{}^t\partial F}{\partial p} + \frac{{}^t\partial E_m}{\partial p} \bullet \gamma \quad i.e.: \quad \nabla_p C = \frac{{}^t\partial F_p(p)}{\partial p} + \frac{{}^t\partial E_m}{\partial p} \bullet \gamma \tag{3.2}$$

once γ has been computed as the solution of the linear system:

$$\frac{{}^t\partial F}{\partial Z} + \frac{{}^t\partial E_m}{\partial Z} \bullet \gamma = 0 \quad i.e.: \quad \frac{{}^t\partial E_m}{\partial Z} \bullet \gamma = -\frac{{}^t\partial F_z}{\partial Z}(Z, Z_{obs}) \tag{3.3}$$

for $Z = Z_p$. The system of equations Equation 3.3, $M_F^\star$, is called the adjoint of M with respect to the diagnostic function F, whose solution γ is the adjoint state variable, or adjoint

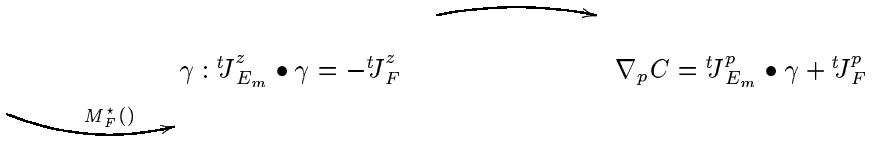

Figure 3.1: Equation-based, adjoint-state manual differentiation method (`MD`).

variable. It can be noted that the adjoint variable depends only on the simulation model M and on the expression of F_z in the diagnostic function, not on the actual choice of controls that are to be identified: indeed, there is no F_p and no differentiation with respect to p in the adjoint system Equation 3.3. This system would be the same should the controls include the exponent n, boundary conditions, or initial conditions in the case of a time-dependent problem. Hence, once γ is known from Equation 3.3, the gradient of the diagnostic function can be computed by Equation 3.2 with respect to any selected control p.

Figure 3.1 summarizes the equation-based adjoint-state `MD` method. The adjoint problem $M^\star_F$ is defined as a linear system of equations that is co-tangential to the original non-linear equation system E_m in the state variable space, at the point Z_p that was found to be the solution of the direct problem for a given parameter set p. It has the same dimension as the original discrete model, the system's matrix being the transposed Jacobian ${}^tJ^z_{E_m}$ of the original model equations E_m with respect to the state variable Z. The second member of Equation 3.3 reflects the discrepancies between available observations and simulated Z values. Once the adjoint linear system has been solved for the adjoint vector γ, the gradient vector $\nabla_p C$ is obtained as the product of the transposed Jacobian of the original equations E_m with respect to the parameters p, ${}^tJ^p_{E_m}$, by the adjoint variable vector γ, augmented with the partial derivatives of F with respect to these parameters, if applicable.

It can be seen that `MD` requires a number of differentiation operations (namely: partial derivatives, with respect to both all state variables and all parameters, of the non-linear direct model equations E_m and of the diagnostic function expression F) that are rather demanding of algebra and programming manpower time. However a great advantage is found in the present case in terms of both human time and computing time savings, in the fact that the model Jacobian $J^z_{E_m}$ is used for the solution of the original model (Equation 2.3) as well as, in transposed form, for the production of the adjoint variable, Equation 3.3. In terms of new code development, only the model's partial derivatives with respect to the parameters (matrix $J^p_{E_m}$) need to be generated, in addition to the much simpler derivatives of the diagnostic function, J^z_F and J^p_F. Also, while several Newton-Raphson iterations are needed to solve the non-linear original model, each requiring a new linear system to be constructed and solved (Equation 2.3), only one linear system solution is needed to compute the gradient with the adjoint method (Equations 3.2 and 3.3), a source of great efficiency for this manually built algorithm. This explains why, as will be seen, the equation-based `MD` method is so much faster than any other method. Code-based methods necessarily abide by the same full computational path as the original, direct model, this path being named "trajectory".

When the problem being modeled includes the time dimension, the adjoint system needs to be solved counterclockwise, because the transposition operator ^{t} transforms a direct system matrix which is lower triangular (block lower triangular if the state variable is multidimensional at each time step) into a (block) upper triangular adjoint matrix. This means that when computing the adjoint state pertaining to a given time, the whole direct problem must have been previously solved over the full simulation period, and the direct state for that particular time must still be available. This problem of restoring "trajectories" will be discussed in more detail in Chapter 4. It must be emphasized that while the equation-based adjoint-state method cannot circumvent the need to keep or restore the state-variable trajectory over time for an evolution problem, it does avoid restoring the full direct-algorithm trajectory, contrarily to a code-based adjoint method (see Chapter 4), regarding the iterative process possibly needed to solve a particular set of non-linear direct equations at any time step: indeed, only the actual state variable solution values at every time step are needed to build the adjoint problem and solve for the adjoint variable.

To conclude with, it is emphasized that the `MD` method used here is of the equation-based, not code-based, type, and more precisely that it differentiates the discretized, finite-difference equations making up the actual original model M, not the fundamental partial-differential equation which M approximates. This means that the gradient computed by this method pertains to the solution of this set of discrete, non-linear equations, which may slightly differ from the theoretical solution of the fundamental PDE as well as from the one actually computed by the direct model's coded algorithm (Fortran program).

Chapter 4

Automatic differentiation (AD)

AD has emerged as a necessity to avoid both the pain of hand-coded differentiation and the drawbacks of numerical approximations, which are expensive in CPU time and contain approximation errors. Several AD software tools are being developed by the applied-math and computing research community, all having both common and distinct features, regarding language(s) supported (Fortran 77, Fortran 90, C, C++), order of derivatives produced, derivation modes (forward and reverse, see hereafter), and current limitations. Information on these various tools is available in [FN99]. For more extensive background information on AD, see [GC91, BBCG96]. This section first briefly presents the basic concepts of AD (Section 4.1), then the main features of the Odyssée tool (Section 4.2), and finally the current limitations both of AD in general and of Odyssée in particular (Section 4.3).

4.1 AD concepts

AD is meant to produce the derivatives of the outputs from a code unit, or from a group of code units, with respect to all or part of its inputs, designated as "active" input variables. It therefore requires that the original code be piecewise differentiable, and exploits the fact that any piece of computer program basically executes a finite sequence of assignment statements and elementary arithmetic operations (after expression expansion and introduction of temporary variables). Each of these elementary statements may be viewed as a simple function f_k that leaves unchanged all the program variables except the one modified by the statement. Hence, the whole program amounts to a composition of these functions: $f(x_0) = (f_q \circ f_{q-1} \circ \cdots \circ f_2 \circ f_1)(x_0)$, x_0 being the initial value set of the program variables, at the entry into the piece of code being considered. AD is based on repeated application of the chain rule on this function composition. If one calls J_k the Jacobian matrix of f_k at the entry point $(f_{k-1} \circ f_{k-2} \circ \cdots \circ f_1)(x_o)$, then the Jacobian matrix of f at point x_o is:

$$J = J_q \bullet \cdots J_2 \bullet J_1 \tag{4.1}$$

(matrix product of the local Jacobians of the elementary functions, computed right to left, from 1 to q), and its transposed Jacobian is:

$$^tJ = {}^tJ_1 \bullet {}^tJ_2 \cdots \bullet {}^tJ_q \tag{4.2}$$

(right to left, from q to 1). In practice, local Jacobians have variable dimensions, depending on the distribution of variables in the various code segments.

Two approaches to `AD` are possible: the forward and reverse modes, which are distinguished by how the chain rule is used to propagate derivatives through the computational flow. The forward mode follows the original code's logic in the forward direction (top to bottom, after expansion) and therefore implements the first above matrix computation formula (Equation 4.1), producing the so-called "tangent" derivatives. The reverse mode (also called "adjoint" mode because it amounts to the adjoint method applied to an explicit, closed-form problem) computes the "cotangent" derivatives by following the code's logic in the reverse direction (bottom to top), according to the second formulas (Equation 4.1 or Equation 4.2). These two modes are hereafter abbreviated `tAD` and `cAD`, respectively. If f is a function to be differentiated with respect to many active input variables but with few outputs, it can easily be shown that the number of arithmetic operations performed by Equation 4.1 (roughly proportional to the active input dimension n_{in}) is much higher than by Equation 4.2 (roughly proportional to the output dimension n_{out}), and vice-versa. Hence `cAD` is theoretically more efficient in computing the Jacobian of f if $n_{in} > n_{out}$. This is particularly so when $n_{out} = 1$, i.e. when f is defined from $\Re^{n_{in}}$ to $\Re$ as in the case of inverse modeling.

In fact, `AD` is generally implemented as the computation of the product of one of the above formulas (Equation 4.1 or Equation 4.2) with a vector, not of the full Jacobians (this way, a single differentiation operation is associated with each direct program statement). Hence, `tAD` computes the directional derivative $J \bullet u$ in a particular direction u of the active input space, whereas `cAD` computes ${}^tJ \bullet v$, the gradient (in the active input space) of a combination v of the output variables. When n_{out} is 1, the Jacobian amounts to the gradient of the single output variable, therefore a single run of the differentiated code in reverse mode is sufficient to produce all derivatives, whereas n_{in} passes are necessary in forward mode, each producing one derivative. In theory, the computational cost of one run is more or less of the same order of magnitude as that of the original direct code, however this holds true only for the differentiation operations proper, and does not account for `cAD`'s substantial need for multiple recomputations of the direct code's trajectory segments (i.e. series of intermediate states: $f_1(x_o), f_2(f_1(x_o)), ...$). This important point is discussed in Section 4.3.

Two classes of `AD` tools may be distinguished, working respectively by operator overloading or by code transformation. `Odyssée` belongs to this dominant, second category, whose advantage is that the differentiated code can then be integrated into any user application, such as an optimizer. Given the two differentiating modes' respective processing methods, it can be seen that the code generated by `tAD` can be used to also produce the output of the direct original code (therefore no longer required, for inverse modeling), whereas `cAD` returns the direct variables with their initial values, making a distinct call to the direct code necessary.

4.2 A brief description of the Odyssée automatic differentiator

The Odyssée software [RDG93, FP98] is an AD tool for Fortran 77 codes, developed by INRIA (see http://www-sop.inria.fr/safir/SAM/Odyssee/odyssee.html for detailed information). It implements both the forward and the reverse differentiation modes, and is able to differentiate a whole application, consisting of a set or subset of program units (subroutines or functions, no main or block-data) whose call-graph forms a tree, defined by its root called the head-unit. The user specifies the head-unit, the differentiation mode, and the active input variables of the head-unit with respect to which he wants the application to be differentiated. These input variables may be either formal arguments of the unit or global variables in a common block. Odyssée produces a new Fortran 77 code having the same unit structure as the initial code, which computes both the application and one directional derivative of it (i.e. a tangent linear application) in forward mode, or a gradient (cotangent linear form) in reverse mode. Odyssée builds the graph of variable dependencies over the whole application, and identifies all intermediate active variables, i.e. those whose values depend on the active input variables. All the active output variables (formal arguments or global variables) of the application will be differentiated (not user selectable).

To each active symbol of a unit, Odyssée associates an additional derived symbol having the same status (argument, global, or local). Variable names are suffixed with ttl for the forward mode and ccl for the reverse mode, while the suffixes tl and cl are used for the unit names. If the active input and output variables of the original application (or of any program unit) are respectively noted I and $O = f(I)$, the tangent linear code ftl computes $O = f(I)$ and the directional derivative of O, $Ottl = f'(I) \bullet Ittl$, at point I in direction $Ittl$: $(O, Ottl) = ftl(I, Ittl)$. The cotangent code fcl updates the dual (=adjoint) variables $Iccl$, from the initial input variables I and dual variables $Iccl$ and $Occl$: $Iccl = Iccl + fcl(I, Occl)$ (owing to the transposition, $Occl$ becomes input, while $Iccl$ is both input and output). For instance, if a program unit named sample has two active input arguments (or globals) U and V and two active outputs Y and Z, X being an inactive variable for code differentiation (i.e. an input variable with respect to which differentiation is not desired, or an inactive output), and if it is assumed that Y depends on U and V, and Z on V alone, then the differentiated code will schematically write:

forward mode

$$\begin{aligned} Yttl &= \frac{\partial Y}{\partial U} \cdot Uttl + \frac{\partial Y}{\partial V} \cdot Vttl \\ Zttl &= \frac{\partial Z}{\partial V} \cdot Vttl \end{aligned}$$

reverse mode

$$\begin{aligned} Uccl &= Uccl + \frac{\partial Y}{\partial U} \cdot Yccl \\ Vccl &= Vccl + \frac{\partial Y}{\partial V} \cdot Yccl + \frac{\partial Z}{\partial V} \cdot Zccl \\ Yccl &= 0. \\ Zccl &= 0. \end{aligned}$$

A more detailed example of code produced by `Odyssée` is provided in the appendix. It can be seen that the original program control structure, at the instruction-block level, is fully preserved in the forward mode: the tangent code basically consists in an augmented original code where each original statement is preceded by a new, associated statement for the derivative computation. The cotangent code is much more elaborate, owing to the need for restoring the trajectory followed by the original code when proceeding with the backward evaluation of the transposed Jacobian matrix. This trajectory consists in all the values of the variables computed all along the direct computational path. The strategy used in `Odyssée` is to recompute the trajectory locally within each program unit call. Hence, the body of each cotangent unit is composed of a first half that recomputes the local trajectory by replicating the original code and saving all successive symbol values in new local variables used as temporaries, and of a second half which computes the derivatives in the reverse direction while progressively restoring backwards the values of the original variables: a cotangent unit call returns all original symbols unchanged.

When calling the generated code, i.e. the new derived head-unit, the additional arguments and global variables associated to the active input/output variables of the original head-unit must have been properly initialized, in accordance with the differentiation mode: for the tangent code, *Ittl* set to the chosen differentiation direction (e.g.: all null values but one equal to 1, to obtain a column of the application's Jacobian, or one gradient component if the application's output is a single real); for the cotangent code, *Iccl* set to 0 and *Occl* to the chosen linear combination of output variables (e.g. all null values but a single 1 to obtain a Jacobian line, which is the whole gradient if the output is a scalar). It is emphasized that *Occl* values are changed to 0 upon return to the calling program.

Compared to other `AD` tools, the key features of `Odyssée` are: implementation of both the forward and the reverse modes of differentiation; multi-procedural capabilities (few existing tools are able to differentiate an entire application, particularly in reverse mode; in this respect, we do not know of an alternative for Fortran 77 source codes); support of black-box units, whose source code is not available; user interaction by a command language or a graphic interface. To our knowledge, `Odyssée` is the only tool that has so far been used for `AD` experiences in reverse mode with very large, operational applications [CG97, Fau98]. The other `AD` tools that support this mode (e.g.: AdolC, Tamc, Padre2) cannot yet be applied to very big codes. `Odyssée`'s capabilities are also being extended to support message-passing parallel programs (MPI library). Figure 4.1 schematizes the application of `Odyssée`'s forward and reverse differentiation modes to our sample problem. All results presented in this paper have been obtained with `Odyssée` v1.6. The version 1.7 is to be released soon, its main difference with v1.6 being the addition of the "flow reversal" algorithm, an alternative method for reverse mode differentiation to the basic algorithm (which has been named "syntax reversal" in v1.7).

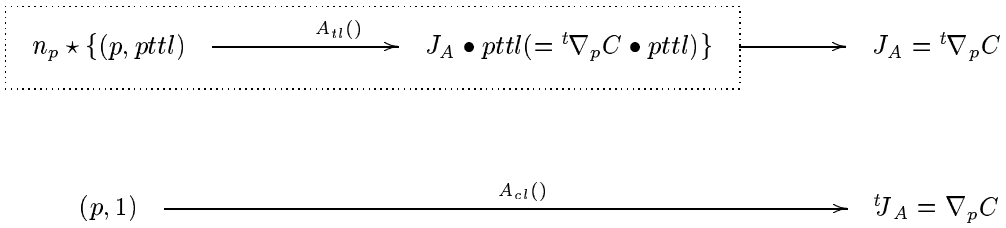

Figure 4.1: `Odyssée`'s differentiated codes: forward A_{tl} and reverse A_{cl} modes.

4.3 Current limitations of `AD` and of `Odyssée`

`AD` tools, be it `Odyssée` or any other existing package, are still very young programs that are rapidly evolving. As such, they obviously show a number of limitations, some of which will probably soon be removed. Hereafter are examples of restrictions or shortcomings that were encountered in the application of `Odyssée` to our sample case, most of which are duly reported in the `Odyssée` User's Reference Manual [FP98], while some others were noticed in the course of our work. Some of `Odyssée`'s current limitations relate to a few Fortran 77 legal constructs not presently supported by `Odyssée`, or not correctly dealt with (the user has to verify the generated code for such occurrences). Examples are: equivalence statements, recursive procedure calls, heterogeneous common block declarations, save or data statements for active variables (local variables are not considered as input or output to a subroutine), function passed as an argument to a subroutine (external), assumed-size arrays, side effects of function calls in if statements. Comments are lost from the source code, while I/O statements (read, write, format) are commented out (until a future version); include files are inlined in the differentiated code. Commons must be declared explicitly in all the routines that are in the calling path to a unit where its variables are referenced, in order for `Odyssée` to correctly build the tables of input/output variables and dependencies for all program units.

Some restrictions are specific to the `cAD` mode: branching instructions such as goto or code-jumping return (both enabled with the new alternative "flow reversal" algorithm of version 1.7), and multiple access points into a procedure (entry) are not allowed, while instructions stopping program execution (stop, pause) are forbidden within differentiated subroutines. An intrinsic difficulty with the reverse mode, for all `AD` tools that support it, is the question of the trajectory restoration, which involves very significant increases in core memory size (or disk space if saving in files) and in wall-clock run time (recomputations, I/O if disk saving), compared to the original code execution. With `Odyssée`, the local trajectory recomputation within each program unit means that each subroutine call of the original program produces two calls in the differentiated code : if unit A of the original code calls unit B, unit Acl will call B during its trajectory computation and then Bcl during

its differentiating phase proper. Within Bcl, the local trajectory will again be recalculated, thereby duplicating the computations previously done by the call to the direct routine B. This means a combinatorial increase of computation volume with depth of the calling tree. In order to alleviate this problem, [CG97, Cha98] proposed a method that reduces trajectory computations and saves in direct access files. Similar ideas were developed by [FD98]. The use of checkpoints is a technique that allows a trade-off between saving in memory and recomputing [Gri92, Cha98]. A checkpointing facility is available in `Odyssée` ("optimal reverse mode"). All these techniques are the subject of on-going research by several teams.

Saving the trajectory using additional local variables produces another inconvenience for the user: when arrays are needed for this purpose, with dimension declarators that may not be taken among the original code's local parameters, or that `Odyssée` cannot easily derive from such existing parameters, new arbitrarily-valued parameters named "ODY*" are used, that generally need to be reassigned larger values by the user. Beside the inconvenience of such manual operations, this may seriously inflate the memory requirements since the user himself often has little choice but to oversize the array dimensions. It should be pointed however that this need for user-managed memory allocation parameters disappears in the new, "flow reversal" alternative algorithm proposed by version 1.7 for `cAD` mode: this method uses a pile structure (constructed during the trajectory computation phase and un-piled along the reverse differentiation phase) to manage the trajectory variable values as well as the flow path between elementary blocks of code instructions. This transformed flow path management is performed through the use of a specific integer block pointer, and makes reading of the generated code much more difficult.

In order to limit as much as possible the prior code modifications necessary before processing by an `AD` tool such as `Odyssée` (no need for manual preprocessing is unlikely), some programming rules should be followed beyond the few syntax restrictions quoted above, particularly with respect to reverse differentiation. These rules notably pertain to the way subroutine arguments are managed: for instance, using implicit pointers or aliasing through argument passing should be avoided. More generally speaking, it can be said as a general rule that a good code for `AD` is one that does not hide in any way the dependencies between variables.

Chapter 5

Comparison of differentiation methods on the sample problem

The purpose of this chapter is to compare, in terms of accuracy (Section 5.1) and of computing efficiency (Section 5.2), the differentiated codes `tAD` and `cAD` respectively produced by `Odyssée`'s forward and reverse modes for our surface-modeling sample application, with the `MD` code. In both subsections, the finite-difference approximation (perturbation method) is used as a reference to analyze the performances of the three differentiated codes (`MD`, `tAD`, `cAD`). All codes are run in double precision.

5.1 Accuracy

For nearly all trials that have been made with the three differentiated codes, the computed gradients are essentially the same, however for a few cases slight differences have been observed. Therefore, the accuracy of these computations is investigated hereafter, using a simple test case for which such a discrepancy is obtained. This case consists of 8 input parameters p_i ($i = 1$ to m with $m = 8$) on a 10×10 space grid with Dirichlet boundary conditions, and $n = 1$. Table 5.1 shows the values of the gradients with respect to these parameters in the three differentiation modes.

It can be seen that the derivatives computed by `tAD` and `cAD` have between 10 and 11 significant digits in common, against only between 2 and 4 when compared with `MD`. Figure 5.1 shows, for each parameter ($i = 1, 8$; horizontal axis), the relative differences between methods (vertical axis, logarithmic scale): the upper curve shows `tAD` (or `cAD`) and `MD`, the lower one shows the comparison between `tAD` and `cAD`. This figure illustrates the much better agreement between the two automatic modes (lower curve: differences close to machine precision) than with `MD` (upper curve: differences in the order of 10^{-3}).

i	MD	tAD	cAD
1	−239.66663416556	−240.13622203918	−240.13622204033
2	539.25487620235	540.31145861541	540.31145861799
3	−47.851450017157	−47.837523573843	−47.837523573086
4	107.66675071374	107.63541592869	107.63541592699
5	29.860639684194	29.912187879136	29.912187878957
6	−452330.15035587	−453111.00445059	−453111.00444789
7	8.1106014306548	8.1168627075904	8.1168627077914
8	−122859.71109140	−122954.55716194	−122954.55716498

Table 5.1: Gradients of diagnostic function with respect to 8 input parameters, for 3 differentiation modes.

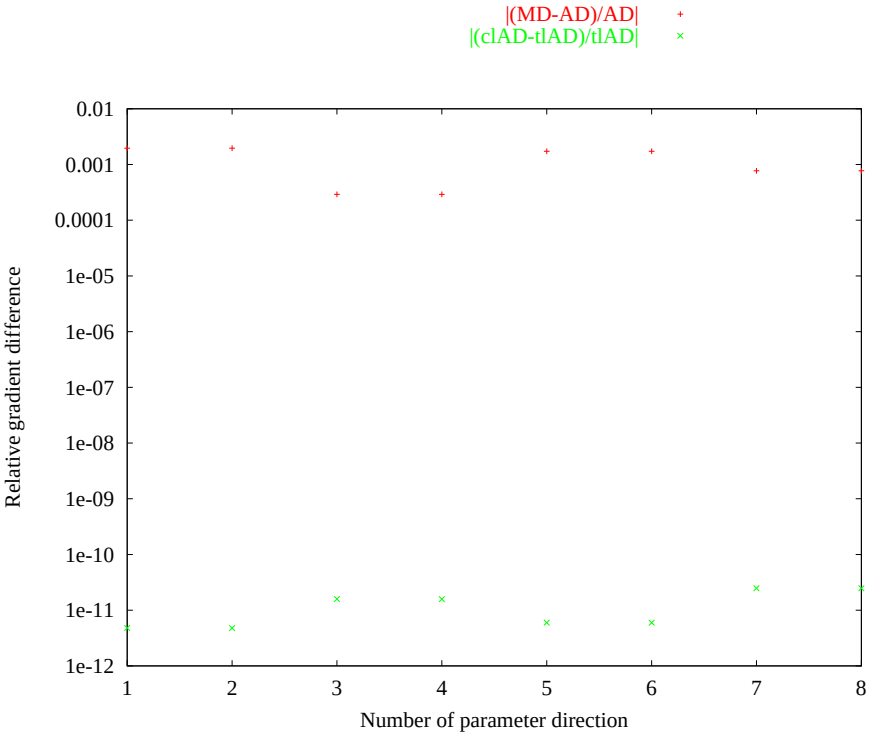

Figure 5.1: Relative gradient differences: manual MD vs. automatic AD (upper curve), and forward tAD vs. reverse cAD (lower curve), for each parameter direction.

Given the above discrepancy, a natural question that arises is whether the difference between `MD` and `AD` is due to numerical errors (caused either by the numerical schemes used, or by computer rounding) or to some programming mistake in any of the codes. With this aim, the Taylor test, based on the finite-difference approximation, has been applied to all three differentiated codes. A function A that produces a scalar output from the input vector p:

$$A: \begin{array}{ccc} \Re^m & \rightarrow & \Re \\ p & \rightarrow & A(p) \end{array}$$

may be expanded around p for a perturbation ϵ in the direction δp:

$$A(p+\epsilon\delta p) = A(p) + \epsilon < \nabla A(p), \delta p > + o(\|\epsilon\delta p\|2)$$

The Taylor test of the gradient is satisfied if it is possible to verify for any direction:

$$\lim_{\epsilon \to 0} \frac{A(p+\epsilon\delta p) - A(p)}{\epsilon < \nabla A(p), \delta p >} - 1 = 0$$

with a convergence rate of order ϵ. This test was performed successively in each basic direction of the input space, i.e. for each of the 8 input parameters. From Table 5.1 it can be seen that the components of ∇A in the various directions have quite different magnitudes, i.e. that the sensitivity of A to a perturbation is highly variable from one direction to another. In order to homogenize the Taylor test in all directions, the perturbation in each direction i is weighted by the inverse of the corresponding gradient component, and by $A(p)$ to make ϵ non-dimensional: $\delta p_i = \frac{A(p)}{\nabla A(p)_i}$.

Figure 5.2 presents the responses to the Taylor test of `MD` and `AD` (`tAD` and `cAD` are practically indistinguishable), for perturbations around the same parameter set previously used, in one parameter direction. The response to the test is nearly insensitive to the direction of perturbation. With `MD` the test heads towards 0 until ϵ reaches a value close to 10^{-4}, then it slightly degrades before remaining flat at a value of the order of 10^{-3}. This is not the case for `tAD` and `cAD`: convergence occurs until ϵ decreases to 10^{-8} (for which a test value of 10^{-7} is reached) before the test starts to degrade. In order to verify that this final degradation is due to rounding errors, this behavior of the Taylor test was compared with that obtained for the numerical computation of a simple analytical function and of its known derivative: the $cos(p)$ function was used for this purpose, around the working value $p = 1$. It can be seen (Figure 5.2, blue line) that the behavior is similar to that of the `AD` codes, with the same ϵ threshold of 10^{-8}. This threshold therefore appears as being a computer precision limit for that test. The rather constant difference between the two curves can be explained by differences in second-order terms between the sample model and the cosine function. Table 5.2 presents the actual numerical values obtained for the Taylor test with the three differentiation modes in a single parameter direction, together with those produced by the cos(p) function.

It can be concluded from these results that the gradient vector is estimated by the two code-based `AD` methods with as much accuracy as machine precision permits. This is not

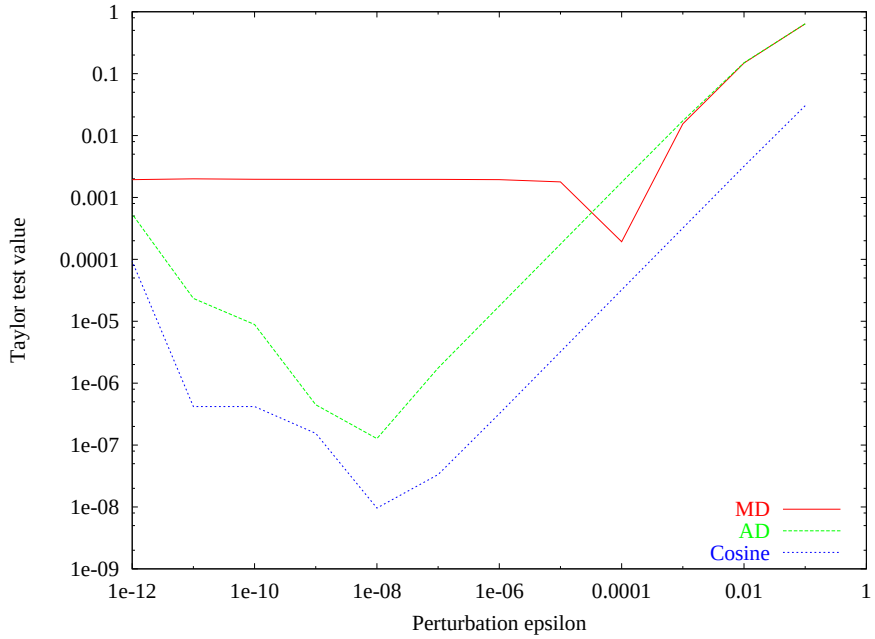

Figure 5.2: Taylor test value for decreasing perturbation magnitude ϵ.

ϵ	MD	tAD	cAD	$\cos(p)$
10^{-1}	0.63804540786915	0.63830121753202	0.63830121753264	0.03041205232236
10^{-2}	0.14846492032968	0.14988046168189	0.14988046168526	0.00319376974257
10^{-3}	0.01542864455830	0.01732063081577	0.01732063081970	0.00032087961450
10^{-4}	0.00019300516490	0.00175943719436	0.00175943719716	0.00003210296518
10^{-5}	0.00178242434520	0.00017622218867	0.00017622218867	0.00000321046302
10^{-6}	0.00194164394420	0.00001762452458	0.00001762452458	0.00000032097333
10^{-7}	0.00195757470760	0.00000175533062	0.00000175533062	0.00000003334770
10^{-8}	0.00195919598800	0.00000012703772	0.00000012703772	0.00000000959880
10^{-9}	0.00195972893490	0.00000044798390	0.00000044798390	0.00000015473100
10^{-10}	0.00196393641010	0.00000886293430	0.00000886293430	0.00000041860772
10^{-11}	0.00199619371990	0.00002339437557	0.00002339437557	0.00000041860772
10^{-12}	0.00194009405060	0.00053760231760	0.00053760231760	0.00009277546033

Table 5.2: Taylor test (absolute value) in one direction for 3 differentiation modes and for cosine function.

the case with MD, which estimates the gradient with significantly less accuracy (relative error estimated around 10^{-3}).

In order to ensure that this MD code is not plagued by some programming mistake, Odyssée was used to verify the various steps of equation differentiation that make up the MD method (see Equations 3.2 and 3.3). Indeed, the adjoint system matrix (left-hand member of Equation 3.3) can be obtained by automatic differentiation, with respect to the state variables (Z), of the subroutines of the original code that compute the right-hand member of the direct model's linearized system (Equation 2.3), i.e. the residuals $R = E_m(Z, p)$ of its non-linear equations (note that this way the system's matrix of Equation 2.3 is also automatically obtained). Comparison of the manually programmed Jacobian and of the Odyssée-derived matrix showed perfect agreement between the two independent evaluations. Similarly, Odyssée was then used to perform the other differentiation steps involved in the equation-based method of Equations 3.2 and 3.3 (see Figure 5.3): differentiation of the non-linear, direct model equations (i.e. of the already-mentioned subroutines that compute the residuals R of these equations) with respect to the parameters p, and of the expression of the objective diagnostic function F with respect to the state variables and parameters. All verifications led to the conclusion that no mathematical or programming errors have been introduced in the MD code. The gradient and Taylor test values obtained with the adjoint state method using either manually or automatically produced derivatives are identical. A reasonable conclusion from this investigation is that the slight bias revealed by the Taylor test for the equation-based method is due to the fact that the gradient calculated by this method is not strictly the gradient of the direct program output, but that of the unknown exact solution of the non-linear, direct model equations, which is only approximated by the direct program output (there subsist non-nil residuals for most iterative numerical procedures, particularly for the solution of the non-linear systems). Similar problems have been reported by other authors [RHH98]. This is one of the reasons why code-based methods have precisely been considered and developed (manually first, then as AD tools), since by construction they are free from such bias. This bias may or may not be a problem, depending on how gradients are then used. In some cases it may alter the convergence of certain iterative optimization procedures. This has not yet been thoroughly tested on the sample inverse model used for this study.

Finally, since the Taylor test amounts to the computation of the relative difference between the gradients respectively obtained with one of the differentiated codes and with the perturbation method (finite-difference approximation), the results shown above illustrate the difficulty to produce a reliable gradient estimate with the latter method, i.e. within a prescribed or known level of uncertainty. Indeed, considering the AD curve of Figure 5.2 as a plot of the relative error of the perturbation method, it can be seen that this error is highly sensitive to the parameter increment value, and that its range of variation is quite large. The choice of a proper perturbation (vicinity of the curve's minimum) is particularly cumbersome since drawing of this curve requires knowledge of the gradient, i.e. of the quantity that is looked for!. Taking too small an increment gives rise to numerical error due

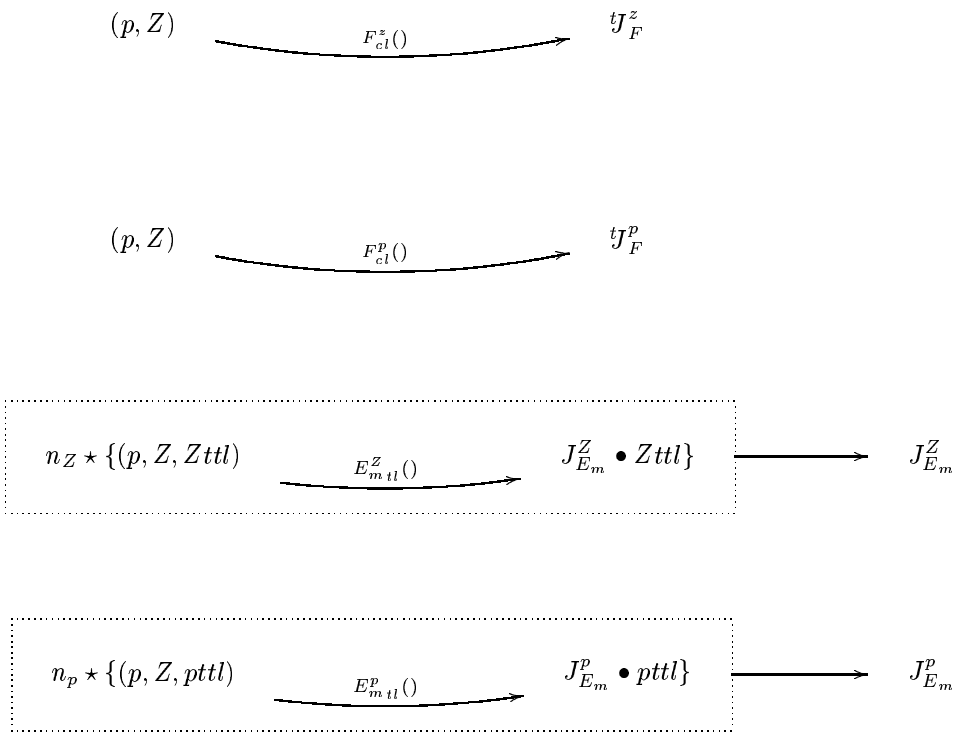

Figure 5.3: Using `Odyssée` to generate components of equation-based adjoint-state method.

	Measured	Theoretical
`MD`	1.2	–
`tAD`	$1.8m$	$4m$
`cAD`	12	5

Table 5.3: CPU-time magnitudes for 3 differentiated codes, relative to original code time.

to machine precision, while a too large value produces second-order approximation error in relation with the problem's non-linearity.

5.2 Efficiency in memory-space and CPU time

To compare the computational efficiency of the three differentiation modes, the codes were run with various numbers of input parameters, for the 10x10 problem size previously used. `MD` and `cAD` need a separate run of the original direct code before they are called, while `tAD` computes concurrently the direct model and its derivatives. To enable comparison, the computation time for each differentiation mode is defined as the total time required to obtain both the direct outputs and their derivatives; it therefore also includes the direct code execution time when a separate call of this code is needed. Wall-clock and CPU times are essentially equal for any of our runs, since no significant I/O operation is included in these codes. The first column of Table 5.3 shows the ratios of the measured computation times to the run time of the original direct code, for the three differentiation modes. Only the forward mode execution time is dependent on the number m of parameters, since `tAD` must be called once for each parameter whereas `MD` and `cAD` are only called once, irrespective of the number of parameters (Section 4.1). The second column displays the corresponding theoretical bounds [Gri89, GLVM91]

It can be seen that `MD` is by far the most computationally efficient method, the gradient computation representing only an additional 20% of the original direct code time. As mentioned earlier, the reverse mode becomes more efficient than the forward mode for a dimension of the input parameter space above a certain threshold, which in our case turns out to be 7 (inclusive). `tAD` scales with the parameter number in accordance with theory since the scaling factor is well within the theoretical bound, whereas `cAD` appears to be significantly less efficient than expected. This latter observation probably comes from the fact that the theoretical value pertains to the differentiation part proper, for hypothetical program structures, and does not include in particular the cost of restoring the trajectory, which is highly dependent on the call-graph depth, as seen before (Section 4.3). The sample code used in this study contains a rather large calling tree depth, a penalizing feature with respect to `Odyssée`'s implementation of the reverse mode. In comparison, the perturbation method takes exactly $m+1$ times the original direct code execution time for the non-centred finite-difference approximation, and $2*m$ times for the more accurate centred approximation.

It is also interesting to compare the size of memory used in the three modes. For the simple test case used here, the requirements are 1.09 Mbytes for the `MD` code, 1.97 Mbytes for `tAD`, and 79.1 Mbytes for `cAD`. This memory inflation for the reverse mode is a consequence of the huge amount of local variables that are needed for the trajectory computation. A large amount of memory may lie in oversized local variables, owing to the indetermination of actual needs outside the execution context (see Section 4.3, "syntax reversal" algorithm). It may thus be necessary for the user to make some code sections such as do loops as explicit as possible in terms of number of iterations, in order to lower those needs. The new, alternative "flow reversal" algorithm of `Odyssée` v1.7 circumvents this particular problem.

Chapter 6

Conclusions

This paper deals with the possible strategies for differentiation of complex non-linear models, whose non-linearities require iterative solution procedures. Repeated computation of vectors or matrices of derivatives is needed for the efficient solution of non-linear systems of algebraic equations as well as for the large-dimension, non-linear minimization problems that arise for inverse modeling. While only the approximate perturbation method or painful, error-prone manual differentiation (`MD`) could be considered until very recently, robust automatic differentiation (`AD`) tools are now becoming available. Based on a time-independent sample model and on the `Odyssée AD` software, we have compared the characteristics and performances of an equation-based, manually-produced gradient computation code and of two code-based, automatically-generated differentiated codes. The two latter codes correspond to the forward and reverse modes of differentiation respectively. The cross combinations: code-based + manual, and equation-based + (semi-)automatic, are also plausible approaches. However, if hand writing a code-based differentiated program could be considered for small-size source codes, it becomes impracticable for any model of significant code length. Using the `AD` tool to automatically produce all the code necessary for the gradient computation (code-based automatic approach) is the most natural way of using such a tool.

Several conclusions may be drawn from this analysis, focusing successively onto the following aspects: comparison of the forward and reverse `AD` modes, of the equation-based and of the code-based approaches, advisable strategies for model differentiation, quality of the `Odyssée AD` tool.

A forward-mode differentiated code is more readily obtained than a reverse-mode code. While some modification of the original source code should always be expected before processing by the `AD` tool, the reverse mode will likely require significantly more such prior manual adaptation work than the forward mode. Also, additional hand work is required after reverse-mode processing by `Odyssée`'s "syntax reversal" algorithm: the user must adjust some new memory allocation parameter to its problem's size. Nevertheless, it can be argued that any such needed code rewriting will always be so much less and faster work than hand coding a whole differentiated algorithm!. Computing accuracy is excellent with

the two code-based differentiating modes and essentially identical, as could be expected. Regarding computing efficiency, the forward mode becomes slower than the reverse mode as soon as the parameter number (or ratio of number of active inputs to number of active outputs) exceeds a certain threshold, and conversely. In our case, this threshold was found to be 7, the difference growing as a factor 1.8 of the parameter number and of the original code run time. However, the reverse mode entails considerably larger memory requirements, which, at present, are probably the most severe shortcoming of the method and obstacle for its operational use. New algorithms to reduce these needs are the topic of current research, and are crucial to the large-scale utilization of `AD` for full-size real-world applications. For the time being, it is therefore advisable, before choosing between the forward and reverse modes of differentiation, to carefully analyze the active input/output ratio of the piece of code being differentiated, and the expected run-times of the differentiated codes within the whole application, in typical execution conditions.

The main advantage of the equation-based method is the much greater computing efficiency that can be achieved, particularly in the case of a non-linear, iteratively-solved model. Whereas a code-based method will differentiate all the computing steps of the model, the equation-based method needs to proceed only with the final computed values of the state variables, meaning much shorter computing times. Also, the equation-based method being implemented manually, efficiency can be largely increased, relatively to the code-based automatic method, in terms of CPU time and, above all, of memory requirements, by taking into account the problem's particular structure and optimizing the code accordingly. Accuracy may be a problem as the computed derivatives are not those of the numerical variable(s) actually computed by the original code but those of the theoretical solution of a set of basic, complex equations. A bias may thus be introduced which, with our sample problem, was observed under certain conditions but remained reasonably slight. It remains to be investigated whether this possible bias is compatible with the minimization procedure for which the derivatives are being calculated. In any case, the major drawback of the equation-based approach, compared to the other differentiation methods, is the heavy hand work necessary to implement it on a real program, in our case through the adjoint-state method. Combining the human time spent in calculus, in programming, and in error-tracking, [MRSM96] estimated the savings brought by `AD` to be in a factor of 30!. Not to mention the buried mistakes that are always possible with such complex manual development, for which debugging generally is anything but a straightforward task.

Some optimal compromise between these various methods may, in the current state of `AD` development, be found in the equation-based + (semi-)automatic differentiation combination, which has also been used on our sample case (Figure 5.3). It consists in using `AD` to perform the more limited code differentiation operations needed in the equation-based adjoint state method, and doing by hand the system assembling & solving part of the algorithm. This midway approach avoids in particular applying `AD` to the computationally-intensive, system-solving parts of the original code, and confines it to those code pieces that perform the evaluation of more elementary functions. In particular, the Jacobian of the non-linear original model equations, needed to solve this direct system as well as for the adjoint state

method of inverse modeling, can be produced automatically. Since it is a square matrix, the forward `AD` mode is preferable. The part of work that is left to be done manually for both direct and inverse modeling is quite small for the time-independent, boundary-value problem used as our example: it mostly consists in the evaluation of residuals for the direct model's basic non-linear equations and of the diagnostic function value. In the case of a time-dependent problem, there might be the need for additional hand programming to perform the saving/restoration (or recomputation) of the required part of the trajectory over time.

Finally, it can be concluded that `Odyssée` is an efficient and reliable `AD` tool, which is very user-friendly and quite easily applied by a non-specialist user. Its on-going development is keeping it up with the state-of-the-art of this active research field. Although a number of problems do remain to be solved to maximize computing efficiency and minimize manual work, it definitely is now a tool of major interest for many modelers.

Acknowledgements:
The financial support brought to David Elizondo by the Ministère des Affaires étrangères (France), by the Pôle Universitaire Européen de Montpellier et du Languedoc-Roussillon (France) and by the Secretaría de Estado de Universidades, Investigación y Desarrollo of Ministerio de Educación y Cultura (Spain), is gratefully acknowledged.

Bibliography

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide, 2nd edn.* SIAM, Philadelphia, PA., 1995.

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

[BCK+] 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.

[CG97] I. Charpentier and M. Ghemires. Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée. Application au code météorologique Meso-NH. Rapport de recherche 3251, INRIA, September 1997.

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

[DS96] J.E. Dennis and R.B. Schnabel. *Numerical methods for unconstrained optimization and non-linear equations (Classics in Applied mathematics).* SIAM, Philadelphia, PA., 1996.

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

[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.

[FN99] C. Faure and U. Naumann, editors. *AD'2000 abstracts: From Simulation to Optimization*, Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France), 1999. INRIA.

[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.

[Gie97] R. Giering. *Tangent linear and Adjoint Model Compiler , Users manual*, 1997. Unpublished, available from `http://puddle.mit.edu/~ralf/tamc`.

[GL95] J.C. Gilbert and C. Lemaréchal. The modules m1qn3 and n1qn3. version 2.0c. Technical report, INRIA, 1995.

[GLVM91] J.C. Gilbert, G. Le Vey, and J. Masse. La différentiation automatique des fonctions représentées par des programmes. Rapport de recherche 1557, INRIA, 1991.

[Gri89] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical programming: recent developments and applications*, pages 83–108, Dordrecht, 1989. Kluwer Academic Publishers.

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

[LT86] F.X. Ledimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects. *Tellus*, 38A:97–110, 1986.

[MAS96] G.I. Marchuk, V.I. Agoshkov, and V.P. Shutyaev. *Adjoint equations and perturbation algorithms in nonlinear problems*. CRC Pr., 1996.

[MRSM96] J.-M. Malé, N. Rostaing-Schmidt, and N. Marco. Automatic differentiation: an application to optimum shape design in aeronautics. In J.-A. Désidéri, C. Hirsch, P. Le Tallec, E. Oñate, M. Pandolfi, J. Périaux, and E. Stein, editors, *Minisymposia of ECCOMAS 96*. John Wiley & Sons, Ltd, September 1996.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipies in Fortran. 2nd edn.* Cambridege University Press, Cambridge, 1992.

[RDG93] N. Rostaing, S. Dalmas, and A. Galligo. Automatic Differentiation in Odyssée. *Tellus*, 45A(5):558–568, 1993.

[RHH98] M.L.J. Rightley, R.J. Henninger, and K.M. Hanson. Adjoint differentiation of hydrodynamic codes. Technical report, Los Alamos National Laboratory, 1998.

Appendix A

Illustration of AD using Odyssée

Differentiation is performed using `Odyssée` v1.6 with respect to 2 input variables `in` and `inout` of the subroutine `head`.

A.1 Original source code

```
subroutine head(n,in,inout,out)
parameter(m=100)
real in(n),inout(n),out(n), local(m)

do i=1,n
   local(i)=(in(i)-inout(i))**2
enddo
if(n.gt.10) call sub(n,local,inout,out)
do i=1,n
   local(i)=local(i)**2
   inout(i)=sin(local(i))
   out(i)=out(i)+log(local(i))**1.5
enddo
end

subroutine sub(n,in,inout,out)
real in(n),inout(n),out(n)
do i=1,n
   out(i)=in(i)**3
   inout(i)=2.*inout(i)+out(i)**4
enddo
end
```

A.2 Odyssée's differentiated code, forward mode

```
COD Odyssee  1.6 (strategy: Standard)
COD Unit: headtl  (derived from unit: head)
COD Active INPUTs:      dummys= inout in
COD Active OUTPUTs:     dummys= out inout
COD Dependencies between INPUTs and OUTPUTs:
COD inout <--  inout in
COD out <--  inout in

        SUBROUTINE HEADTL (N, IN, INOUT, OUT, INTTL, INOUTTTL, OUTTTL)
        PARAMETER (M = 100)
        REAL IN(N), INOUT(N), OUT(N), LOCAL(M)
        REAL INTTL(N), INOUTTTL(N), OUTTTL(N), LOCALTTL(M)
        REAL SR01S, SR01STTL
        DO I = 1, N
                LOCALTTL(I) = 2*(INTTL(I)-INOUTTTL(I))*(IN(I)-INOUT(I))
                LOCAL(I) = (IN(I)-INOUT(I))**2
        END DO
        DO N1 = 1, N
                OUTTTL(N1) = 0.
        END DO
        IF (N.GT.10) CALL SUBTL(N, LOCAL, INOUT, OUT, LOCALTTL, INOUTTTL, OUTTTL)
        DO I = 1, N
                LOCALTTL(I) = 2*LOCALTTL(I)*LOCAL(I)
                LOCAL(I) = LOCAL(I)**2
                INOUTTTL(I) = LOCALTTL(I)*COS(LOCAL(I))
                INOUT(I) = SIN(LOCAL(I))
                SR01STTL = LOCALTTL(I)/LOCAL(I)
                SR01S = LOG(LOCAL(I))
                OUTTTL(I) = OUTTTL(I)+1.5*SR01STTL*SR01S**(1.5-1.)
                OUT(I) = OUT(I)+SR01S**1.5
        END DO
        END
```

```
COD Unit: subtl  (derived from unit: sub)
COD Active INPUTs:      dummys= inout in
COD Active OUTPUTs:     dummys= out inout
COD Dependencies between INPUTs and OUTPUTs:
COD inout <--  inout in
COD out <--  in
        SUBROUTINE SUBTL (N, IN, INOUT, OUT, INTTL, INOUTTTL, OUTTTL)
        REAL IN(N), INOUT(N), OUT(N)
        REAL INTTL(N), INOUTTTL(N), OUTTTL(N)
        DO I = 1, N
                OUTTTL(I) = 3*INTTL(I)*IN(I)**2
                OUT(I) = IN(I)**3
                INOUTTTL(I) = 2.*INOUTTTL(I)+4*OUTTTL(I)*OUT(I)**3
                INOUT(I) = 2.*INOUT(I)+OUT(I)**4
        END DO
        END
```

A.3 Odyssée's differentiated code, reverse mode

```
COD Odyssee  1.6 (strategy: Standard)
COD Unit: headcl  (derived from unit: head)
COD Active INPUTs:      dummys= inout in
COD Active OUTPUTs:     dummys= out inout
COD Dependencies between INPUTs and OUTPUTs:
COD inout <--  inout in
COD out <--  inout in
        SUBROUTINE HEADCL (N, IN, INOUT, OUT, INCCL, INOUTCCL, OUTCCL)
        PARAMETER (M = 100)
        REAL IN(N), INOUT(N), OUT(N), LOCAL(M)
        INTEGER ODYN
        PARAMETER (ODYN=M)
        REAL INCCL(N), INOUTCCL(N), OUTCCL(N), LOCALCCL(M)
        REAL SR01S, SR01SCCL, SAVE5(ODYN), SAVE6(ODYN)
        REAL SAVE7(ODYN), SAVE8(ODYN), SAVE9(ODYN), SAVE10(ODYN)
        LOGICAL TEST4
C
C Initializations of local variables
        DO N1 = 1, M
                LOCALCCL(N1) = 0.
        END DO
        SR01SCCL = 0.
C
C Trajectory
        DO I = 1, N
                LOCAL(I) = (IN(I)-INOUT(I))**2
```

```
        END DO
        TEST4 = N.GT.10
        IF (TEST4) THEN
                DO N1 = 1, N
                        SAVE5(N1) = INOUT(N1)
                        SAVE6(N1) = OUT(N1)
                END DO
                CALL SUB(N, LOCAL, INOUT, OUT)
        END IF
        DO I = 1, N
                SAVE7(I) = LOCAL(I)
                LOCAL(I) = LOCAL(I)**2
                SAVE8(I) = INOUT(I)
                INOUT(I) = SIN(LOCAL(I))
                SAVE9(I) = SR01S
                SR01S = LOG(LOCAL(I))
                SAVE10(I) = OUT(I)
                OUT(I) = OUT(I)+SR01S**1.5
        END DO
C
C Transposed linear forms
        DO I = N, 1, -1
                OUT(I) = SAVE10(I)
                SR01SCCL = SR01SCCL+OUTCCL(I)*(1.5*SR01S**(1.5-1.))
                SR01S = SAVE9(I)
                LOCALCCL(I) = LOCALCCL(I)+SR01SCCL*(1./LOCAL(I))
                SR01SCCL = 0.
                INOUT(I) = SAVE8(I)
                LOCALCCL(I) = LOCALCCL(I)+INOUTCCL(I)*COS(LOCAL(I))
                INOUTCCL(I) = 0.
                LOCAL(I) = SAVE7(I)
                LOCALCCL(I) = LOCALCCL(I)*(2*LOCAL(I))
        END DO
        IF (TEST4) THEN
                DO N1 = N, 1, -1
                        OUT(N1) = SAVE6(N1)
                        INOUT(N1) = SAVE5(N1)
                END DO
                CALL SUBCL(N, LOCAL, INOUT, OUT, LOCALCCL, INOUTCCL, OUTCCL)
        END IF
        DO N1 = N, 1, -1
                OUTCCL(N1) = 0.
        END DO
        DO I = N, 1, -1
                INCCL(I) = INCCL(I)+LOCALCCL(I)*(2*(IN(I)-INOUT(I)))
                INOUTCCL(I) = INOUTCCL(I)-LOCALCCL(I)*(2*(IN(I)-INOUT(I)))
```

```
                LOCALCCL(I) = 0.
        END DO
        END

COD Unit: subcl  (derived from unit: sub)
COD Active INPUTs:      dummys= inout in
COD Active OUTPUTs:     dummys= out inout
COD Dependencies between INPUTs and OUTPUTs:
COD inout <--  inout in
COD out <--  in
        SUBROUTINE SUBCL (N, IN, INOUT, OUT, INCCL, INOUTCCL, OUTCCL)
        REAL IN(N), INOUT(N), OUT(N)
        PARAMETER (M = 100)
        INTEGER ODYN
        PARAMETER (ODYN=M)
        REAL INCCL(N), INOUTCCL(N), OUTCCL(N)
        REAL SAVE2(ODYN), SAVE3(ODYN)
C
C Trajectory
        DO I = 1, N
                SAVE2(I) = OUT(I)
                OUT(I) = IN(I)**3
                SAVE3(I) = INOUT(I)
                INOUT(I) = 2.*INOUT(I)+OUT(I)**4
        END DO
C
C Transposed linear forms
        DO I = N, 1, -1
                INOUT(I) = SAVE3(I)
                OUTCCL(I) = OUTCCL(I)+INOUTCCL(I)*(4*OUT(I)**3)
                INOUTCCL(I) = 2*INOUTCCL(I)
                OUT(I) = SAVE2(I)
                INCCL(I) = INCCL(I)+OUTCCL(I)*(3*IN(I)**2)
                OUTCCL(I) = 0.
        END DO
        END
```

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