An automatic differentiation platform: Odyssée

Christèle Faure

PolySpace Technologies, 28 Rue Estienne d'Orves, 92120 Montrouge, France

Abstract

Numerous automatic differentiation strategies can be imagined to produce all kind of derivative programs under a wide range of complexity constraints, but there is no way to prototype and evaluate them on real size applications with reasonable effort. Since the development of an automatic differentiation platform is prohibitively expensive, using an existing platform to share investments between the different research teams is a good solution. Odyssée is an open automatic differentiation tool that enables the development of program analysis as well as program transformation for automatic differentiation. It has been used to differentiate large size industrial programs (300 000 lines of Fortran 77) and to prototype diverse new automatic differentiation algorithms. Its source is now freely available, a cooperative research project can therefore be based on it without financial or contractual constraint.

Key words: Automatic differentiation, computational differentiation, Odyssee, open platform, source transformation tool.
PACS: 65Y20, 26B05

1 Motivation

Odyssée [FP98] [1] is an automatic differentiation tool for Fortran 77 programs developed by INRIA [2] and UNSA [3] since the early 90's. It has been designed from the beginning to perform reverse mode automatic differentiation as well as forward mode. The key point of this project is that it was intended to be, and has been, evaluated at each of its advancement step by industrial

[1] The Odyssée web page is available at http://www-sop.inria.fr/safir/SAM/Odyssee/odyssee.html

[2] INRIA is the French National Institute for Research in Computer Science and Control.

[3] UNSA is the University of Nice-Sophia Antipolis.

partners. The development of Odyssée was initiated in cooperation between INRIA, UNSA, Météo-France and the European Centre for Medium-Range Weather Forecasts (ECMWF). The initial design of Odyssée, and the implementation of procedural automatic differentiation techniques are described in [RD92,GR93,RDG93]. At that time Odyssée was not a general automatic differentiation tool but dedicated to meteorological programs which have a special structure that facilitates the hand-coding of adjoints. In fact, the original Odyssée was developed to automatize the procedural differentiation and to produce exactly the same adjoint code for procedures as the one hand-written by ARPEGE-IFS developers. ARPEGE-IFS (also known as Arpège [CFG+91,DDBC94]) is the Météo-France operational model that was developed as a joint effort between ECMWF and Météo-France. After automatic generation of sub-program adjoint, their combination was performed by hand to create the complete adjoint programs. This first prototype of Odyssée was also able to verify the ARPEGES-IFS programming rules as well as coherency constraints on the original sub-programs such as: this state variable is only modified once in the main loop.

Thereafter, the development of Odyssée was partially financed by an industrial contract with EDF-DER. [4] They required the extension of the inter-procedural automatic differentiation features: this mainly implied the addition of the call-graph, activity propagation, and the implementation of the standard forward and reverse mode algorithm as shown in [RSH94]. The first application of this version to the mono-dimensional mockup THYC1D code gave good results as described in [DEFG96], and has been the basis of many investigations on split and joint reverse mode algorithms [CG00,FC01]. The application of Odyssée to the industrial program THYC3D [JR95] required the implementation of a non standard reverse mode algorithm called the flow reversal algorithms (detailed in section 4.1) to manage goto statements. Finally, the excessive memory requirement of this first adjoint program for THYC3D lead to some intensive work on checkpointing [FD98,Fau00,FC01] and optimal tracing [FN01].

Other industrial partners like Dassault Aviation [MRSM96,MMRS96,FDF00], Essilor, Elf [Tad99] and Alenia Aeronotica [SF00] financially supported the general development of Odyssée, as well as some specific studies: parallelism [FDF00], iterative loops [GF02], parallel loops [HFH01]. These research directions could be pursued thanks to the modular structure of Odyssée that allows at the same time a robust and efficient implementation, as well as rapid experimentation with new automatic differentiation algorithms.

Odyssée is a source-to-source transformation tool that implements automatic differentiation as a specific program transformation: it takes as input a Fortran 77 program and returns the derivative Fortran 77 program. Each compila-

[4] Electricité de France-Direction des Etudes et Recherches.

tion unit is analysed and structured into an abstract syntax tree, a control flow graph and a symbol table containing information about all existing identifiers. The whole program is a set of compilation units, the *may-call* relation that relates two sub-programs is encoded into the call-graph: a sub-program p_1 *may-call* p_2 if there is a call to p_2 in the body of p_1. All the program analysis, as well as the program transformations are performed on this structure.

This paper gives some insight into Odyssée internal structure: general organization, data structures, derivation and analysis algorithms, as well as the global derivation process. Odyssée's source is now freely available as a set of OCAML[5] sources as well as some compilation and installation scripts and can be obtained from ftp://www.sop-inria.fr/tropics/odyssee. Odyssée is the only open-source automatic differentiation tool for the differentiation of Fortran 77 codes based on source transformation. ADOL-C [GJM+96] and FADBAD/TADIFF [SB00] are open-source automatic differentiation tools for the differentiation of C/C++ codes based on operator overloading. This makes Odyssée the only source-to-source automatic differentiation tool that can be used as a development platform for new automatic differentiation algorithms.

2 The organization of Odyssée

Odyssée is organized as a toolkit of components providing basic services necessary for the implementation of automatic differentiation algorithms, but it could also be used for other source transformations. It is composed of two kinds of tools: basic tools that form the front-end and back-end (the lexical analyser, the parser and the Fortran 77 code generator) and general tools that form the differentiation kernel and can be specialized (tree, control flow graph, or call graph, traversal and transformation). This flexibility is due to specific features of the programming language OCAML developed at INRIA. OCAML is a functional language (functions can be arguments and results), strongly typed (each OCAML 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 OCAML function may be parametrized by the transformation to apply - that transformation being another OCAML function. Let consider for example, the function that transforms each statement into a new block of statements and replicates the flow graph to perform forward mode differentiation. This function can be parametrized by the transformation applied at the statement level to implement one or several directional derivatives. The strong typing ensures a certain level of safety of the performed transformation. Other OCAML features are well adapted to our

[5] OCAML stands for Objective-CAML. It is a language of the ML family developed at INRIA. For more details see http://pauillac.inria.fr/ocaml/index.html

purpose: the concrete types describes the `Fortran 77` syntax and the pattern matching mechanism symbolically applies the transformation rules. This last feature also makes the source code of `Odyssée` self-contained as the rules appear syntactically, and this greatly improve its readability.

In `Odyssée`, each `Fortran 77` unit is represented as an abstract syntax tree (AST) with an associated symbol table. To obtain this structure, the steps are similar to those of a compiler. First the lexical analyser (an existing one written in C has been reused) recognizes `Fortran 77` keywords and produces a sequence of tokens which are then parsed to produce an AST. The parser employs the `OCAML` interface with the `YACC` parser generator which allows for conveniently associate an abstract syntax tree with a program. Some information about the identifiers cannot be deduced from the syntax alone and require a semantic static analysis. All semantic information deduced for each identifier is stored in the symbol table. The identifiers are classified as local, global, or dummy arguments, and described by their semantic type (scalar, array, sub-program, ...) and their `Fortran 77` type.
The control flow graph is built from the abstract syntax tree and represented as a graph: all the statements in the tree are associated with a unique index, and the indices are used in the graph to encode the relation *may-be-successor*. A statement s_2 *may-be-successor* of s_1 if s_2 may be evaluated directly after s_1. The program is represented as a set of units together with a call-graph that is computed by a simple traversal of all program units and a record of the *may-call* arcs. The semantic analysis that produces the control flow and the call graphs are performed at compile time and can only report *may* information. If a call to p_2 appears in p_1 and is guarded by an if-statement, it is impossible, at compile time, to know whether or not it will be really called at run time.

3 Program Analysis

`Odyssée` applies optimized transformations on the original program to generate efficient derivative program in one pass. These optimized transformations are based on information extracted from the original program by a program analysis devoted to automatic differentiation. Within `Odyssée`, all the program analysis are performed on variables: all components $v[i]$ of an array v are collapsed on a symbolic variable v. This approximation may lead to a lack of precision if the components of an array contain completely independent information, but is in general a good compromise between precision and efficiency. Such an approximation could be applied on components of a structure when extended to a new language. Al the analyses described in this section are inter-procedural analyses.

3.1 Differentiation Dependency Analysis

The dependency analysis used in automatic differentiation, so-called differentiation dependency analysis, is a restriction of the general dependency analysis. Only dependencies coming from assignments or sub-program calls are considered: all the variables read by the modified value *influence* all the variables written by the statement. For example, the standard dependencies created by the evaluation of statement

```
if y > 0
then z = x1
else z = x2
```

are $\{(z,v)|v \in \{y, x1, x2\}\}$ where the presence of a couple (z,x) in the list means that variable z *influences* y (or y *depends on* x) . The general dependency list contains a dependency between z and y whereas the differentiation dependency analysis ignores this dependency and computes the sub-set $\{(z,v)|v \in \{x1, x2\}\}$.

Denote by I_i the set of input variables of statement i (read by statement i) and by O_i the set of output variables of i (written by statement i). Denote by D_i the set of couples $(y,x) \in I_i \times O_i$ such that x *influences* y (in the sense described above) through the evaluation of assignment or call statement i. The operator $\cup_D$ denotes the union operator on list of dependency couples: it performs a strong update or a standard update of the dependency list, depending on the modified path. If the written path is a variable z, $\cup_D$ removes all the influences on z from the old dependency before adding the new dependency: it is a strong update. If the written path is an array access $z[i]$, $\cup_D$ operates a merge between the old influence on z and the new one: it is a standard update.

The dependency analysis decomposes into a five levels analysis:

Step-1 Elementary information:
The dependencies D_i^+ after statement i are computed from the dependencies D_i^- using equation (1).

$$D_i^+ = D_i^- \cup_D \{(y,x) \in I_i \times O_i | x \textit{ influences } y\} \tag{1}$$

Step-2 One-step forward propagation:
Denote $pred_i$ the set of all predecessors j of statement i in the control flow graph. The differentiation dependency information before statement i denoted by D_i^- is computed from D_j^+ with equation (2).

$$D_i^- = \cup_{D\, j \in pred_i} D_j^+ \tag{2}$$

Step-3 Sub-program forward propagation:
To initialize the propagation within a sub-program, one must consider that its dummy arguments and global variables are independent from one another, the set of dependencies is therefore initialized to the empty set. The equations (1) and (2) described above are applied for all statements in the sub-program in an order compatible with the evaluation order (represented by the *may-be-successor*): from the first statement until the last statement of the sub-program throughout all intermediate statements. This pass computes the information D_i^- and D_i^+ for all statements i in the sub-program.

Step-4 Program forward propagation:
The dependencies of a sub-program can only be determined if all the sub-programs it *may-call* have been analysed: this analysis must therefore be performed bottom-up in the call-graph. For each sub-program an abstract of the dependencies that concern dummy arguments and global variables is recorded in order to prepare the analysis of all its *callers* in the call-graph.

Step-5 Fixed-point iteration:
During the program forward propagation, the backward arcs of the control flow graph have not been considered. The information associated to the origin of backward arcs should be taken into account in equation (2), but due to the forward propagation, it is not yet computed at the time it is required. The propagation of this information is somehow delayed to the next pass. As a result the program forward propagation must be iterated until a fixed-point is reached. The information is stabilized if the elementary information D_i^- and D_i^+ is the same for two successive iterations. Within `Odyssée`, a global fixed-point iteration is computed: the whole program is analysed and the iteration is performed on the whole program.

The differentiation dependency information describes the program it and do not depend on the differentiation parameters (dependent or independent variables). The abstract dependencies recorded to perform the bottom-up call-graph traversal can be written and read by `Odyssée` as a specific file in place of the analysis itself. This saves further computation when several derivatives of the same program are to be generated. This mechanism is also used to allow users to give dependency information for black-box routines or to replace the dependencies automatically generated by more precise ones if required.

3.2 Activity Propagation

A variable v is active if it does depend on the input variables for which derivatives are desired (so-called independent variables), and does impact the output variables for which derivatives should be computed (known as dependent variables). There is no sense computing derivatives for non-active (also called passive) variables as we know them to be zero.

Denote by $\overrightarrow{A}_i^-$ the set of variables influenced by the independent variables before the execution of statement i and $\overrightarrow{A}_i^+$ the set of variables influenced by the independent variables after statement i.

The same five level algorithm as for the dependency analysis applies here:

Step-1 Elementary information:
The set of active variables after the execution of statement i is computed from the active variables before i as shown in equation (3). We define the predicate $dep_i(x, y)$ as true if and only if x *influences* y which is equivalent to $(y, x) \in D_i^-$.

$$\overrightarrow{A}_i^+ = \overrightarrow{A}_i^- \cup \{y \in O_i | \exists x \in (\overrightarrow{A}_i^- \cap I_i) \& dep_i(x, y) = true\} \tag{3}$$

Step-2 One step forward propagation of the elementary information:
The activity information before statement i denoted by $\overrightarrow{A}_i^-$ is computed from $\overrightarrow{A}_j^+$ with equation (4).

$$\overrightarrow{A}_i^- = \cup_{j \in pred_i} \overrightarrow{A}_j^+ \tag{4}$$

Step-3-4-5 This analysis must be performed by a top-down propagation in the call-graph and a forward propagation into the control flow graph. Here also a global iterative computation is performed to reach a fixed-point.

In the same way the backward activity propagation could be implemented in `Odyssée` using the elementary equations (5) and (6) and a backward propagation across the program (top-down in the call-graph and backward in the control flow graph). Denote by $succ_i$ the set of indices of all successor statements of i in the flow graph.

$$\overleftarrow{A}_i^- = \overleftarrow{A}_i^+ \cup \{x \in I_i | \exists y \in (\overleftarrow{A}_i^+ \cap O_i) \& dep_i(x, y) = true\} \tag{5}$$

$$\overleftarrow{A}_i^+ = \cup_{j \in succ_i} \overleftarrow{A}_j^- \tag{6}$$

The activity propagation is the combination of both analyses, and the set of active variables at statement i is denoted by A_i such that $A_i = \overleftarrow{A}_i \cup \overrightarrow{A}_i$. For historical reasons, `Odyssée` only computes the forward activity, witch is a partial activity information and lead to useless computations.

The activity propagation is used to avoid the generation of derivative instructions that compute always zero values.
Let consider a general assignment i written as $p(v) := e(v_1, ..., v_n)$ where $p(v)$ is a general path constructed from variable v and $e(v_1, ..., v_n)$ a general expression using variables $\{v_1, ..., v_n\}$. Statement $p(v) := e(v_1, ..., v_n)$ is active if

and only v is active after statement i which means that $v \in A_i^+$. If i is active, the derivative assignment i' will be generated from the local Jacobian matrix $J - i$ such as $J_i = \frac{\partial e(v_1,...,v_n)}{\partial \{v_1,...,v_n\} \cap A_i^-}$.
The activity mask of a sub-program is the set of active dummy arguments or global variables. In `Odyssée`, a maximal activity mask is computed for a sub-program: the activity mask of a sub-program p is the union of all the activity masks computed at each point code where p is called. Each sub-program is differentiated with respect to its maximal activity mask, thus only one derivative is associated to each sub-program. Each call is whether active or passive: if it is passive no derivative call is generated and if it is active a derivative call is generated together with zero assignment of actually passive elements of the active mask. This is a choice, and the maximal activity mask could be replaced by several exact activity masks, but then one derivative would be associated to each activity mask, thus potentially several derivatives would be associated to each sub-program.

3.3 Propagation of Required Variables

In reverse mode, or in forward mode when several directions are propagated, some original values must be recorded to be used during the computation of partial derivatives. This problem can be looked at from a value or a variable point of view as described in [Fau02b] pages 31-35. In this paper, the variable point of view is considered: [FN01] defines the notion of *required variable* and describes the three components of the trace. We recall here that a variable is required if it is used by some derivative statement for the computation of local partial derivative, index or control information.

Let now consider only the variables required for the computation of the partial derivatives. For each statement i, the set of required variables is N_i and is mathematically described by equation (7).

$$N_i = \{x \in I_i | \exists y \in O_i, \frac{\partial^2 y}{\partial^2 x} \neq 0\} \tag{7}$$

This equation means that only variables that appear non linearly in the original statement will appear in the derivative statement, and are therefore required.

The values to be recorded in the trace are those of the required variables that may be modified between their definition and their use in the derivative computation. Let call the scope of a variable v the interval of statement indices $[i..j]$ such that i is the index of the statement that computes v and j is the index of the statement that recomputes v. Variable v must be recorded after

statement i and before statement k (where $k < j$) if it is used in statement k.

We denote by L_i the set of variables such that statement i belongs to their current scope, $N_i \subset I_i$ the set of variables required by statement i and O_i the set of output variables of statement i.

$$L_i^+ = (L_i^- \cup N_i) \setminus O_i \tag{8}$$
$$L_i^- = \cup_{j \in pred_i} L_j^+ \tag{9}$$

The computation of required variables can be described as the forward propagation of the scopes of all original variables using equations (8) and (9), iterated until a fixed-point is reached. The computation of required variables is described in [FN01] and has been implemented in a prototype of `Odyssée` and evaluated on `THYC3D`. In this paper we have shown that on this large industrial program, the trace size can be reduced by a factor of 5.

After this first analysis, all the variables to be recorded are known as well as the interval of statements during which the recording must be performed. If one choose to record the variables before each modification (at the end of their scope), the set of variables to be recorded before statement i denoted by T_i can be computed using equation (10).

$$T_i = L_i^+ \cap O_i \tag{10}$$

As above, the information N_i, O_i and $pred_i$ can be pre-computed for each statement and stored into a table. This analysis can be simply modified to store required variables before their first use, or to group several store statement to diminish the number of accesses to the trace. The generation of optimized tracing statements by required variables is not implemented within `Odyssée`, but could easily be done.

The three components of the trace (local control, indices, partial derivatives) can be dealt with by a simple redefinition of N_i and L_i, and use the same equations. Treating them simultaneously but independently would allow for more diverse tracing strategies.

4 Program Transformation

Automatic differentiation can be viewed as a particular kind of source-to-source transformation. The main difference between automatic differentiation and other program transformations is that the generated program computes

not only the original values, but other values derived from the original ones by some rules of mathematical calculus. This method is in general implemented by what we call syntactical differentiation: the program structure is replicated or reversed but there is a perfect match between an original statement and its derivative one. The forward and reverse mode differentiation can be implemented using such an approach. As for the reverse mode, syntactical differentiation leads to an important limitation on the original program, because all statements cannot be reversed. For example, `goto` statements are not syntactically reversible but can be dealt with using a reversal of the flow graph instead of the tree reversal.

`Odyssée` implements the standard scalar forward mode. Each original unit p_i is differentiated into a derivative unit p_i' that computes a directional derivative. This simple algorithm has not been extended to a vector forward mode as in `ADIFOR [BCC+92]` or to a Taylor development as in `PADRE2 [Kub96]`. Such extensions where not required by our project partners even though these axes would have been interesting to investigate.

4.1 Two Reverse Mode Algorithms

`Odyssée`, and more recently `odyper [Fan02]` and `tapenade [Gro02]` implement a reverse mode based on the transposition of the flow graph whereas the standard algorithm implemented in `Odyssée`, `TAF [Fas00]` also known as `TAMC [Gie99]`, `ADIFOR 2 [BCKM96]` reverses the abstract syntax tree. We have implemented the flow reversal algorithm within `Odyssée` to be able to differentiate industrial programs like `THYC3D` that make use of `goto` statements.

Both reverse mode algorithms implement the so-called joint reverse mode: for each original sub-program p_i, the derivative program calls p_i^s, the forward component of its cotangent code, followed by $p_i'^r$, the reverse component. The sub-program p_i^s computes the same output as p_i and stores the trace of the execution (partial derivatives, indices, ...). The sub-program $p_i'^r$ restores this information and computes the adjoint values. One can see the transformation on a simple example in [FP98] pages 23-26. In the syntactical algorithm the trace of the execution contains variables required for the computation of the partial derivatives and the indices, the flow reversal based algorithm traces the execution path plus variables required by the evaluation of the partial derivatives. The trace of the flow reversal is therefore larger, but it is the only way to deal with dynamic control. This is the reason why the existing implementation of reverse mode in `Odyssée` uses the syntactical algorithm and only if the `-goto` option is activated the flow reversal algorithm is used. A more efficient algorithm should be able to merge both approaches: at least at the program level (some units syntactically reverted, and the other flow

reverted), and even at the unit level (some statements syntactically reverted, and the other flow reverted).

In the tree reversal algorithm the trace is recorded within `Fortran 77` arrays whereas when the flow reversal algorithm, the trace of required variables is performed by external calls to dynamic allocation functions written in `C`. This implementation is not satisfying as it seems to stop `Fortran 77` optimization, and interrupt the derivative execution too often by memory actions. It would better be implemented as a two levels management: first the values are stored in (or retrieved from) a `Fortran 77` buffer, and second the buffer is written in (or read from) memory using `C` procedures. This two levels method could be further optimized by grouping trace instructions to limit the number of interruptions of derivative calculations.

4.2 Checkpointing

The checkpointing strategy first proposed by A. Griewank in [Gri92], but also studied at INRIA [GPRS96] has been integrated into `Odyssée` in such a way that user need to perform just a few modifications on the original program to make use of it. This optimal schedule can only be applied on loops, under the hypothesis that all iterations are equivalent in terms of execution time and memory requirement. Moreover, the actual (or at least maximal) number of iterations of the loop must be known at compile time. The only implementation constraint is that the body of the loop to be check-pointed must be extracted and put into a sub-program: this is the only modification required by the user. Then `Odyssée` automatically computes the sub-set of program variables have to be check-pointed and generates all the pieces of codes necessary to run the optimal checkpointing schedule for the given number of checkpoints: the scheduler chooses when to store and when to re-compute using the same schedule as in [Gri92], the checkpointing management sub-programs saves and restores the checkpoints on files. A sample of check-pointed loop generated by `Odyssée` can be seen in [FP98] pages 26-29. This implementation is robust, but could be generalized by coupling it with the specific package `treeverse/revolve [GW99]` to make profit of potential new features and by generating checkpoint management sub-programs using diverse libraries.

4.3 Generalization of automatic differentiation algorithms

`Odyssée` applies sensible default strategies, but as shown in [BH96], [Gri00], [Fau02a], [Fau02b] a lot of predefined parts of differentiation algorithms could be made differentiation strategy parameters, or library implementation choices. A differentiation strategy should contain the following ingredients:

traversal mode
traversal direction
tracing mode
derivative variable generator
sub-program level differentiation
block-level differentiation
statement level differentiation.

The traversal mode and direction indicate the way the program is traversed. The tracing mode indicates how the trace is built: required variables (or values), buffer of required variables (or values), before modification or before use, and which component are traced (control, array indices, partial derivatives). The derivative variable generator generates a derivative variable from the original one. Standard automatic differentiation tools replicate the original definition: the derivative variable has the same type (real, float, int, ...), the same status (local, global, dummy argument), and the same size (scalar, array, ...). The sub-program, block and statement level differentiation indicate the kind of derivative computed and the way the derivatives are composed to compute the derivatives of the dependent values with respect to the independent values.

Within `Odyssée` all sub-programs of a program are differentiated using the same strategy, this should also be generalized and each sub-program could be associated a strategy. In particular in `OCAML` this could be easily implemented by passing the respective strategies as parameters.

Denote by i the index of current statement, by J_i the local (statement level) Jacobian matrix, by d_i the local direction and by $d_i^\star$ the local adjoint vector value. The standard forward mode of `Odyssée` could be described as:

traversal mode: tree,
traversal direction: forward,
tracing mode: none,
derivative variable generator: replicates original,
sub-program level differentiation: none,
block-level differentiation: none,
statement level differentiation: $J_i \times d_i$.

The two reverse mode algorithms differ on the traversal mode: one is based on a tree traversal whereas the other uses a flow traversal, but are identical for the rest.

traversal mode: tree (for the standard algorithm) or flow (for the `-goto` algorithm),
traversal direction: reverse,
tracing mode: all modified variables before modification,

derivative variable generator: replicates original,
sub-program level differentiation: joint,
block-level differentiation: none,
statement level differentiation: $J_i^t \times d_i^\star$.

5 Conclusion

`Odyssée` is the only existing open source-to-source automatic differentiation platform: it has been designed from the beginning for this purpose, the language `OCAML` it is written in is quite easy to read and strongly typed. `Odyssée`'s source is available free of charge at `ftp://ftp-sop.inria.fr/tropics/odyssee`. All these characteristics make it a possible base for a cooperative development of new automatic differentiation algorithms. `Odyssée` could be a general automatic differentiation platform: with a little effort a lot of new program analysis and automatic differentiation algorithms could be implemented. Changing the target language could be done by the adaption of the existing front-end and back-end to the `Odyssée` intermediate language and the coupling of this front-end and back-end with the `Odyssée` kernel.

The optimization of the generated derivative program is the key point to go from pinpoint use of automatic differentiation to a general use. The existing optimizations performed by current automatic differentiation tools should be expended in the following way. Forward and backward activity propagation is used to avoid the generation of useless derivative instructions. This optimization could be replaced by a forward propagation of constants and the slice of the derivative program with respect to the derivatives of the dependent variables. In the same way the required propagation described in [FN01,GK01] could be replaced by some liveness analysis applied to the derivative program. Slicing and constant-propagation are general optimization technics: their use would discard useless original instructions as well as derivative ones, thus the generated derivative program would be more efficient. But generating a naive derivative program and optimizing it afterwards could be costly because optimization time is related to the number of variables and statements, and the differentiation process enlarges these parameters by a factor of at least two compared to the length of the original program. The best solution would probably use both technics: optimized-differentiation and general optimization. New program analysis for solving particular problems like automatic detection of sparse Jacobian matrices [TEF98], could become also become necessary. The design of `Odyssée` allows the extension of the optimization module to general program optimization algorithms besides automatic differentiation specific transformations.

References

[BCC+92] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[BCKM96] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs. *IEEE Computational Science & Ingenieering*, 3(3):18–32, 1996.

[BH96] C. Bischof and M. Haghihat. Hierarchical approaches to automatic differentiation. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 83–94. SIAM, 1996.

[CFG+91] P. Courtier, C. Freydier, J.-F. Geleyn, F. Rabier, and M. Rochas. The ARPEGE project at Météo-France. In *Proceedings of the ECMWF seminar on Numerical Methods in Atmospheric Models, Reading, 9-13 September 1991*, pages 193–232, 1991. Available from ECMWF.

[CG00] I. Charpentier and M. Ghémirès. Efficient adjoint derivatives: Application to the atmospheric model Meso-NH. *Optimization Methods and Software*, 13(1):35–63, 2000.

[DDBC94] M. Deque, C. Dreveton, A. Braun, and D. Cariolle. The ARPEGE/IFS Atmosphere Model A contribution to the French Community Climate Modeling. *Climate Dynamics*, 10:249–266, 1994.

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

[Fan02] S. Fantino. *Odyper*, 2002. `http://www-math.unice.fr/~fantino/old_index.html`.

[Fas00] FastOpt. *TAF: Transformation of Algorithms in Fortran*, 2000. `http://www.FastOpt.de`.

[Fau00] C. Faure. Generating Adjoints of Industrial Codes with Limited Memory. *Flow, Turbulence and Combustion*, 65(3-4):453–467, 2000.

[Fau02a] C. Faure. Adjoining Strategies for Multi-layered Programs. *Optimization Methods and Software*, 17(1):129–164, 2002.

[Fau02b] C. Faure. *Dérivation de Programme Assistée par Ordinateur*. Mémoire d'Habilitation à Diriger des Recherches (HDR), University of Nice-Sophia Antipolis, 2002.

[FC01] C. Faure and I. Charpentier. Comparing Global Strategies for Coding Adjoints. *Scientific programming*, 9(1):1–11, 2001.

[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.

[FDF00] C. Faure, P. Dutto, and S. Fidanova. Automatic Differentiation and Parallelism. In *Procceedings of The 3rd European Conference on Numerical Mathematics and Advanced Applications, Jyväskylä, Finland, July 26-30, 1999*, pages 478–485. World Scientific, 2000.

[FN01] C. Faure and U. Naumann. Minimizing the Tape Size. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer Verlag, 2001.

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

[GF02] A. Griewank and C. Faure. Reduced functions, gradients and hessians. *Numerical Algorithms*, 30:113–139, 2002.

[Gie99] R. Giering. *TAMC: Tangent linear and Adjoint Model Compiler, Users manual 1.4*, 1999. `http://www.autodiff.com/tamc`.

[GJM+96] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM TOMS*, 22:131–167, 1996.

[GK01] R. Giering and T. Kaminski. Generating recomputations in reverse mode. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, pages 283–290. Springer-Verlag, 2001.

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

[GR93] A. Galligo and N. Rostaing. Une nouvelle technique de Calcul Formel appliquée à un problème d'Optimisation en Météorologie. In J.A Désidéri, L. Fezoui, B. Larrouturou, and B. Rousselet, editors, *Optimisation et Contrôle. Actes du colloque en l'honneur du soixantième anniversaire du professeur Jean Cea*, pages 143–160. cépaduès-éditions, 1993.

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

[Gri00] A. Griewank. *Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[Gro02] Tropics Group. *Tapenade : On-line Automatic Differentiation Engine*, 2002. `http://www-sop.inria.fr/tropics/tapenade.html`.

[GW99] A. Griewank and A. Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or ajoint mode of computational differentiation. *ACM Trans. Math. Software*, 26(1):19, 1999. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[HFH01] L. Hascoet, S. Fidanova, and C. Held. Adjoining Independent Computations. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, pages 299–304. Springer-Verlag, 2001.

[JR95] T. Jouhanique and P. Rascle. A fifth equation to model the relative velocity in the 3-d thermal-hydraulic code thyc. In R.C. Block and F. Feiner, editors, *Proceedings of the sixth Nuclear Reactor Thermal-hydraulic (NURETH) Congress*. American Nuclear Society, Nuclear Regulatory Commission Publication., 1995.

[Kub96] K. Kubota. PADRE2 - FORTRAN precompiler for automatic differentiation and estimates of rounding errors. In M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 463–471, Philadelphia, 1996. SIAM.

[MMRS96] B. Mohammadi, J.-M. Malé, and N. Rostaing-Schmidt. Automatic differentiation in direct and reverse modes: Application to optimum shapes design in fluid mechanics. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*, pages 309–318. SIAM, 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*, pages 87–91. John Wiley & Sons, Ltd, September 1996.

[RD92] N. Rostaing and S. Dalmas. Automatic analysis and transformation of FORTRAN programs using a typed functional language. In R. Glowinski, editor, *Proceeding of the 10th international conference on Computing methods in applied sciences and engineering*, pages 527–535. Nova Science, 1992.

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

[RSH94] N. Rostaing-Schmidt and E. Hassold. Basic functional representation of programs for automatic differentiation in the Odyssée system. In F.X. Le Dimet, editor, *Proceedings of the workshop on High Performance Computing in the Geosciences*, Les Houches (France), 1994. Kluwer Academic Publishers, NATO ASIE SERIES.

[SB00] O. Stauning and C. Bendtsen. *FADBAD-TADIFF*, 2000. `http://www.imm.dtu.dk/fadbad.html`.

[SF00] V. Selmin and C. Faure. The Role of Sensitivity Analysis in Aircraft Design. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer Verlag, 2000.

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

[TEF98] M. Tadjouddine, F. Eyssette, and C. Faure. Sparse Jacobian Computation in Automatic Differentiation by Static Program Analysis. In G. Levi, editor, *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 311–326. Springer-Verlag, September 1998.