Tellus (1993), 45A, 558–568
Printed in Belgium – all rights reserved

TELLUS
ISSN 0280–6495

Automatic differentiation in Odyssée

By NICOLE ROSTAING[1,*], STÉPHANE DALMAS[2] and ANDRÉ GALLIGO[1], [1]*Projet SAFIR, Université de Nice Sophia-Antipolis, Laboratoire de mathématiques URA 168 du CNRS, B.P. 71, 06108 Nice Cedex 2, France*; [2]*Projet SAFIR, INRIA Sophia Antipolis, B.P. 93, 06908 Sophia Antipolis Cedex, France*

(Manuscript received 4 November 1992; in final form 13 May 1993)

ABSTRACT

This paper describes the design of *Odyssée*, a system for FORTRAN programs manipulations and its application to automatic differentiation. The *Odyssée* system manipulates FORTRAN programs as symbolic objects. It is an open system built as a toolkit, written in a high-level programming language adapted to this purpose. The use of a variational method to perform data assimilation requires the computation of the gradient of a cost function represented by a large-size FORTRAN program. The usual drawback in the reverse automatic differentiation method is the storage requirement. The *Odyssée* system allows one to implement storage/recomputation strategies in order to fit the needed compromizes. We present the implementation of the strategy used in the weather forecasting ARPEGE/IFS project to produce the adjoint code from the code representing the numerical model. *Odyssée* produces the same code as the hand-written adjoint code for the ARPEGE/IFS project.

1. Introduction

We are interested in applying computer algebra to numerical problems. The ARPEGE/IFS project (a joint work between Météo-France and the European Centre for Medium-Range Weather Forecasts) uses a variational method to perform data assimilation (Courtier and Talagrand, 1987a) (Courtier and Talagrand, 1987b). This requires the computation of the gradient of a cost function. For this matter, O. Talagrand and Ph. Courtier have developed an efficient automatic differentiation method. This method has been applied by hand for now. The automation of the ARPEGE/IFS method was a good approach to test our ideas on the collaboration between numeric and symbolic computing. This automatic differentiation technique is worth studying because it is up to now the only one working on such large size FORTRAN programs. This is due to a strategy of storage and recomputation adapted to their problem.

We have developed a system, *Odyssée*, to manipulate FORTRAN programs as symbolic objects. *Odyssée* is closer to symbolic manipulation systems (like *Maple* or *Mathematica*) than common automatic differentiation packages. The user can request built-in actions (such as producing the derivatives of a given subroutine, verifying some programming norms ...) as well as programming its own transformations or analysis. This can be done in CAML, a polymorphic functional language, the implementation language of *Odyssée*. *Odyssée* is organized as a toolkit of components providing basic services (parsing FORTRAN programs, performing various kind of iterations ...). We also have implemented ready to use automatic differentiation programs. A graphical user interface provides a convenient access to the system.

This study has produced a prototype to test our approach: we can automatically obtain the same adjoint codes as those handwritten by the ARPEGE/IFS developers. It is not yet a general automatic differentiation tool. Until now, to adapt *Odyssée* to a particular code, one needs to program in CAML, but we hope that after several

* Corresponding author.

experiences we will take into account most of the usual needs of meteorological or oceanographic models and incorporate them in *Odyssée*. A work to adapt *Odyssée* to ARPS (Advanced Regional Prediction System developped at the University of Oklahoma) is in progress. However, if we apply the current *Odyssée* system on a particular code computing a function, the output will be the gradient of the initial function, but its integration in the global code may require some work from the user, this work can be avoided by adapting *Odyssée* to the application.

In the Section 2, we will give our point of view on automatic differentiation, then we will briefly present the *Odyssée* system. The last part of the paper describes the automation of the ARPEGE/IFS method and its implementation in *Odyssée*.

This work is done within the SAFIR group (I3S et Laboratoire de Mathématiques de l'UNSA et du CNRS, INRIA Sophia-Antipolis) and is partially supported by DRET and also by the GDR "Méthodes variationnelles en Météorologie et Océanographie" of CNRS.

2. Automatic differentiation

Differentiation is a very usual operation in numerical computing, especially in optimization problems. But purely numerical approaches such as finite differences may, in some cases, lead to inacceptable rounding errors or prohibitive computing times when the number of independant variables (input variables) is large. Hence, several techniques have been used (since the seventies) to overcome these difficulties. They consist in mechanically constructing another program (subroutine) from the subroutine computing the given function. This produced subroutine computes the jacobian matrix, or the jacobian applied to a vector or even higher order derivatives. This is globally known as *automatic differentiation.*

The basic principle of these methods is fairly simple: it is based on the observation that a sequence of statements can be regarded as the composition of the functions representing the computations performed by each statement, therefore the chain rule can be applied.

There exist two classes of automatic differentiation algorithms: the more intuitive one is called the *direct mode* and performs the transformations statement by statement in the order of their occurences in the initial program. A more sophisticated class of methods is the *reverse mode.* The key idea is to introduce and to generate "adjoint variables" which evaluate the derivatives of one output variable (or a combination of them) with respect to the intermediate variables. A result of complexity (Baur and Strassen, 1983) (Morgenstern, 1985) shows that using the reverse mode, it is possible to compute a scalar function and all its first-order derivatives in a time proportional to the time needed to evaluate the function (i.e., it does not depend on the number of variables). But the storage requirement can be very expensive.

The *direct mode* of differentiation is an application of the chain rule formula

$$T_{x_0}(f \circ g \circ h) = T_{g \circ h(x_0)} f \circ T_{h(x_0)} g \circ T_{x_0} h,$$

where the notation $T_x f$ represents the linear tangent mapping of f at point x. It becomes by transposition:

$${}^t T_{x_0}(f \circ g \circ h) = {}^t T_{x_0} h \circ {}^t T_{h(x_0)} g \circ {}^t T_{g \circ h(x_0)} f.$$

The *reverse mode* computes derivatives following this second rule from right to left. We can see that the value $g \circ h(x_0)$ need to be known by the program computing the linear applications in the reverse mode. The values x_0, $g(x_0)$ and $g \circ h(x_0)$ may be stored or recomputed. In the automatic differentiation algorithms the key point is then the trade-off between storage and arithmetic operations. In fact, the reverse mode is efficient provided that the memory requirement does not become too large.

Of course, applying these methods by hand is very tedious and error-prone. What a scientist or an engineer would like is a tool that can automatically produce the required programs from the input program. The problems to achieve this goal are two-fold.

- Algorithmic problems: to design an algorithm taking into account control structures (iterative and conditional) and able to work on a sufficiently large class of programs (the user should have as little constraints as possible to write its code)
- Technological problems: this tool should manipulate a program as a structured object,

performing symbolic manipulations. It requires to implement some sophisticated tools relevant to compilation techniques.

Another important pratical concern is that the result of the process should be easy to incorporate into the final application. In a very pratical setting, we can distinguish two classes of implementation methods according to their technology.

- Implementations based on preprocessing or overloading techniques avoiding a full analysis of the program. The drawbacks are that they require a large space of storage and that they constraint the form of input programs. Among the existing softwares we can mention PADRE2 (Kubota, 1991), ADOL-C (Griewank et al., 1990).
- Implementations that use compilation methods, working with a structured representation of the program. In this class, we can cite ADIFOR (Bischof et al., 1992) (which is mainly based on the direct mode).

Odyssée belongs to this second class of implementations. Our aim is to be able to perform automatic differentiation in reverse mode as well as in the direct mode.

3. Odyssée: a toolkit of software components

3.1. General presentation

Odyssée takes as input FORTRAN programs. Each compilation unit (subroutine, function or main program) is analysed and structured into an abstract syntax tree and a symbol table containing information about the occuring identifiers. All the manipulations are performed on this tree structure. Odyssée returns the FORTRAN code obtained from the transformed tree or required information in the case of code analysis.

The tree structure provides an homogeneous representation of the different elements constituting a program or a subroutine: the statements, the expressions and the specifications. In the case of expressions the nodes of the tree are the algebraic operators or the names of basic functions (such as the trigonometric functions), the subtrees are the expressions representing their arguments. At each kind of statement corresponds a node (Do for a loop, If for a conditional statement, Let for an assignment ...) whose subtrees are expressions or sequences of statements. The specifications have an analogous representation.

When designing *Odyssée*, we had in mind two targets: providing an open system where it would be easy to implement different algorithms (automatic differentiation methods as well as other transformations) and being able to deal with "real life" FORTRAN code and not just with academic examples. This lead us to choose an adapted architecture and a high level programming language (Rostaing and Dalmas, 1992).

3.2. Architecture and programming language

Odyssée is composed of basic tools (the lexical analyser, the parser and the FORTRAN code generator), of general tools which can be specialized (the components for tree traversals and transformations) and of specific tools devoted to a particular application. *Odyssée* is a programmable system: it provides a toolkit of reusable components for the programmer who wishes to implement FORTRAN code transformations or analysis.

This has been possible thanks to the programming language CAML (Weis et al., 1990) (developped at INRIA and Ecole Normale Supérieure). It is a functional language (functions can be arguments and results), strongly typed (each CAML expression has its type inferred automatically by the compiler) and polymorphic (the type inferred is the most general possible). It is then possible to write general components: the tree traversal CAML function may be parametrized by the transformation to apply, this transformation being another CAML function. For example, the function that transforms a statement into a new statement in the tangent linear code is parametrized by the storage/recomputation strategy, which works on a sequence of statements. The strong typing insures a certain safety of the performed transformations.

The memory allocation is dynamic in CAML, it is very important for easily manipulating FORTRAN programs of any size. Other CAML features are adapted to our purpose like the concrete types to describe the FORTRAN syntax, the pattern matching mechanism to symbolically representing and applying the rules of transformation.

In *Odyssée*, the FORTRAN program is strucured into an abstract syntax tree with an associated symbol table. To get this structure, the steps

are similar to those of a compiler: the lexical analyser (we have used an existing one written in C) recognizes FORTRAN keywords and produces a sequence of tokens. Then a parsing is performed. The parser has been built thanks to the CAML interface with the YACC parser generator. This interface gives the possibility of conveniently associating an abstract syntax tree to a program. Some information about the identifiers cannot be deduced from the syntax only. We find this information during parsing and record it in a symbol table (Rostaing and Dalmas, 1992).

3.3. *Applications in code analysis*

For the moment, *Odyssée* is essentially devoted to automatic differentiation. However, we also have developed a few tools that can help the user to find some errors prior to running the code. As an example of the flexibility of *Odyssée*, we would like to mention that the CAML program verifying the ARPEGE/IFS norms is about 400 lines long and the one for control of coherence is about 950 lines.

3.3.1. Verification of programming norms for the ARPEGE/IFS project. As the ARPEGE/IFS code is large and developed in several sites, rules on the identifier names have been set by the ARPEGE/IFS code designers. A name must begin with a different letter according to its FORTRAN type (integer, real, double precision ...), its status in the code (local variable, parameter, dummy variable ...) and sometimes its use (loop variable). We have written a CAML tool in Odyssée which automatically enforces these norms.

Verifying the norms depending on the type and the status is easy, all the information is already recorded in the symbol table. A more interesting problem is how to check the use of loop variables. The difficulty is to characterize the authorized occurrences of loop variables. A loop variable in the ARPEGE/IFS code is not only an identifier appearing behind the **Do** keyword but also an identifier used as a constant index of array. In this last case, it is often assigned to an integer expression before the loop (meaning that the iteration is performed on a row or a column for example). These uses have been formalized with some heuristic choices (as how far from a loop it is permitted to use its loop variable) and implemented in *Odyssée*. It requires an accurate analysis of the abstract syntax tree. The ARPEGE/IFS developers now check their code with this tool.

It would be interesting to allow other users to give their own norms. There are two possible levels of adaptation: changing some elementary rules, the existing functions can be easily parametrized to accomodate such facilities, or adding sophisticated criteria (as checking loop variables) which requires some CAML programming.

3.3.2. Control of coherence. We have written a program which checks that the number of parameters and the types of these parameters in function and subroutine calls correspond to the number and types of dummy variables in the corresponding definitions. It also verifies COMMON contents. It has been successfully tested on the whole MODULEF library (400,000 lines of FORTRAN code) (Bernardou et al., 1986).

This tool takes as input a list of FORTRAN files. If the definition is not available when a call to a subroutine is encountered, its validity control remains unresolved until its definition is reached (so that there is no need to enter the subroutines in the order of their occurrences). There exists also a useful facility for dumping a commonly used library that avoids a complete analysis. For the moment the source code of the libraries must be available for our tool. This problem may be avoided by providing just a template of the involved subroutines.

4. Automation of the method and its implementation

4.1. *Automatic differentiation as program transformation*

Analysis of automatic differentiation methods shows that we can distinguish three kinds of transformation rules.

- algebraic differentiation rules on the expressions
- rules depending on the mathematical meaning given to each statement
- non local rules describing the organization of the resulting program

Derivation of expressions is a classical process: we apply the symbolic mathematical formulas. We then rather focus on the transformation of statements. Dealing with control structures (conditional and iterative) raises some semantical

Tellus A/45/5-15

difficulties. A safe and not too restrictive approach in practice is to require that the control structures of the original program are preserved in the derived program. The same branches will be taken and the same number of iterations will be performed in loops. These requirements induce the following rules of transformation.

- **Do**: We differentiate the body of the loop and preserve the **Do** structure (same number of iterations, same initial and final values).
- **If**: it represents the definition of a piecewise function. We thus keep the test and differentiate the two branches.
- **Assignment**: it gives a name to the result of a computation. If this computation involves variables it is differentiated, if not it is kept as an intermediary computation.
- **Call**: each subprogram call can be seen as a "factorized" sequence of statements, as it may be used several time with different values. Thus when differentiating we replace this call by a call to the derivative of the subprogram obtained by suitable means (which can be specified by the user).

The last set of rules concerns the organization of calculations. As the differentiation process is based on the chain rule, we need in the derived program some values already computed. These values can be stored or recomputed when necessary: the trade-off between the storage and the recomputations defines the strategy.

According to the philosophy of *Odyssée*, this lead us to develop a set of tools for implementing automatic differentiation methods in our system: differentiation and simplification of expressions (computer algebra components) and differentiation of statements (statement transformation following the rules we gave above).

The differentiation components are parametrized with respect to functions that abstract the differences between methods and the choices of a particular application. For instance, they describe how to name the generated identifiers or give the recomputation strategy.

4.2. *The ARPEGE/IFS method*

In the ARPEGE/IFS project, a weather forecasting project, variational methods are used to perform data assimilation. The evolution of the atmospheric state is governed by a system of partial differential equations. This system is discretized on a global grid for the whole sphere and can be formally given by $X_{i+1} = G(X_i)$. The state vector X_i contains the atmospheric state (pressure, temperature, wind components) at time t_i at each point of the grid. It is then of a large size (10^6 components). We need an initial condition for this system. The observations are not sufficient to provide a suitable initial condition. Hence it is estimated using the model and the observations. The technique consists in finding the initial state X_0 which minimizes the distance function

$$J(X_0) = \tfrac{1}{2} \sum_i \langle X_i - X_i^{\text{ref}}, X_i - X_i^{\text{ref}} \rangle,$$

where X_i is the model state at time t_i and X_i^{ref} is the reference state at the same time. The minimization algorithm used requires to compute the gradient of this cost function.

As there are too many input variables, finite differences cannot be used to compute the gradient. The ARPEGE/IFS team then developed a particular automatic differentiation method. The gradient is computed using the reverse mode. The size of the state vector explains that it has been necessary to spend some computing time to recover space memory.

An implementation of the direct mode exists inside ARPEGE/IFS, it is the tangent linear code. But it is mainly used as an intermediary step in the elaboration of the adjoint code.

4.2.1. Tangent linear code. It computes the tangent linear approximation (a directional derivative, (Morgenstern, 1973)) of the function given by the original program. It is an implementation of the direct mode except that it separates the computation of the function from the computation of the tangent linear approximation in two FORTRAN units.

This separation implies that the values involved in non linear computations in the original code will be needed in the differentiated code. The ARPEGE/IFS code choices are: the output values of a subroutine are stored when computing the initial function and given as parameters to the tangent linear subroutine, the local variables are recomputed in the tangent linear code (by copying the corresponding statement).

Moreover, they use a trick: instead of renaming the derivative variables, they give a new name to

the old one (called trajectory variables). This avoids to produce the tangent linear code for a subroutine computing a linear function (since only non linear computations in the original code will be recomputed and then renamed in the tangent linear code (Talagrand, 1991)).

The parametrization of the general automatic differentiation tools in *Odyssée* is representative of the direct mode. It is easy to adapt this tool to applications just by writing the specialized functions. Consequently, the implementation of the tangent linear method in *Odyssée* has essentially consisted in programming the identifier renaming function and the storage/recomputation function. This last component is decomposed in two more general and systematic functionalities. An assignement to a variable local identifier becomes two assignments in a first pass: the same one (but renamed according to the ARPEGE/IFS design) and its derivative. Then a second tool eliminates the unused assignments. It is done thanks to a quite simple dependency analysis between the statements.

4.2.2. Adjoint code. The adjoint code may be seen as a formal transposition of the tangent linear code (as it computes the transpose of the jacobian applied to a vector given as input) (Talagrand, 1991). The tangent linear approximation computes a set of linear forms, then the adjoint code can be obtained by transposing each linear form. A typical statement like:

$$\mathrm{d}z = \partial_1 f \, \mathrm{d}x + \partial_2 f \, \mathrm{d}y$$

becomes the sequence of statements

$$\mathrm{d}x = \mathrm{d}x + \partial_1 f \, \mathrm{d}z,$$

$$\mathrm{d}y = \mathrm{d}y + \partial_2 f \, \mathrm{d}z.$$

This exchanges the input variables $(\mathrm{d}x, \mathrm{d}y)$ with the output variable $\mathrm{d}z$.

Let us now consider the following template of program:

$$z = f_1(x, y),$$

$$r = z + f_2(x, y).$$

The input variables are x and y, the output variable is r and z is an intermediate computation. The corresponding tangent linear code will be:

$$\mathrm{d}z = \partial_1 f_1 \, \mathrm{d}x + \partial_2 f_1 \, \mathrm{d}y, \quad (1)$$

$$\mathrm{d}r = \mathrm{d}z + \partial_1 f_2 \, \mathrm{d}x + \partial_2 f_2 \, \mathrm{d}y. \quad (2)$$

And the corresponding adjoint code will be:

$$\mathrm{d}z = 0 \qquad \mathrm{d}x = 0 \qquad \mathrm{d}y = 0$$

$$(2) \begin{cases} \mathrm{d}z = \mathrm{d}z + \mathrm{d}r \\ \mathrm{d}x = \mathrm{d}x + \partial_1 f_2 \, \mathrm{d}r \\ \mathrm{d}y = \mathrm{d}y + \partial_2 f_2 \, \mathrm{d}r \end{cases}$$

$$(1) \begin{cases} \mathrm{d}x = \mathrm{d}x + \partial_1 f_1 \, \mathrm{d}z \\ \mathrm{d}y = \mathrm{d}y + \partial_2 f_1 \, \mathrm{d}z \end{cases}$$

The input variable is $\mathrm{d}r$ and the output variables are $\mathrm{d}x$ and $\mathrm{d}y$.

In this example, we can point out two important rules to produce the adjoint code. The input and intermediary variables of the tangent linear code ($\mathrm{d}x$, $\mathrm{d}y$ and $\mathrm{d}z$) have to be initialized in the adjoint code. And as the linear forms are not independent (due to the intermediate variable z), their transpose must be computed in the reverse order in the adjoint code. This necessary inversion implies difficulties when bringing this method into operations. In general, the computation of the partial derivatives $\partial_i f_j$ involves constant intermediary values that will be recomputed in the adjoint code. Some come from the non linear computations in the direct code (the numerical model), some are due to loop variables initializations, ... The difficulty is to put all the constant intermediary computations where they are needed by the linear forms in the adjoint code. This is not trivial because they may depend on each other and a same identifier may be reassigned during the program. To summarize, the difficulty is that, in the adjoint code, the transpose of the linear forms must be computed in the reverse order with respect to their corresponding statement in the tangent linear code, but the intermediary computations (not involving derivatives) must be computed in the same order as in the tangent linear code, and must affect the linear form they affected in the previous code.

Loops are the biggest problem in this algorithm. A loop from 1 to n is a repeated sequence of statements (the body of the loop). According to the previous rule, not only the body of the loop must be transposed starting from the last state-

ment, but the resulting loop should be executed from n to 1. The body of the loop can contain intermediate constant computations of arrays that must not be inverted (a statement like $x_{i+1} = f(x_i)$). The first idea is to do this computation in another loop performing the computations in the original way. Of course, it can be impossible due to the dependencies between these computations and the linear forms remaining in the loop. The only way to overcome this difficulty is to store parts of the intervening arrays. One important characteristic of the ARPEGE/IFS code structure is that such storage is not necessary.

This adjoint code algorithm is basically simple (just a tangent linear code transposition). It is remarkable that on a straight-line program (without control structures), it produces the same code as the usual reverse mode algorithm (Morgenstern, 1985). The existing implementations of the reverse mode use systematic storages that can be here avoided thanks to a careful analysis of the code joined with a particular organization of the computations in the numerical model, enforced by the separation between the code computing the function and the one computing the derivatives. Then the strategy we have implemented is lead by this code organization. Thanks to our tool, we have been able to reproduce this organization in the tangent linear code and the adjoint code: the values assigned to local identifiers are recomputed instead of being systematically saved, the values given as parameters are given as parameters, the global identifiers are not modified then do not need to be saved.

Practically, a storage/recomputations strategy is composed of two parts: the general code organization (how the computations are separated in different subroutines) and the computations done within a subroutine. To modifying the general organization of the code, one needs considering all the model code instead of treating subroutines one by one. Inside a subroutine, we can implement sophisticated strategies on the intermediary local computations (deciding locally what is recomputed or stored). If some of them involves large arrays, it may be useful.

4.3. *Implementation*

This section is devoted to the tools for automatic differentiation we have developed and the techniques we have used to implement these methods.

Odyssée takes as input a subroutine and the list of parameters with respect to which the subroutine will be differentiated. From this set of variables, the system can automatically infer the status (variable or constant) of local scalar identifiers. It depends if they are assigned to expressions containing at least one variable. This status may change dynamically in the course of the program (a local identifier can be reassigned). To avoid to recompute this information each time it is needed, it is done once for all during a preprocessing stage and attached to the relevant statements. This preprocessing stage also takes care of the `min`, `max`, `abs` and `sign` INTRINSIC functions. They are transformed into `if` statement to be differentiated. A last pass over the whole generated code inserts the correct declarations for the generated identifiers.

Our automatic differentiation tools accept as input a code containing subroutine calls without requiring that the source code of the subroutine is available. We collect the relevant information in a data base. For each subroutine the way to differentiate it and the list of its variable parameters are recorded. This data base can be saved in a file and easily updated through a convenient graphical user interface. The information to differentiate function statements is automatically inferred by the system during the preprocessing stage.

The adjoint method implementation lead us to enrich the computer algebra part of *Odyssée* (linear forms manipulations) as well as the program analysis one (criteria of dependency).

For the moment, *Odyssée* accepts programs written in standard FORTRAN77. Some CRAY extensions are implemented. The basic components could be adapted to other usual extensions. We have not yet considered the case of FORTRAN90.

4.4. *Current functionalities of* ***Odyssée***

Odyssée takes as input a FORTRAN77 file containing one or several subroutines. Using a graphical interface, the user can apply code transformation or request analysis.

1. The implemented operations are:
   - automatic differentiation in the direct mode: it produces the tangent linear code.

At each subroutine from the original file will correspond a new subroutine
- automatic differentiation in the reverse mode: it produces the adjoint code. At each subroutine from the original file will correspond a new subroutine
- programming norms verification (until now ARPEGE/IFS norms only)

2. Information can be provided to *Odyssée* through this interface:

- essentially filling the basis of knowledge necessary for differentiating a subroutine: the list of variables among the dummy parameters, the way to differentiate a subroutine called in the current subroutine and to built the list of dummy parameters for the obtained subroutine (a representation is given for both, the user does not need to program it in CAML)
- consulting this basis

From a practical point of view, the time spent for differentiating a subroutine is reasonnable. Something like 10 seconds (CPU time) for a 500 lines subroutine in the direct mode, and 13 seconds in the reverse mode (on a workstation). The complexity is roughly linear for the direct mode and quadratic for the reverse mode (relatively to the number of statement lines).

Odyssée works on any computer supporting the CAML language (currently we use SUN or DEC workstations). The graphical interface requires x11.

4.5. Example of generated programs

Let us consider the following subroutine. It comes from the ARPEGE/IFS code. To provide a more readable example, we just have kept a part of the real subroutine. Of course the meanings of the computations are not the point here. We are interested in the symbolic automatic treatment.

```
      SUBROUTINE gpgrp (kproma, kprof, kflev, pvc, pvbh, prt,
     &prtl, pprl, porogl, psgrtl)

      REAL pvc(kflev), pvbh(0:kflev)
      REAL prt(kproma,kflev), prtl(kproma,kflev)
      REAL pprl(kproma), porogl(kproma), psgrtl(kproma,kflev)
      REAL zgphl(kproma,kflev)

      DO 102 jrof=1, kprof
         zgphl(jrof,kflev)=porogl(jrof)
  102 CONTINUE

      DO 104 jlev=kflev, 2, -1
         DO 106 jrof=1, kprof
            zgt=prt(jrof,jlev)*pvc(jlev)
            zgphl(jrof,jlev-1)=zgphl(jrof,jlev)-
     &        ((prtl(jrof,jlev)*zgt)*pprl(jrof))
  106    CONTINUE
  104 CONTINUE

      DO 202 jlev=1, kflev
         DO 204 jrof=1, kprof
            zrt=prt(jrof,jlev)*pvbh(jlev)
            psgrtl(jrof,jlev)=zgphl(jrof,jlev)+((prtl(jrof,jlev)*zrt)
  204    CONTINUE
  202 CONTINUE
      RETURN
      END
```

Among the formal parameters, the following are variables: "prt", "prtl", "pprl", "psgrtl". The local arrays are always considered as variables, then "zgphl" is a variable, and the status of each local scalar identifier ("zgt", "zrt") is computed by the system.

The renaming ARPEGE/IFS convention is to add a "5" behind the identifier names corresponding to the value they had in the original code. We then obtain the following tangent linear code:

```
      SUBROUTINE gpgrptl (kproma, kprof, kflev, pvc, pvbh, prt, prtl,
     &pprl, porogl, psgrtl, prt5, prtl5, pprl5)

      REAL pvc(1:kflev), pvbh(0:kflev)
      REAL prt(1:kproma,1:kflev), prt5(1:kproma,1:kflev),
     &prtl(1:kproma,1:kflev), prtl5(1:kproma,1:kflev)
      REAL pprl(1:kproma), pprl5(1:kproma), porogl(1:kproma)
      REAL psgrtl(1:kproma,1:kflev)
      REAL zgphl(1:kproma,1:kflev)

      DO 102 jrof=1, kprof
         zgphl(jrof,kflev)=0.
  102 CONTINUE

      DO 104 jlev=kflev, 2, -1
         DO 106 jrof=1, kprof
            zgt5=prt5(jrof,jlev)*pvc(jlev)
            zgt=pvc(jlev)*prt(jrof,jlev)
            zgphl(jrof,jlev-1)=zgphl(jrof,jlev)-
     &       ((((prtl(jrof,jlev)*zgt5)+(prtl5(jrof,jlev)*zgt))*
     &       pprl5(jrof))+((prtl5(jrof,jlev)*zgt5)*pprl(jrof)))
  106    CONTINUE
  104 CONTINUE

      DO 202 jlev=1, kflev
         DO 204 jrof=1, kprof
            zrt5=prt5(jrof,jlev)*pvbh(jlev)
            zrt=pvbh(jlev)*prt(jrof,jlev)
            psgrtl(jrof,jlev)=zgphl(jrof,jlev)+((prtl(jrof,jlev)*
     &       zrt5)+(prtl5(jrof,jlev)*zrt))
  204    CONTINUE
  202 CONTINUE
      RETURN
      END
```

In the corresponding adjoint code, the header and the declarations are the same as in the tangent linear code. We then extract only the statements from the generated adjoint subroutine:

```
      DO 12 j4=1, kproma
         DO 16 j8=1, kflev
            zgphl(j8,j4)=0.
   16    CONTINUE
   12 CONTINUE
      zrt=0.
      zgt=0.
      DO 202 jlev=1, kflev
         DO 204 jrof=1, kprof
            zrt5 =prt5(jrof,jlev)*pvbh(jlev)
            zgphl(jrof,jlev)=zgphl(jrof,jlev)+psgrtl(jrof,jlev)
            prtl(jrof,jlev)=prtl(jrof,jlev)+(psgrtl(jrof,jlev)*zrt5)
            zrt=zrt+(psgrtl(jrof,jlev)*prtl5(jrof,jlev))
            psgrtl(jrof,jlev)=0.
```

```
C
          prt(jrof,jlev)=prt(jrof,jlev)+(zrt*pvbh(jlev))
          zrt=0.
  204     CONTINUE
  202 CONTINUE

      DO 104 jlev=2, kflev
        DO 106 jrof=1, kprof
          zgt5=prt5(jrof,jlev)*pvc(jlev)
          zgphl(jrof,jlev)=zgphl(jrof,jlev)+zgphl(jrof,jlev-1)
          prtl(jrof,jlev)=prtl(jrof,jlev)-(zgphl(jrof,jlev-1)
     &      *(pprl5(jrof)*zgt5))
          zgt=zgt-(zgphl(jrof,jlev-1)*(pprl5(jrof)*prtl5(jrof,jlev)))
          pprl(jrof)=pprl(jrof)-(zgphl(jrof,jlev-1)*
     &      (prtl5(jrof,jlev)*zgt5))
C
          prt(jrof,jlev)=prt(jrof,jlev)+(zgt*pvc(jlev))
          zgt=0.
  106     CONTINUE
  104 CONTINUE
```

5. Conclusion and perspectives

Odyssée has been a valuable tool for the study and the implementation of these automatic differentiation methods. Its toolkit structure and its programming language (CAML) enabled us to write general components that can be reused to implement other methods or strategies.

The study of the ARPEGE/IFS method allowed us to gain a better understanding of its applicability. The ARPEGE/IFS code have some properties that make it work: the separation of the code (two subroutines, one for the original code, one for the adjoint code) helps the generation of the adjoint code but forbids the modification of global variables. The control structures (conditional and iterative) follow the meanings we have given. For example, the number of iterations of a loop never depends on values where variables occur, and the code is rather "clean" (no programming tricks, `goto` statements ...). This leads us to believe that the ARPEGE/IFS method could be successfully used for other applications.

The *Odyssée* system is a prototype. The tests performed on ARPEGE/IFS subroutines are quite encouraging. Two directions of further research could be the study of more versatile storage/recomputation strategies (see for example (Griewank, 1992)) that can be tailored to a particular application and how to cope with more general iterative processes (see (Gilbert, 1992) for Newton-like processes).

REFERENCES

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

Bernardou, M., George, P. L., Hassim, A., Joly, P. et al. 1986. *MODULEF: a modular library of finite elements.* INRIA Rocquencourt, France.

Bischof, C., Carle, A., Corliss, G. and Griewank, A. 1992. ADIFOR: Automatic Differentiation in a Source Translator Environment. In: (Wang Paul S., editor) *Proceedings of the International Symposium on Symbolic and algebraic computation.*

Courtier, P. and Talagrand, O. 1987. Variational assimilation of meteorological observations with the adjoint vorticity equation I: Theory. *Q. J. R. Meteorol. Soc. 113*, 1311–1328.

Courtier, P. and Talagrand, O. 1987. Variational assimilation of meteorological observations with the adjoint vorticity equation II: Numerical Results. *Q. J. R. Meteorol. Soc. 113*, 1329–1345.

Gilbert, J. C. 1992. Automatic differentiation and iterative processes. *Optimization Methods and Softwares 1*, 13–21.

Griewank, A., Juedes, D. and Srivasan, J. 1990.

ADOL-C, a package for the automatic differentiation of algorithm written in C/C++. Technical report. Argonne National Laboratory, Illinois USA.

Griewank, A. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization methods and softwares 1*, 35–54.

Kubota, K. 1991. PADRE2, A Fortran precompiler yielding error estimates and second derivatives. In: (A. Griewank, G. Corliss, editor) *Automatic differentiation of algorithms: theory, implementation and application*, pp. 251–262.

Morgenstern, J. 1973. Algorithmes linéaires tangents et complexité. *Compte Rendu à l'Académie des Sciences* t.277:367.

Morgenstern, J. 1985. How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen. *Sigact News 16*, 60–62.

Rostaing, N. and Dalmas, S. 1992. Automatic FORTRAN programs analysis and transformation using a typed functional language. In: (R. Glowinski, editor) *Proceedings of the 10th International Conference on Computing methods in applied sciences and engineering*, 527–535. Nova Science Publisher.

Talagrand, O. 1991. The use of adjoint equations in numerical modelling of the atmospheric circulation. In: (A. Griewank, G. Corliss, editor) *Automatic differentiation of algorithms: theory, implementation and application*, pp. 169–180.

Weis, P., Mauny, M., Laville, A. and Suarez, A. 1990. *The CAML Reference Manual*. INRIA Rocquencourt, France.