TAPENADE 2.1 user's guide

Laurent Hascoët, Valérie Pascual

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

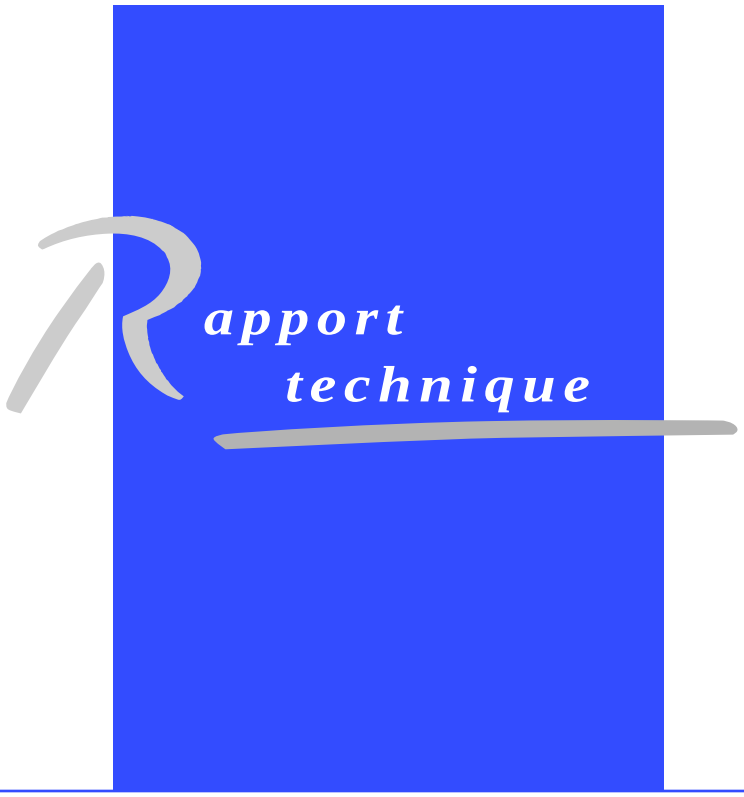
TAPENADE 2.1 user's guide

Laurent Hascoët — Valérie Pascual

N° 0300

Septembre 2004

Thème NUM



ISSN 0249-0803

INRIA
SOPHIA ANTIPOLIS

TAPENADE 2.1 user's guide

Laurent Hascoët* , Valérie Pascual*

Thème NUM — Systèmes numériques
Projet Tropics

Rapport technique n° 0300 — Septembre 2004 — 78 pages

Abstract: This is the user's manual for the version 2.1 of the Automatic Differentiation tool TAPENADE. Given a source computer program that computes a differentiable mathematical function F, TAPENADE builds a new source program that computes some of the derivatives of F, specifically directional derivatives ("tangent mode") and gradients ("reverse mode"). This report summarizes the mathematical justifications of Automatic Differentiation, then describes in full detail the differentiation model that TAPENADE implements. Our goal is to give the users of TAPENADE a precise understanding of the actions and choices made while differentiating programs, so as to improve their confidence in the produced source programs. This report documents all the available options and parameterizations that the users can give to TAPENADE, and conversely all the diagnosis and requirements that TAPENADE may issue towards the users. After a brief description of TAPENADE's architecture and performances, this report describes more fully the validation and improvement techniques for differentiated codes.

Key-words: automatic differentiation, algorithmic differentiation, adjoint, gradient, optimization, inverse problems, static analysis, data-flow analysis, compilation

* Projet Tropics, INRIA Sophia-Antipolis

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Manuel de l'utilisateur de TAPENADE 2.1

Résumé : Ce rapport est le manuel d'utilisation de la version 2.1 du logiciel de Différentiation Automatique TAPENADE. Etant donné un programme source qui calcule une fonction mathématique différentiable F, TAPENADE construit un nouveau programme source qui calcule certaines dérivées de F, et plus particulièrement des dérivées directionnelles ("mode direct") et des gradients ("mode inverse"). Ce rapport rappelle succintement les justifications mathématiques de la Différentiation Automatique, puis décrit en détail le modèle de différentiation implémenté dans TAPENADE, dans le but d'améliorer la compréhension et la confiance des utilisateurs dans le code différentié. Ce rapport documente les options de la commande TAPENADE, les fichiers de configuration disponibles, ainsi que les diagnostics et résultats produits par l'outil. Ce rapport décrit rapidement l'architecture de TAPENADE et ses performances, puis discute plus en détail des techniques de validation et d'amélioration des programmes différentiés produits par TAPENADE.

Mots-clés : différentiation automatique, adjoint, gradient, optimisation, problèmes inverses, analyses statiques, analyses data-flow, compilation

Contents

List of Figures

1 INTRODUCTION

As computational power increases, the domains of computational *simulation*, *optimization*, and *inverse problems* are developing rapidly. They widely use *derivatives*. When a function is already discretized and solved, Automatic Differentiation (AD) can return its derivatives without going back to the discretization step. AD transforms a program that computes or simulates a mathematical vector function into a new program that computes derivatives of this function. Further information about AD can be found in the monograph [15] and in the collection of articles from the past AD conferences [16, 2, 5, 3]. These articles cover most topics related to AD, from theoretical questions in computer science and numerical science to AD tools development and industrial applications.

AD is a program transformation, and is therefore performed by software tools similar to compilers or parallelizers. This is the user's guide to TAPENADE, an AD tool with a strong focus on the "*reverse*" mode, that computes *gradients*. Our guideline in this development is to reuse and transpose technology from the compilation field [1] to AD, in order to produce efficient differentiated code that can compete with hand-coded derivatives.

This is the user's guide to version 2.1of TAPENADE. The main difference from previous version 2.0 is that program sources in FORTRAN95 [22] are accepted in addition to FORTRAN77. At the moment when we write this report, however, some development is not yet terminated and the pointer and memory allocation part of FORTRAN95 is still missing. See section 10 about the others developments to come soon.

For readers who know AD and other AD tools, section 2 gives a quick reference to the classical characteristics of TAPENADE. After a brief description in section 3 of the theoretical basis of AD, section 4 describes the AD model implemented by TAPENADE, showing how it relates to the theoretical description. This part is based on concrete examples to gain understanding of programs produced by TAPENADE. Section 5 focuses on refinements to the AD model for adjoint codes. There are two ways to use TAPENADE, either as a shell command from the command line or from a `Makefile`, or as a server on the web with a HTML graphical interface. Section 6 presents the fundamental command line mode, with the available options and configuration files. Section 7 presents the alternative web interface. Section 8 gives a quick overlook of TAPENADE's internal architecture, and discusses the performances. Section 9 addresses the problem of validation and Section 10 concludes with known problems and further developments to come in TAPENADE.

2 QUICK REFERENCE GUIDE

Automatic Differentiation Tool

Name: TAPENADE version 2.1

Date of birth: January 2002

Ancestors: Odyssée 1.7

Address: `www.inria.fr/tropics/tapenade.html`

Specialties: AD Reverse, Tangent, Vector Tangent, Restructuration
Reverse mode Strategy: Store-All, Checkpointing on calls
Applicable on: FORTRAN95, FORTRAN77, and older
Implementation Languages: 90% JAVA, 10% C
Availability: Java classes for Linux and Solaris, or Web server

Internal features: Type-Checking, Read-Written Analysis,
Forward and Backward Activity, Adjoint Liveness analysis, TBR, Reduced Snapshots

Figure 1: TAPENADE*'s ID card*

For readers who know AD and other AD tools, here is a quick description of TAPENADE, about the usual features of AD tools. Figure 1 just puts it in a funnier way! TAPENADE is a software tool for automatic differentiation of FORTRAN95 and FORTRAN77 programs, with the source regeneration approach. TAPENADE is not based on overloading. It produces tangent ("Jacobian times vector") codes and adjoint ("Transposed Jacobian times vector") codes. In its so-called vector or multi-directional mode, it produces tangent code for several directions at a time. Several tactics and analyses are used to produce better code. Some are classical, others are specific. Internal implementation is mostly JAVA. There are two ways you can use TAPENADE: as a web server, always with the latest improvements, or as a downloadable tool to be called from your command line or Makefiles. Of course, TAPENADE is "alive", and therefore evolves constantly.

3 AUTOMATIC DIFFERENTIATION OF COMPUTER PROGRAMS

Automatic or Algorithmic Differentiation (AD) differentiates *programs*. An AD tool takes as input a source computer program P that, given a vector argument $X \in I\!R^n$, computes some vector function $Y = F(X) \in I\!R^m$. The AD tool generates a new source program that, given the argument X, computes some derivatives of F. In a sense, an AD tool behaves largely like a compiler, which first understands the meaning of the original program, represents it in an internal form, performs analyses to get a deeper understanding, and generates an object code. The main differences from a compiler are:

- the meaning of the object code is different from the source. AD defines this new meaning as a mathematical transformation (differentiation) on the original function.
- the produced object code is generally not in machine form, but rather in source form again. This difference is in principle minor, but most users prefer to see the results of differentiation in their favorite source language, and this allows the compiler, later on, to run the usual optimization steps on the differentiated code.

Why is such a transformation possible? AD actually relies on a number of (reasonable) assumptions on the program P. To begin with, AD assumes that P represents all its possible run-time sequences of instructions, and it will in fact differentiate these sequences. Therefore, the *control* of P is put aside temporarily, and AD will simply reintroduce this control into the differentiated program. In other words, P is differentiated only piecewise. Experience shows that this is reasonable in most cases, and going further is still an open research problem.

Then, any sequence of instructions is identified with a composition of vector functions, each of which is assumed differentiable. Thus, for a given control:

$$\begin{array}{lcl} \mathtt{P} & \text{is} & \{I_1; I_2; \ldots I_p;\}, \\ F & = & f_p \circ f_{p-1} \circ \cdots \circ f_1, \end{array} \tag{1}$$

where each f_k is the elementary function implemented by instruction I_k. In general, the functions implemented by the arithmetic operations of the programming language are indeed differentiable. However in some rare cases, this is not true. For example the square root is defined on a null argument, and its derivative is not. Again experience shows that this assumption is verified in most cases.

Finally, AD simply applies the chain rule to obtain derivatives of F. If we write for short X_k the values of all variables after each instruction I_k, i.e. $X_0 = X$ and $X_k = f_k(X_{k-1})$, the chain rule gives the Jacobian F' of F

$$F'(X) = f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \ldots \times f'_1(X_0) \tag{2}$$

which can be mechanically translated back into a sequence of instructions I'_k, and these sequences inserted back into a copy of the control of P, yielding program P'. This can be generalized to higher level derivatives, Taylor series, etc.

3.1 Tangent and Reverse AD modes

In practice, the above Jacobian $F'(X)$ is often far too expensive to compute and store. Notice for instance that equation (2) repeatedly multiplies matrices, whose size is of the order of $m \times n$. Fortunately, resolution of many problems requires only some projections of $F'(X)$. For example, one may need only *sensitivities*, also known as *directional derivatives*, which are $F'(X) \times \dot{X}$ for a given direction $\dot{X}$ in the input space. Using equation (2), sensitivity is

$$F'(X) \times \dot{X} = f'_p(X_{p-1}) \times f'_{p-1}(X_{p-2}) \times \ldots \times f'_1(X_0) \times \dot{X}, \tag{3}$$

which is efficiently computed from right to left, because this keeps multiplying a matrix by a vector. This is easy to implement, because the derivatives of the first instructions are required first, and thus they can be interleaved with the original program instructions. This is the principle of the *tangent mode* of AD, which is the most straightforward, of course available in TAPENADE. The program differentiated in tangent mode is often called the *tangent program* of P, written $\dot{\mathtt{P}}$.

However in optimization, data assimilation [21], inverse problems, or adjoint problems [13, 6, 19], the appropriate derivative is the *gradient* of F. Notice that the gradient is only defined for a scalar-valued function: in the general case where Y is a vector, we need to define a scalar linear combination $Y^* \times \overline{Y}$ as the new result. The star in Y^* denotes transposition, which makes it a row vector. The weighting vector $\overline{Y}$ is an input to the differentiated program. The gradient of $Y^* \times \overline{Y}$ is therefore $F'^*(X) \times \overline{Y}$, where F'^* is the *transposed* Jacobian. Using equation (2), the gradient writes

$$F'^*(X) \times \overline{Y} = f'^*_1(X_0) \times f'^*_2(X_1) \times \ldots \times f'^*_{p-1}(X_{p-2}) \times f'^*_p(X_{p-1}) \times \overline{Y}, \tag{4}$$

because the transpose of $A \times B$ is $B^* \times A^*$. Again, this is most efficiently evaluated from right to left, because matrix×vector products are so much cheaper than matrix×matrix products. This is the principle of the *reverse mode* of AD, which is available in TAPENADE. The program differentiated in reverse mode is often called the *adjoint program* of P, written $\overline{\mathtt{P}}$.

This turns out to make a very efficient program, at least theoretically [15, Section 3.4]. The computation time required for the gradient is only a small multiple of the run time of P. When the number m of outputs of F is small, which is usually the case for optimization problems, the reverse mode of AD can be used to compute the complete Jacobian row by row at a very reasonable cost. In particular, this cost is independent from the number of input parameters n, which can be very large.

However, we observe in equation (4) that the X_k are required in the *inverse* of their computation order. If the original program *overwrites* a part of X_k, the differentiated program must restore X_k before it is used by $f'^*_{k+1}(X_k)$. This is the main drawback of the reverse mode of AD. There are two classical strategies to cope with that:

- **Recompute-All (RA):** the X_k are recomputed when needed, restarting P on input X_0 until instruction I_k. Figure 2 graphically illustrates this RA strategy. The $\overleftarrow{I}_k$

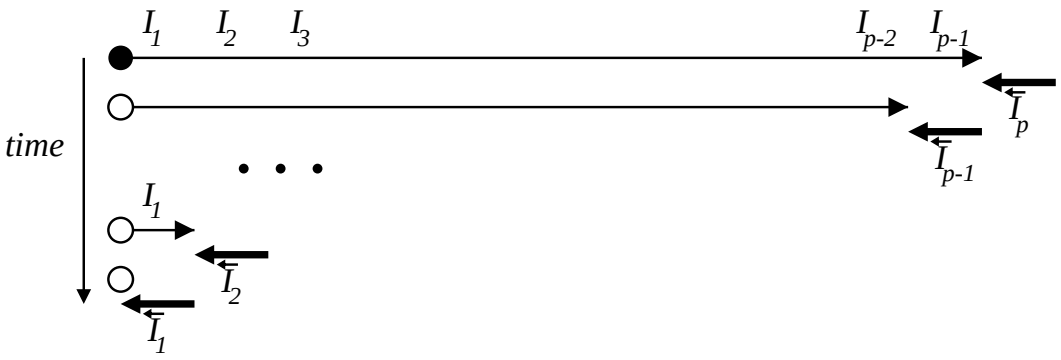


Figure 2: The Recompute-All strategy

instructions and backwards arrows represent the execution of the differentiated instructions, that effectively compute $\overline{Y}_{k-1} = f_k'^*(X_{k-1}) \times \overline{Y}_k$. The big black and white dots represent respectively the storage and retrieval of enough values to be able to restart computation from the initial point. Brute-force RA strategy has essentially a quadratic time cost with respect to the total number of run-time instructions p. The TAF [11] tool uses this strategy, together with a *slicing* strategy, called ERA [12], to limit recomputations to those actually needed to compute the required X_k.

- **Store-All (SA):** the X_k are restored from a stack when needed. This stack is filled during a preliminary run of `P` that additionally stores variables on the stack just before they are overwritten. These stored variables form what is sometimes called the *trajectory* (deprecated) or the *tape*.

  This preliminary run is called the *forward sweep* $\overrightarrow{\mathtt{P}}$. The differentiated instructions strictly speaking form the *backward sweep* $\overleftarrow{\mathtt{P}}$. Figure 3 graphically illustrates this SA strategy. The small black and white dots represent respectively the storage of

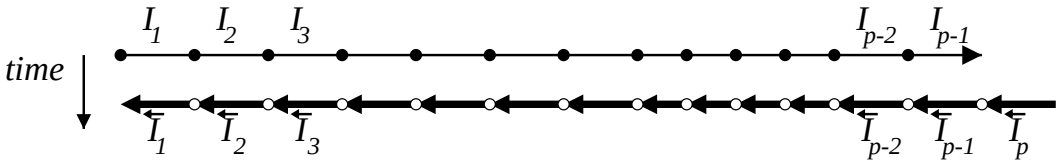


Figure 3: The Store-All strategy

intermediate values just before they are overwritten by an instruction I_k and their corresponding retrieval during the backward sweep. Brute-force SA strategy has a

linear memory cost with respect to p. The ADIFOR [4] and TAPENADE tools use this strategy.

In all cases, these back-and-forth strategies are a problem regarding vocabulary. What would mean *before* or *after* a given instruction? In the sequel, *before* and *after* refer to the global execution order of the complete differentiated program. On the other hand, we shall use *downstream* and *upstream* to refer to the order of the corresponding original instructions. In other words, in the *backward* sweep, the instruction *after* is also *upstream* (to the original program's origin), and the instruction *before* is also *downstream* (to the original program's end). All specialists of the reverse mode (e.g. salmons?) will testify how much harder it is to go upstream.

3.2 Checkpointing in the reverse mode

On real large programs, neither the RA nor the SA strategy can work. Both need a special storage/recomputation trade-off in order to be really efficient. This trade-off is called *checkpointing*. The idea is to select one or many sub-parts `S` of the program `P`, possibly nested. For each `S`, one can spare some repeated recomputation in the RA case, some memory in the SA case, at the cost of remembering a *snapshot*, i.e. enough variables to be able to restart execution from a given point. Specifically in the RA case, the snapshot is taken at the end of `S`, in order to reduce the length of recomputation sequences. Symmetrically in the SA case, the snapshot is taken at the beginning of `S` so that the initial forward sweep does no storage at all, but an extra run of `S` will be done later, this time with storage. Figure 4 illustrates checkpointing for both RA and SA strategies, for one single checkpointed sub-part (**Ckp**) as well as for nested checkpointed parts (**Ckp***).

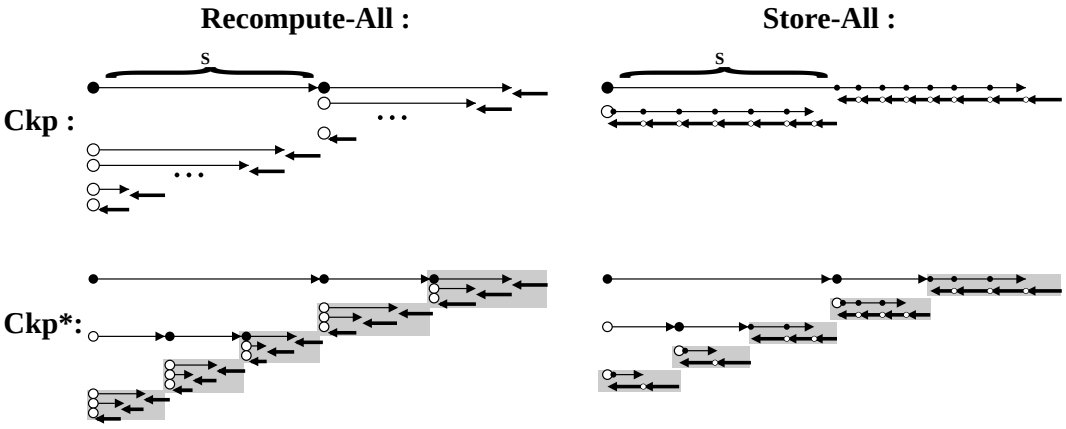


Figure 4: Checkpointing in the contexts of RA and SA

Interestingly, fig. 4 shows that when the number of checkpoints increases, the overall structure of the adjoint program becomes similar, and so do the costs in extra recomputation and snapshot storage. The difference lies in the smaller program parts (shown in grey on fig. 4) in which the pure RA or SA strategy is applied. Usually, these parts are the size of a procedure call, a loop body, or even a basic block of instructions.

TAPENADE uses the SA strategy and applies checkpointing to procedure (subroutine or function) calls. This is illustrated on figure 5, on which we define some vocabulary and

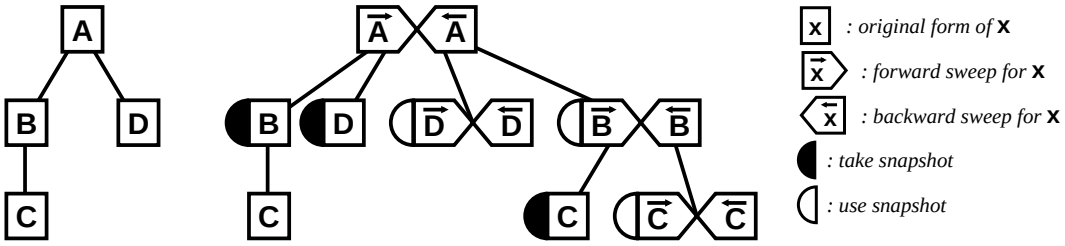


Figure 5: Checkpointing on calls in TAPENADE reverse AD

graphical notations. Execution of a procedure `A` in its original form is shown as `A`. The *forward sweep*, i.e. execution of `A` augmented with storage of variables on the stack just before they are overwritten, is shown as $\overrightarrow{\texttt{A}}$. The *backward sweep*, i.e. actual computation of the gradient of `A`, which pops values from the stack when they are needed to restore the X_k's, is shown as $\overleftarrow{\texttt{A}}$. With no checkpointing, the adjoint program is just $\overline{\texttt{A}} = \overrightarrow{\texttt{A}}; \overleftarrow{\texttt{A}}$. TAPENADE applies checkpointing to every call to a user procedure call, such as `B` in `A`. Procedure `B` will be run *without* storage during $\overrightarrow{\texttt{A}}$. When the backward sweep $\overleftarrow{\texttt{A}}$ reaches `B`, it runs $\overrightarrow{\texttt{B}}$, i.e. `B` again but this time with storage and then immediately it runs the backward sweep $\overleftarrow{\texttt{B}}$ and finally the rest of $\overleftarrow{\texttt{A}}$. Duplicate execution of `B` requires that some variables used by `B` (a *snapshot*) be stored. Figure 5 shows the resulting differentiated call graph for an example initial program call graph. If the program's call graph is actually a well balanced call tree, the memory size as well as the computation time required for the reverse differentiated program grow only like the depth of the original call tree, i.e. like the logarithm of the size of `P`, which is satisfactory.

3.3 Activity

In many real situations, the end-users of AD need only the derivatives of some selected outputs of `P` with respect to some selected inputs of `P`. Whatever the differentiation mode (tangent, reverse,...), these restrictions allow the AD tool to produce a much more efficient differentiated program. Essentially, fixing some inputs and neglecting some outputs allows AD to just forget about several intermediate differentiated variables. This has two main consequences:

- several differentiated variables just disappear from the differentiated code, because they will contain either null or useless derivatives. Memory usage of the differentiated code becomes smaller.
- several differentiated instructions are simplified or erased because one of their derivative arguments has a known trivial value. Execution time of the differentiated code becomes shorter.

Like most other AD tools, TAPENADE has a specific analysis, known as *activity analysis*, that detects these situations, therefore allowing for a better differentiated code.

TAPENADE allows the end-user to specify that only some output variables (the "*dependent*") must be differentiated with respect to only some input variables (the "*independent*"). We say that variable `y` *depends on* `x` when the derivative of `y` with respect to `x` is not trivially null. We say that a variable is "*varied*" if it depends on at least one independent. Conversely we say that a variable is "*useful*" if at least one dependent depends on it. Finally, we say that a variable is "*active*" if it is at the same time varied *and* useful. In the special case of the tangent mode, it is easy to check that when variable `v` is not varied at some place in the program, then its derivative $\dot{v}$ at this place is certainly null. Conversely when variable `v` is not useful, then whatever the value of $\dot{v}$, this value does not matter for the final result. Symmetric reasoning applies for the reverse mode of AD: observing that differentiated variables go upstream, we see that a useless variable has a null derivative, in other words the partial derivative of the output with respect to this variable is null. Conversely Conversely when variable `v` is not varied, then whatever the value of $\overline{v}$, this value does not matter for the final result. TAPENADE will simplify the differentiated program accordingly, and section 4.5 explains on examples how far this simplification can go.

Activity analysis is global (*cf* [18]), running on the complete call graph below the topmost differentiated procedure. This is why there is no "separate compilation" in AD: the whole call graph must be analyzed globally for a good activity analysis. Let's explain in more detail: activity analysis propagates the information whether each variable depends on an independent, downstream on the program. It also propagates the information whether some dependent depends on each variable, upstream on the program. Each time this propagation encounters a call to some procedure `R`, we need the precise dependence pattern of each of `R`'s outputs on each of `R`'s inputs. This *differentiable dependency matrix*, must be precomputed for each procedure in the call graph, bottom-up, therefore calling for a global analysis on the call graph, known as the *Differentiable Dependency analysis* .

When, for some reason, one (*"black-box"*) procedure in the call graph cannot be analyzed for its differentiable dependency matrix, then some default conservative dependency matrix must be used, somehow making each output depend on each input. This has dramatic consequences on activity analysis: usually, all variables downstream the call become "varied", all variables upstream become "useful", a lot of variables thus become active, and the differentiated code is less efficient. To avoid this degradation of the quality of activity analysis, the differentiable dependency matrix of each black-box procedure must be provided to the AD tool in some manner. For example TAPENADE provides an interface for the end-user to specify these differentiable dependency matrices, described in section 6.2.

4 THE DIFFERENTIATION MODEL OF TAPENADE

The previous section showed the theoretical basis of Automatic Differentiation. It gave a rough idea of what a differentiated program looks like. In this section, we plan to describe precisely the actual differentiation model of TAPENADE. The goal is to gain a deeper understanding and familiarity with programs produced by TAPENADE.

In its current version, TAPENADE can differentiate FORTRAN77 programs and FORTRAN95 programs. Thus we shall describe TAPENADE's differentiation model for these languages only. Differentiation of C is planned in the near future, and will call for a few specific additions to the model, which are not described here. In the following sections we shall address the constructs of FORTRAN95. The restricted model for FORTRAN77 can be simply derived from the general model for FORTRAN95.

4.1 Symbol names

First consider symbol names. If a variable `v` is of differentiable type (*cf* section 4.2), and currently has a non-trivial derivative (*cf activity* 3.3), this derivative is stored in a new variable that TAPENADE names after `v` as follows: `vd` ("`v` *dot*") in *tangent* mode, and `vb` ("`v` *bar*") in reverse mode. Derivative names for user-defined types, procedures and `COMMONS` are built appending "`_D`" or "`_d`" in *tangent* mode and "`_B`" or "`_b`" in reverse mode. Please bear in mind that FORTRAN is case independent, so the choice of the case is just a matter of style. TAPENADE only puts type and procedure names in upper case. Figure 6 summarizes that. Notice that TAPENADE automatically checks for possible conflicts with names already

original program	TAPENADE **tangent**	TAPENADE **reverse**

`SUBROUTINE T1(a)`
`  REAL a(10),w`
`  INTEGER jj`
`  TYPE(OBJ1) b(5)`
 	
`  COMMON /param/ jj,w`
	`SUBROUTINE T1_D(a,ad)`
	`  REAL a(10),ad(10),w,wd`
	`  INTEGER jj`
	`  TYPE(OBJ1) b(5)`
	`  TYPE(OBJ1_D) bd(5)`
	`  COMMON /param/ jj,w`
	`  COMMON /param_d/ wd`
		`SUBROUTINE T1_B(a,ab)`
		`  REAL a(10),ab(10),w,wb`
		`  INTEGER jj`
		`  TYPE(OBJ1) b(5)`
		`  TYPE(OBJ1_B) bb(5)`
		`  COMMON /param/ jj,w`
		`  COMMON /param_b/ wb`

Figure 6: Names of differentiated symbols

used in the program, in which case it appends `0`, then `1`, etc after the derivative name until conflicts disappear. The way that differentiated symbols are built can be changed via command line options.

4.2 Differentiated data types

When a variable has a non-trivial derivative, TAPENADE builds and declares a new differentiated variable that holds this derivative. The type of this derivative comes from the

type of the original variable. In simple cases, when the variable is `REAL`, `REAL*8`, `DOUBLE PRECISION`, `COMPLEX`, or any other primitive type, the differentiated variable has exactly the same type. When the original variable is an array, the differentiated variable is also an array with the same dimensions.

When the variable is a `LOGICAL`, `INTEGER`, `CHARACTER`, or any other primitive type for which no notion of derivative is defined, things are even simpler since there is no differentiated variable at all.

Things are slightly more complex for user-defined structured types. The FORTRAN95 standard made the unfortunate choice to call them *derived types*. Instead, we shall call them *structured types* to avoid confusion with differentiation itself. TAPENADE has a principle of keeping the structure of the original program as much as possible. In particular, the derivatives of variables of a structured type are themselves structured. However, their type is slightly different, because it must have room to accommodate only the derivatives of components which may have a derivative. So TAPENADE needs to introduce and declare a differentiated structured type. For each component `x` of the original structured type `T`, there will be a component `x` of the same name and type in the differentiated structured type `T_D` or `T_B` if somewhere in the program there is a variable of type `T` with a non-trivial derivative for its component `x`.

original program	TAPENADE **tangent**
...	...
	TYPE VECTOR_D
	REAL :: x,z
	END TYPE VECTOR_D
TYPE VECTOR	TYPE VECTOR
CHARACTER(64) :: name	CHARACTER(64) :: name
REAL :: x,y,z	REAL :: x,y,z
END TYPE VECTOR	END TYPE VECTOR
TYPE (VECTOR) u,v,w	TYPE (VECTOR) u,v,w
...	TYPE (VECTOR_D) ud
FUNCTION TEST(a,b)	...
REAL TEST	FUNCTION TEST_D(a,ad,b,bd,test)
TYPE (VECTOR) a,b	REAL test, TEST_D
	TYPE (VECTOR) a,b
	TYPE (VECTOR_D) ad,bd
TEST = a%x*b%x + u%z	test_d = a%x*bd%x + ad%x*b%x + ud%z
...	TEST = a%x*b%x + u%z
	...

Figure 7: Differentiated Data Types

Notice that this way it may happen that some component of the differentiated structured type is not really used for some differentiated variables. Avoiding this would require

definition of many differentiated types for some `T`, one for each combination of differentiated and non-differentiated components. This would not gain much in memory space and would cost a lot in readability and complexity. We apply here a general principle, which is to prefer *generalization* to *specialization*, especially when specialization leads to combinatorial complexity. This principle will be applied again for subroutines and functions (*cf* section 4.8).

As a last refinement, when all components of the original structured type appear in the differentiated structured type, then the differentiated type is simply replaced by the original type.

Figure 7 illustrates this on a small tangent mode example, where structured type `VECTOR` is differentiated into `VECTOR_D`, and type components `y` and `name` do not appear in `VECTOR_D` because there are never any derivatives to store there.

4.3 Simple assignments

Now consider an assignment I_k. In *tangent* mode (*cf* equation (3)), derivative instruction $\dot{I}_k$ implements $\dot{X}_k = f'_k(X_{k-1}).\dot{X}_{k-1}$, with initial $\dot{X}_0 = \dot{X}$. In *reverse* mode (*cf* equation (4)), derivative instruction(s) $\overleftarrow{I}_k$ implements $\overline{Y}_{k-1} = f'^*_k(X_{k-1}).\overline{Y}_k$, with initial $\overline{Y}_p = \overline{Y}$. Here also, TAPENADE uses the principle of keeping the original program's structure: just like the original program *overwrites* variables, the differentiated program overwrites the differentiated variables, writing values $\dot{X}_k$ over previous values $\dot{X}_{k-1}$ in tangent mode, or writing values $\overline{Y}_{k-1}$ over previous values $\overline{Y}_k$ in the reverse mode. For example, if I_k is `a(i)=x*b(j) + COS(a(i))`, it is straightforward to write the Jacobian of the corresponding function

$$f_k : \begin{array}{ccc} I\!R^3 & \rightarrow & I\!R^3 \\ \begin{pmatrix} \dot{a}(i) \\ \dot{b}(j) \\ \dot{x} \end{pmatrix} & \mapsto & \begin{pmatrix} \dot{a}(i) \\ \dot{b}(j) \\ \dot{x} \end{pmatrix} \end{array}$$

which is the matrix

$$\begin{pmatrix} -SIN(a(i)) & x & b(j) \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Recalling that transposing a matrix is just applying a symmetry with respect to the diagonal, we can write the operations that $\dot{I}_k$ and $\overleftarrow{I}_k$ must implement:

$$\dot{I}_k \quad \text{implements} \quad \begin{pmatrix} \dot{a}(i) \\ \dot{b}(j) \\ \dot{x} \end{pmatrix} = \begin{pmatrix} -SIN(a(i)) & x & b(j) \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \dot{a}(i) \\ \dot{b}(j) \\ \dot{x} \end{pmatrix},$$

$$\overleftarrow{I}_k \quad \text{implements} \quad \begin{pmatrix} \overline{a}(i) \\ \overline{b}(j) \\ \overline{x} \end{pmatrix} = \begin{pmatrix} -SIN(a(i)) & 0 & 0 \\ x & 1 & 0 \\ b(j) & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \overline{a}(i) \\ \overline{b}(j) \\ \overline{x} \end{pmatrix},$$

and therefore TAPENADE produces the derivative instructions shown on figure 8.

TAPENADE **tangent**	TAPENADE **reverse**

`ad(i) = xd*b(j)  &`
`&  + x*bd(j)  &`
`&  - ad(i)*SIN(a(i))`
	`xb = xb + b(j)*ab(i)`
	`bb(j) = bb(j) + x*ab(i)`
	`ab(i) = -SIN(a(i))*ab(i)`

Figure 8: Differentiation of a simple assignment

4.4 Array or Vectorial notation

FORTRAN95 provides *array notation*, allowing the user to specify systematic operations on sets of array elements in one single instruction. Instead of writing a loop, one can write an assignment whose operands are *array sections* instead of array elements. Each operation operates uniformly on each element of these array sections. Array instructions are useful because they can be compiled very efficiently for parallel and vectorial target architectures. TAPENADE can differentiate these array instructions, producing new array instructions whenever possible. Figure 9 illustrates this, emphasizing that two intrinsic array functions play a

original program	TAPENADE **tangent**	TAPENADE **reverse**

`a(0:n:3) = x*SUM(b(:))`	`ad(0:n:3) = xd*SUM(b(:))&`
	`&  + x*SUM(bd(:))`
		`xb = xb+SUM(b(:))*SUM(ab(0:n:3))`
		`bb(:) = bb(:)+x*SUM(ab(0:n:3))`
		`ab(0:n:3) = 0.0`

Figure 9: Differentiation of an array assignment

very special role with differentiation: the `SUM` and `SPREAD` intrinsics. The `SPREAD` intrinsic is often implicit, like here on the scalar `x`, which is implicitly spread as an array section of the appropriate dimension. The differentiated instructions come from the fact that the tangent derivatives of a `SUM` is a `SUM`, and the one of a `SPREAD` is a `SPREAD`. Conversely, the adjoint of a `SUM` is a `SPREAD` and vice-versa. This is easily checked by writing down the local Jacobian matrix and its transpose, like in section 4.3.

The other array intrinsic functions, such as `PROD`, are treated like black-box routines whose differentiation is given to TAPENADE in special library files.

4.5 Activity and reinitializations of derivatives

As introduced in section 3.3, only the derivatives of the active variables need be computed and manipulated. If variable `v` is not varied then `vd` is certainly null and the value of `vb` does not matter. Conversely, if `v` is not useful, then the value of `vd` does not matter, and `vb` is

certainly null. TAPENADE automatically detects active and inactive variables and simplifies the differentiated program accordingly. In the example of figure 10, `x` immediately becomes

original program	TAPENADE **tangent**	TAPENADE **reverse**

`x = 1.0`
`z = x*y`
`t = y**2`
`IF (t .GT. 100) ...`
	`x = 1.0`
	`zd = x*yd`
	`z = x*y`
	`t = y**2`
	`IF (t .GT. 100) ...`
		`x = 1.0`
		`z = x*y`
		`t = y**2`
		`IF (t .GT. 100) ...`
		`...`
		`yb = yb + x*zb`

Figure 10: Instructions simplifications due to Activity Analysis

not varied, and `t` is useless. Therefore, TAPENADE knows that `xd` and `tb` are null: they can be simplified and even never computed. Resetting them explicitly to zero would be just a waste of time. We shall say that these derivatives are *implicit-null*. Symmetrically, `td` and `xb` are non-null but do not matter, and therefore need not be evaluated. However, if control flow merges later downstream and the other incoming flow has an explicit non-null derivative for this variable, TAPENADE is forced to reset explicitly the corresponding implicit-null variable just before control flow merges.

This has is a somewhat puzzling consequence: some of the user-given independent and dependent variables may turn out to be inactive after activity analysis. If so, TAPENADE removes them automatically, which may be surprising. The next example on figure 11 illustrates this.

TAPENADE summarizes its decisions about activity of variables by setting comments before each differentiated procedure. Figure 11 shows these comments on a small subroutine `S1`, for which tangent and reverse differentiation were required for dependents `v1, v3, v5, v6, v7` with respect to independents `v1, v3, v4, v6, v7`: One can run activity analysis by hand on this small example. The input independent variables `v3, v6, v7` are useless, and the output dependent variables `v3, v6, v7` are not varied. This justifies the two comments where only `v1, v4`, and `v5` remain. Differentiated variables `v3d, v4d, v6d, v3b, v5b`, and `v6b` are indeed *implicit-null* at the end of the differentiated subroutine, so they are not even reset to zero because the generated calling subroutine will build shorter code that takes care of that. However in the special case where subroutine `S1` is the topmost differentiated routine, i.e. it will be used in user-written code, the differentiated routines are slightly different, and so are the comments. To avoid misuses of the `S1_D` and `S1_B` in the hand-written code, the *implicit-null* variables that are visible from the calling context are explicitly reset to zero. This is reflected by the "`output variables`" list in the tangent code, which will feature in addition `v3, v4`, and `v6`, and by the "`input variables`" list in the reverse code, which will feature in addition `v3, v5`, and `v6`. This can be surprising because the end user *did not* declare `v3` as independent nor `v4` as dependent !

Similar extra initializations are inserted before and after calls to differentiated black-box procedures, to allow for a more natural hand-coding of the derivative of the black-

original program	TAPENADE **tangent**	TAPENADE **reverse**

	`! ... forward (tangent) mode:`
	`! ... output variables: v1 v5`
	`! ... input variables: v1 v4`
		`! ... reverse (adjoint) mode:`
		`! ... input variables: v1 v4`
		`! ... output variables: v1 v5`

`subroutine S1(v1,&`
`&v2,v3,v4,v5,v6,v7)`
`real v1, v2, v3, &`
`& v4, v5, v6, v7`
`real tmp1, tmp2`
	`SUBROUTINE S1_D(v1, v1d, v2, &`
	`& v3, v4, v4d, v5, v5d, v6, v7)`
	`REAL v1, v1d, v2, v3, v4, &`
	`& v4d, v5, v5d, v6, v7`
	`REAL tmp1, tmp2`
		`SUBROUTINE S1_B(v1, v1b, v2, &`
		`& v3, v4, v4b, v5, v5b, v6, v7)`
		`REAL v1, v1b, v2, v3, v4, &`
		`& v4b, v5, v5b, v6, v7`

`tmp1 = v1`
`tmp2 = v2`
	`tmp1 = v1`
	`tmp2 = v2`
	`v1d = v2*v1d`	

`v1 = v1*v2`
`v2 = tmp1*v3*v6`
`v3 = tmp2*tmp2`
	`v1 = v1*v2`
	`v2 = tmp1*v3*v6`
	`v3 = tmp2*tmp2`
	`v5d = v4d*v4 + v4*v4d`
		`v4b = 2*v4*v5b`
		`v1b = v2*v1b`
		`END`

`v5 = v4*v4`
`v4 = 1.0`
`v6 = 2.0`
`end`
	`v5 = v4*v4`
	`v4 = 1.0`
	`v6 = 2.0`
	`END`	

Figure 11: Effect of Activity Analysis on a complete procedure

box. Similarly, this is indicated by a message (`AD09`) (*cf* section 6.3) which is issued at differentiation time, that specifies which independent and dependent are required for the black-box from its calling contexts.

4.6 Control Flow structure

Now that we have a clearer idea of how a single instruction is differentiated and why, let's focus on the flow of control. In *tangent* mode, equation (3) allows derivative instructions $\dot{I}_k$ to run along with the original I_k. In fact $\dot{I}_k$ is *just before* I_k, because I_k may overwrite a part of X_{k-1} that is used by $f'_k(X_{k-1})$ in $\dot{I}_k$. Sequences of instructions, i.e. basic blocks , obviously produce differentiated basic blocks. The control flow instructions that glue basic blocks together are essentially reproduced identically. Actually, what is reproduced identically is the structure of the Flow Graph which serves as the internal representation of procedures. As a consequence, the last regeneration step in TAPENADE, that goes from a flow graph to a syntactic tree, makes some arbitrary choices while building the final syntactic tree, most often for the better for example removing `GOTO`'s or cleaning up badly nested conditionals. In some rare situations, these choices actually make the code appearance worse, and we are progressively working to correct this. Therefore one should keep in mind that the original control flow structure is sometimes slightly altered by TAPENADE's differentiation model.

In *reverse* mode, TAPENADE applies the *Store-All* strategy (*cf* section 3), resulting in a *forward sweep* followed by a *backward sweep*. Like the tangent mode, the forward sweep of the reverse mode reproduces the control constructs of the original code. In addition, the forward sweep stores into a stack (the "tape") the intermediate variables potentially required by the derivatives, plus some flags, or *control information*, that will allow the backward sweep to implement the reverse of the original control flow between the $\overleftarrow{I}_k$. The stack is used classically through several `PUSH*` and `POP*` subroutines, according to the type of the stacked value.

Its internal representation of programs as Flow Graphs allows TAPENADE to use structured programming in the backward sweep like in the forward sweep, using very little memory space to store the control, and with no restriction on the original control constructs (all sorts of `GOTO`'s, arithmetic `IF`'s, alternate procedures or I-O returns,...). The principle is: the right time to store the control is when the original control flow *merges*, and what must be stored then is *where* the control actually *came from*. For iterative constructs, TAPENADE tries to store an iteration counter, which is a single integer, so that the backward sweep can also feature an iterative loop. For most control constructs, the backward sweep tries to use the same constructs. Thus the reverse of an `IF` is an `IF`, the reverse of a `SELECT CASE` is a `SELECT CASE`, the reverse of a `DO` loop is most often a `DO` loop, etc. In particular situations of loops with multiple entries or exits, TAPENADE must store more control information on the forward sweep, and the backward sweep may be unable to keep a clean `DO` loop structure. Figure 12 illustrates how TAPENADE builds the control structure of the differentiated procedures.

original program

```
SUBROUTINE S1(a, n, x)
...
DO i=2,n,7
  IF (a(i).GT.1.0) THEN
    a(i) = LOG(a(i)) + a(i-1)
    IF (a(i).LT.0.0) a(i)=2*a(i)
  END IF
ENDDO
END
```

TAPENADE **reverse: forward sweep**

```
DO i=2,n,7
  IF (a(i).GT.1.0) THEN
    CALL PUSHREAL4(a(i))
    a(i) = LOG(a(i)) + a(i-1)
    IF (a(i).LT.0.0) THEN
      CALL PUSHREAL4(a(i))
      a(i) = 2*a(i)
      CALL PUSHINTEGER4(3)
    ELSE
      CALL PUSHINTEGER4(2)
    END IF
  ELSE
    CALL PUSHINTEGER4(1)
  END IF
ENDDO
CALL PUSHINTEGER4(i - 7)
```

TAPENADE **tangent**

```
SUBROUTINE S1_D(a, ad, n, x)
...
DO i=2,n,7
  IF (a(i).GT.1.0) THEN
    ad(i) = ad(i)/a(i) + ad(i-1)
    a(i) = LOG(a(i)) + a(i-1)
    IF (a(i).LT.0.0) THEN
      ad(i) = 2*ad(i)
      a(i) = 2*a(i)
    END IF
  END IF
ENDDO
END
```

TAPENADE **reverse: backward sweep**

```
CALL POPINTEGER4(ad_to)
DO i=ad_to,2,-7
  CALL POPINTEGER4(branch)
  IF (branch .GE. 2) THEN
    IF (branch .GE. 3) THEN
      CALL POPREAL4(a(i))
      ab(i) = 2*ab(i)
    END IF
    CALL POPREAL4(a(i))
    ab(i-1) = ab(i-1) + ab(i)
    ab(i) = ab(i)/a(i)
  END IF
ENDDO
```

Figure 12: Differentiation of the flow of control

4.7 Input-Output (I-O) procedures

Input-Output (I-O) procedures are essentially not differentiated, because they do not participate in the numerical computation. However, TAPENADE takes care of some side-effects that affect derivatives. When a variable is overwritten by an I-O procedure (e.g a `READ`), it becomes not-varied downstream, and useless upstream. Either its derivative can become *implicit null*, or TAPENADE automatically forces reinitialization of its derivative to zero. However, the end-user should check that this is the behavior wanted, and therefore TAPENADE sends a warning message (`AD05`) at differentiation time. Please refer to section 6.3 for a complete description of TAPENADE's messages.

There is a subtler problem related to I-O: in some programs, I-O files are used as temporary storage. If the stored variables indeed should have a derivative, TAPENADE is unable to detect this. Should it detect this, TAPENADE would have to create a new file, as well as the I-O instructions that write and read the derivatives to and from this file. TAPENADE does not do that. The least TAPENADE *can* do, though, is to warn the user when this might happen. Thus each time a varied variable is written into a file which is not the standard output, a message (`AD03`) is sent. Also each time a useful variable is read from a file which is not the standard input, another message (`AD04`) is sent. The end-user is responsible for checking that everything is OK.

Figure 13 illustrates this in the tangent mode on some representative I-O calls: The first

original program	TAPENADE **tangent**

	`DO ii1=34,66,2`
	`  bd(ii1) = 0.0`
	`ENDDO`

`read*,b(34:66:2)`	`READ*, b(34:66:2)`
`...`	`...`
`write(22,*), (a(i),i=1,100,3)`	`WRITE(22, *), (a(i), i=1,100,3)`
`...`	`...`

	`DO i=0,33`
	`  td(i) = 0.0`
	`ENDDO`

`read(22,*), (t(i),i=0,33)`	`READ(22, *), (t(i), i=0,33)`
`...`	`...`
	`bd(1:33) = t(1:33)*ad(34:66)`
`b(1:33) = a(34:66)*t(1:33)`	`b(1:33) = a(34:66)*t(1:33)`
`File:  (AD05) Active variable b[34:66:2] overwritten by I-O`	
`File:  (AD03) Active variable (a[i] ,i=1,100,3) written by I-O to file 22`	
`File:  (AD04) Useful variable (t[i] ,i=0,33) read by I-O from file 22`	

Figure 13: Differentiation of I-O procedure calls, showing warning messages

message (`AD05`) refers to the first `READ`: if the user wants to study derivatives with respect

to the values read into `b`, then the program should be modified to take this `READ` out of the differentiated section. Otherwise one need not worry. The next pair of messages (`AD03`) and (`AD04`) point out the possible communication of an active value through file 22. If the user knows this is the case, this is not treated by TAPENADE, and therefore the result is probably incorrect. For example, we see that variable `ad` was not written along with `a`, so `t` has not received a derivative and the differentiation of the `t*a` product is incorrect.

4.8 Procedure calls

TAPENADE treats procedure calls differently from simple instructions, because a procedure call indeed represents a bunch of instructions, possibly with control. Therefore the differentiated instructions cannot be put *before* the original call, but rather *inside*, yielding a differentiated procedure, with additional arguments for the derivatives of original arguments which are active, either just before or just after the call.

In *tangent mode*, a call to `SUB` just becomes a call to the differentiated `SUB_D`. Notice that if `SUB` is a function that returns an active value, TAPENADE makes `SUB_D` a function too, that returns the derivative of the value, and it also "returns" the original value, but this time as an extra output argument named `SUB`. In *reverse mode*, TAPENADE *checkpoints* the procedure call: the forward sweep calls the original `SUB` and the backward sweep calls the differentiated `SUB_B`, that gathers its own forward and backward sweeps (*cf* figure 5). Notice that even if `SUB` is a function that returns an active value, TAPENADE always makes `SUB_B` a subroutine, because the result of `SUB` is not an output of `SUB_B` (*cf* section 5.5), and the derivative of the result of `SUB` (`subb`) is an additional *input* argument of `SUB_B`. Figure 14 illustrates this when `SUB` is a subroutine.

We recall that TAPENADE prefers procedure *generalization*, as opposed to *specialization*. Thus, even if a procedure is called many times, with arguments sometimes active, sometimes not, only one differentiated procedure is built, i.e. for the most general activity of arguments. Thus, specific calls are sometimes given dummy derivatives, either to feed them with a null derivative input, or to receive a derivative result which will not be used. Assume `SUB` is called elsewhere with an active 3^{rd} argument, whereas the 4^{th} argument is never active. This explains the "`0.0`" argument in tangent, and the "`arg3b`" in reverse, which is not used later.

In the reverse mode, checkpointing requires taking a *snapshot*. TAPENADE runs a classical *Read-Written analysis* , plus specific *Adjoint Liveness* and *Adjoint Written* analyses to find a minimal snapshot, made of variables that are both *used* by the differentiated procedure and *overwritten* before the differentiated procedure is called. On the example of figure 14, the combination of the above analyses could prove that this is only the case for `x`.

In some rare situations a variable must be stored as part of the snapshot, and also as part of the tape, i.e. because it is needed by derivatives of previous instructions (*cf* section 5.1). Instead of `PUSH`'ing the value twice, TAPENADE uses a special `LOOK` function, that reads the value from the stack without removing it, so that it can be `POP`'ed again later.

original program	TAPENADE **reverse: forward sweep**

`x = x**3`
 	
`CALL SUB(a, x, 1.5, z)`
 	
`x = FUN(x*y)`
	`CALL PUSHREAL4(x)`
	`x = x**3`
	`CALL PUSHREAL4(x)`
	`CALL SUB(a, x, 1.5, z)`
	`arg = x*y`
	`CALL PUSHREAL4(x)`
	`CALL PUSHREAL4(arg)`
	`x = FUN(arg)`

TAPENADE **tangent**	TAPENADE **reverse: backward sweep**

`xd = 3*x**2*xd`
`x = x**3`
`CALL SUB_D(a, ad, x, xd,`
`  1.5, 0.0, z)`
`argd = xd*y + x*yd`
`arg = x*y`
`xd = FUN_D(arg, argd, x)`
	`CALL POPREAL4(arg)`
	`CALL FUN_B(arg, argb, xb)`
	`CALL POPREAL4(x)`
	`xb = y*argb`
	`yb = yb + x*argb`
	`CALL POPREAL4(x)`
	`CALL SUB_B(a, ab, x, xb, 1.5, arg3b, z)`
	`CALL POPREAL4(x)`
	`xb = 3*x**2*xb`

Figure 14: Differentiation of procedure calls

4.9 Overloading

Overloading allows the user to call different procedures with the same name. At compile-time (like in FORTRAN95) or at run-time (like in object-oriented languages), the system decides which is the actual procedure called, generally by comparing the data types of the procedure's arguments. Furthermore, FORTRAN95 allows the user to overload predefined operators such as `+`, `-`, `*`, `/` by user functions, and to overload the assignment `=` by a user subroutine.

During its type-checking phase, TAPENADE decides which procedure or predefined operator calls are overloaded, and replaces them by calls to the appropriate procedures. Then differentiation can proceed like for any other procedure call. Figure 15 shows an example of that, where `*` and `=` are overloaded as `TM` and `TS` respectively when their arguments are of the user-defined structured type `T`. Similarly, function `F` is overloaded either as `TF` or `RF` according to the type of the first argument. Overloading is often combined with modules, so this example anticipates on section 4.10 for this aspect. One can verify on the tangent and reverse differentiated results that the overloaded procedures are detected, replaced, and finally differentiated following the standard differentiation model of TAPENADE (*cf* section 4.8). Notice also that the overloaded form is preserved in the differentiated procedure whenever possible. However, the calls to `TF_D` and `TF_B` could also be replaced by calls to more general overloaded interface functions `F_D` and `F_B`, following the pattern of interface function `F`. This will probably be done in future versions of TAPENADE

4.10 Overall Modular Structure

FORTRAN95 files have an overall structure of nested modules and internal/external procedures, with interfaces. The structure of FORTRAN77 is much flatter in comparison. TAPENADE produces a differentiated code that reproduces this modular structure.

An important question is whether the differentiated modules should contain (1) the differentiated procedures only, or (2) also the original procedures. The answer is (2) because both the original procedures and the differentiated procedures must sometimes operate on global module variables, which may be private. Therefore both procedures must be inside the same differentiated module to gain access to this global variable. The example on figure 16 illustrates this differentiation of modular structured programs, in tangent mode, on an example module that contains one type definition, variables, and one function. Figure 15 also illustrates differentiation of modules.

This relates to the more general question of whether the differentiated object can exist independently of the original object, or must they be attached inside the same enclosing level. For example a differentiated instruction must be in the same procedure as the original instruction because they share a common control. Similarly a differentiated procedure must be in the same module as the original because both may access a private object of this module. On the contrary, a differentiated field `x` of a structured type `T` need not be added into `T`, but can rather go into a stand-alone "differentiated" structured type `T_D` or `T_B`, therefore saving memory space for the objects of type `T` which are not active. Similarly a

original program

```
MODULE MOD_T
 TYPE T
  real :: x,y
  integer :: i
 END TYPE T
 INTERFACE OPERATOR(*)
  MODULE PROCEDURE TM
 END INTERFACE
 INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE TS
 END INTERFACE
CONTAINS
 SUBROUTINE TS(a,b)
  ...
 FUNCTION TM(a, b)
  ...
END MODULE MOD_T

FUNCTION TF(v,u)
 ...
FUNCTION RF(v,u)
 ...

SUBROUTINE HEAD(a,b,c,r)
 use MOD_T
 TYPE(T) :: a,b,c
 REAL r
 INTERFACE F
  FUNCTION TF(v,u)
   use MOD_T
   TYPE(T) :: v
   real u,TF
  END FUNCTION TF
  FUNCTION RF(v,u)
   real u,RF,v
  END FUNCTION RF
 END INTERFACE
 ...
 r = b*c
 r = F(a,2.0) * r
 ...
END
```

TAPENADE **tangent**

```
MODULE MOD_T_D
 TYPE T
   REAL :: x,y
   INTEGER :: i
 END TYPE T
 TYPE T_D
   REAL :: x,y
 END TYPE T_D
 ...


CONTAINS
 SUBROUTINE TS_D(a,ad,b,bd)
  ...
 FUNCTION TM_D(a,ad,b,bd,tm)
 ...
END MODULE MOD_T_D

FUNCTION TF_D(v, vd, u, tf)
...
FUNCTION RF_D(v, vd, u, rf)
...

SUBROUTINE HEAD_D(a, ad, b,&
&          bd, c, cd, r, rd)
 USE MOD_T_D
 TYPE(T) :: a, b, c, arg1
 TYPE(T_D) :: ad, bd,      &
&             cd, arg1d
 REAL :: r, rd, result1,   &
&             result1d
 ...



 arg1d=TM_D(b,bd,c,cd,arg1)
 CALL TS_D(r,rd,arg1,arg1d)
 result1d = TF_D(a, ad,    &
&                2.0,result1)
 rd = result1d*r+result1*rd
 r = result1*r
 ...
END SUBROUTINE HEAD_D
```

TAPENADE **reverse**

```
MODULE MOD_T_B
 TYPE T
   REAL :: x,y
   INTEGER :: i
 END TYPE T
 TYPE T_B
   REAL :: x,y
 END TYPE T_B
 ...


CONTAINS
 SUBROUTINE TS_B(a,ab,b,bb)
  ...
 SUBROUTINE TM_B(a,ab,b,bb,tmb)
 ...
END MODULE MOD_T_B

SUBROUTINE TF_B(v, vb, u, tfb)
...
SUBROUTINE RF_B(v, vb, u, rfb)
...

SUBROUTINE HEAD_B(a, ab, b, &
&           bb, c, cb, r, rb)
 USE MOD_T_B
 TYPE(T) :: a, b, c, arg1
 TYPE(T_B) :: ab, bb,      &
&             cb, arg1b
 REAL :: r, rb, result1,   &
&             result1b
 ...
 arg1 = b * c
 r = arg1
 result1 = F(a, 2.0)
 CALL PUSHREAL4(r)
 r = result1*r
 ...
 CALL POPREAL4(r)
 result1b = r*rb
 rb = result1*rb
 CALL TF_B(a,ab,2.0,result1b)
 CALL TS_B(r,rb,arg1,arg1b)
 CALL TM_B(b,bb,c,cb,arg1b)
 ...
END SUBROUTINE HEAD_B
```

Figure 15: Differentiation model in presence of Overloading

original program	TAPENADE **tangent**

`module example1`
` type vector`
`  real :: x,y,z`
` end type vector`
` type(vector) :: u,v,w`
`contains`
` function dot_prod(a,b)`
 	
`  type(vector) :: a,b`
 	
`  real :: dot_prod`
 	
`  dot_prod = a%x*b%x + &`
`&  a%y*b%y + a%z*b%z`
` end function dot_prod`
 	
`end module`
	`MODULE EXAMPLE1_D`
	`  TYPE VECTOR`
	`  REAL :: x,y,z`
	`  END TYPE VECTOR`
	`  TYPE(VECTOR) :: u, v, w`
	`CONTAINS`
	`  FUNCTION DOT_PROD_D(a, ad, &`
	`&  b, bd, dot_prod)`
	`  TYPE(VECTOR) :: a, b`
	`  TYPE(VECTOR) :: ad, bd`
	`  REAL :: dot_prod`
	`  REAL :: dot_prod_d`
	`  dot_prod_d = ad%x*b%x + &`
	`&  a%x*bd%x + ad%y*b%y + &`
	`&  a%y*bd%y + ad%z*b%z + &`
	`&  a%z*bd%z`
	`  dot_prod = a%x*b%x + &`
	`&  a%y*b%y + a%z*b%z`
	`  END FUNCTION DOT_PROD_D`
	
	`  FUNCTION DOT_PROD(a, b)`
	`  IMPLICIT NONE`
	`  TYPE(VECTOR) :: a, b`
	`  REAL :: dot_prod`
	`  dot_prod = a%x*b%x + &`
	`&  a%y*b%y + a%z*b%z`
	`  END FUNCTION DOT_PROD`
	`END MODULE EXAMPLE1_D`

Figure 16: Differentiation of Modular Programs

differentiated top-level subroutine may exist in a separate file from the original subroutine, avoiding code duplication for further maintenance.

4.11 Multi-directional differentiation

Until here, the tangent mode of AD takes as input a single vector $\dot{X}$ which is the direction in the input space, along which the directional derivatives must be computed. Conversely, the reverse mode of AD takes as input a single weight vector $\overline{Y}$ that defines the composite optimization criterion for which the gradient must be computed.

Actually, nothing forbids the tangent mode to run simultaneously on several values of $\dot{X}$, nor *respectively* the reverse mode to run on several values of $\overline{Y}$. This is called *multi-directional* tangent or reverse mode. This is functionally equivalent to running the tangent code (*respectively* the adjoint code) many times, with the same value of X but with different values of $\dot{X}$ (*respectively* $\overline{Y}$). The difference lies in the execution time, because multi-directional code computes the original function only once.

One classical use of multi-directional mode is to compute the complete Jacobian matrix, either column by column through multi-directional tangent mode, or row by row through multi-directional reverse mode. If the number of rows (i.e. output variables) is notably smaller than the number of columns (i.e. input variables), then the multi-directional reverse mode is recommended. Otherwise the multi-directional tangent mode should be preferred.

The multi-directional extension can be applied to both tangent and reverse modes of AD. However for the time being, TAPENADE only provides the multi-directional tangent mode. Next versions of TAPENADE will probably feature the multi-directional reverse mode as well.

The multi-directional differentiation model is based on spreading each memory cell that holds a single derivative: instead of holding just a scalar, it now holds an array whose dimension is the maximum number of differentiation "directions" $\dot{X}_v$ or $\overline{Y}_v$. Figure 17 shows multi-directional tangent differentiation of function `FF`, yielding subroutine `FF_DV`. Several things must be noted:

- Differentiated variables have an extra dimension with respect to their original variable. In the case of arrays, this new dimension is put first, so that when one element of the array is passed to a procedure (e.g. `y(i)` in the call to `F2_DV`), the array of all its derivatives may be passed along (e.g. `yd(1, i)`).

- Since FORTRAN does not allow a function to return an array, differentiated functions with an active result must become subroutines. This is what happens here to function `FF`, which becomes a subroutine `FF_DV` with output parameters `ff` and `ffd`.

- A simple instruction such as an assignment produces now a differentiated instruction which is a loop that iterates for each differentiation direction. To minimize loop overhead, TAPENADE performs a data dependence analysis to reorder the differentiated instructions. The goal is to create sequences of differentiated loops, as long as possible, and then to fuse these loops because they have the same iteration space and

original program	TAPENADE **multi-directional tangent**

function FF(x,y)
real x(100),y(100),v,ff
ff = 1.0
Do i = 1, 100
  v = (x(i) + y(i))/2.0
  ff = ff + x(i)*y(i)
  x(i) = v*ff
  v = v*x(i)
  call F2(v,y(i))
enddo
end
	SUBROUTINE FF_DV(x, xd, y, yd, ff, ffd, nbdirs)
	INCLUDE 'DIFFSIZES.inc'
	! Hint: nbdirsmax should be the maximum number
	! of differentiation directions
	INTEGER nbdirs, i, nd
	REAL ff, ffd(nbdirsmax), x(100), xd(nbdirsmax,100), &
	& y(100), yd(nbdirsmax,100), v, vd(nbdirsmax)
	DO nd=1,nbdirs
	  ffd(nd) = 0.0
	ENDDO
	ff = 1.0
	DO i=1,100
	  v = (x(i)+y(i))/2.0
	  DO nd=1,nbdirs
	  ffd(nd) = ffd(nd) + xd(nd, i)*y(i) + x(i)*yd(nd, i)
	  ENDDO
	  ff = ff + x(i)*y(i)
	  x(i) = v*ff
	  DO nd=1,nbdirs
	  vd(nd) = (xd(nd, i)+yd(nd, i))/2.0
	  xd(nd, i) = vd(nd)*ff + v*ffd(nd)
	  vd(nd) = vd(nd)*x(i) + v*xd(nd, i)
	  ENDDO
	  v = v*x(i)
	  CALL F2_DV(v, vd, y(i), yd(1, i), nbdirs)
	ENDDO
	END

Figure 17: Multi-Directional tangent differentiation

independent iterations. On the example, TAPENADE can create a cluster of three differentiated instructions. Data dependences on variable `x(i)` prevent merging of the fourth differentiated loop. Notice that the differentiated loops are essentially parallel, and telling this to the compiler can bring an interesting speedup.

- As usual, differentiated assignments are separated from their original assignment, whereas procedure calls become one single call to the differentiated procedure.
- There is a distinction between the two new integer variables `nbdirsmax` and `nbdirs`. Variable `nbdirsmax` is actually a constant which defines the size of the extra dimension in derivatives. Therefore it is the maximum number of simultaneous differentiation directions that can be handled by the differentiated program. It must be defined as a constant, with its static value (e.g. 25), in the include file named `DIFFSIZES.inc`, for example as:

```
INTEGER nbdirsmax
PARAMETER (nbdirsmax=25)
```

  There is a comment in the differentiated program and also a warning message during differentiation (`AD10`) (*cf* section 6.3) to remind the user of this. On the other hand, `nbdirs` is the actual number of differentiation directions for one particular run. It is a run-time parameter passed to each differentiated subroutine, used to avoid computing derivatives along undefined directions. Of course `nbdirs` must be smaller or equal to `nbdirsmax`.

5 SPECIFIC REFINEMENTS FOR THE ADJOINT

The above section already made it clear that the reverse mode is far more complex than the tangent mode. This is the price for efficient gradient computation. This section presents specific aspects of the model of reverse AD. This is necessary to fully understand some particular structures in adjoint codes, that would appear very strange otherwise. Most of these refinements can be disactivated via command-line options, for example for training purposes, and also for debugging.

5.1 *To Be Restored* analysis

We saw that intermediate values need to be "taped" before overwritten, but this is *only* when they will be used by the differentiated instructions. A specific program static analysis, called *To Be Restored* (TBR) [9, 23] does this in TAPENADE. In the example of figure 18, TAPENADE could prove that neither `x` nor `y` were needed by the differentiated instructions, and therefore did not `PUSH` them on nor `POP` them from the stack.

original program	**reverse mode: naive backward sweep**	**reverse mode: backward sweep with TBR**

`x = x + EXP(a)`
`y = x + a**2`
`a = 3*z`
	`CALL POPREAL4(a)`
	`zb = zb + 3*ab`
	`ab = 0.0`
	`CALL POPREAL4(y)`
	`ab = ab + 2*a*yb`
	`xb = xb + yb`
	`yb = 0.0`
	`CALL POPREAL4(x)`
	`ab = ab + EXP(a)*xb`
		`CALL POPREAL4(a)`
		`zb = zb + 3*ab`
		`ab = 0.0`
		`ab = ab + 2*a*yb`
		`xb = xb + yb`
		`yb = 0.0`
		`ab = ab + EXP(a)*xb`

Figure 18: Removing unnecessary storage through *To Be Restored* analysis

5.2 Gathering incrementation instructions

Many reverse differentiated instructions *increment* a differentiated variable. Others just reset them, often to zero. These instructions may fall far apart in the differentiated program, which uses the reversed original instruction order, and therefore it is likely that the compiler will not optimize them. TAPENADE implements a *data dependency* analysis that allows it to safely move and gather initializations and incrementations of the same differentiated variable, whenever possible. Also, when a differentiated variable is *implicit-null*, an incrementation of this variable becomes a simple assignment. The result is a shorter code, called the *non-incremental code*, which is closer to what one would write when programming an *adjoint*

code by hand, and which improves data locality. Figure 19 illustrates the effect of this refinement on the same example as in the previous section.

original program	**reverse mode: backward sweep with TBR**	TAPENADE **reverse: non-incremental backward sweep**

`x = x + EXP(a)`
`y = x + a**2`
`a = 3*z`
	`CALL POPREAL4(a)`
	`zb = zb + 3*ab`
	`ab = 0.0`
	`ab = ab + 2*a*yb`
	`xb = xb + yb`
	`yb = 0.0`
	`ab = ab + EXP(a)*xb`
		`CALL POPREAL4(a)`
		`zb = zb + 3*ab`
		`xb = xb + yb`
		`ab = 2*a*yb + EXP(a)*xb`
		`yb = 0.0`

Figure 19: Gathering instructions and improving locality through instructions reordering

5.3 Detection of Aliasing

Program transformation tools, and AD tools in particular, assume that two different variables represent different memory locations. The program can specify explicitly that two different variables indeed go to the same place, using pointers or the `EQUIVALENCE` declaration. In this case the tool must cope with that. But it is not recommended (and forbidden by the standard) that the program hides this information, e.g declaring a procedure with two formal arguments and giving them the same variable as an actual argument. This is called *aliasing*. TAPENADE detects this situation and issues a warning message (`DF02`) (*cf* section 6.3) to the user. This message should not be overlooked, because it may point to a future problem in the differentiated code, especially in the reverse mode.

Figure 20 shows another form of aliasing, local to an instruction, where an assigned variable may or may not be the same as a read variable. In this situation, it is impossible to write a single *reverse* differentiated instruction, because the differentiated code strongly depends on the fact that the assigned variable is also read or not. TAPENADE detects this situation and automatically inserts a temporary variable (e.g. `tmp`), therefore removing local aliasing through instruction splitting. For example on figure 20, there is a local aliasing in the third instruction, because equality between `i` and `n-i` could not be decided.

5.4 Splitting complex expressions

The derivative of complex expressions often turns out to be even more complex and longer! Even if the original expression contains no duplication, naive differentiation introduces duplicate sub-expressions. TAPENADE provides an automatic splitting of expressions that virtually eliminates all duplication coming from differentiation. For the moment, only the reverse

original program	TAPENADE **reverse: forward sweep**	TAPENADE **reverse: backward sweep**

`a(i) = 3*a(i) + a(i+1)`
 	
`a(i+2) = 2*a(i)`
 	
`a(n-i) = a(i)*a(n-i)`
	`CALL PUSHREAL4(a(i))`
	`a(i) = 3*a(i) + a(i+1)`
	`CALL PUSHREAL4(a(i+2))`
	`a(i+2) = 2*a(i)`
	`tmp = a(i)*a(n-i)`
	`CALL PUSHREAL4(a(n-i))`
	`a(n-i) = tmp`
		`CALL POPREAL4(a(n-i))`
		`tmpb = ab(n-i)`
		`ab(i) = ab(i) + a(n-i)*tmpb`
		`ab(n-i) = a(i)*tmpb`
		`CALL POPREAL4(a(i+2))`
		`ab(i) = ab(i) + 2*ab(i+2)`
		`ab(i+2) = 0.0`
		`CALL POPREAL4(a(i))`
		`ab(i+1) = ab(i+1) + ab(i)`
		`ab(i) = 3*ab(i)`

Figure 20: Detection and correction of aliasing in the reverse mode

mode uses this mechanism, but the tangent mode will use it in further versions. Expressions are not split during the forward sweep, to avoid intermediate variables that might need to be added to the tape. Splitting occurs only in the backward sweep, and occurs only at carefully selected places in the differentiated expressions. Figure 21 illustrates this on a "not so long" expression, to keep things readable. On this example, splitting spares one exponentiation and one division.

original program	**reverse mode: backward sweep**	TAPENADE **reverse: split backward sweep**

`r1 =  a*SIN(b)`
`    - x**y/a`
	`ab = ab + SIN(b)*r1b`
	`    + x**y*r1b/(a*a)`
	`bb = bb + a*COS(b)*r1b`
	`xb = xb`
	`    - y*x**(y-1)*r1b/a`
	`yb = yb`
	`    - x**y*LOG(x)*r1b/a`
	`r1b = 0.0`
		`tempb = -(r1b/a)`
		`temp = x**y`
		`ab = ab + SIN(b)*r1b`
		`    - temp*tempb/a`
		`bb = bb + a*COS(b)*r1b`
		`xb = xb + y*x**(y-1)*tempb`
		`yb = yb + temp*LOG(x)*tempb`
		`r1b = 0.0`

Figure 21: Factorizing duplicate computations through minimal splitting

5.5 Adjoint Liveness analysis

Reverse differentiation of the program P that computes function F yields program $\overline{\text{P}}$ that computes the gradient of F. The original results of P, which are also computed by the forward sweep of $\overline{\text{P}}$, are *not* a result of $\overline{\text{P}}$. Only the gradient is needed by the end-user. Moreover in

original program	**reverse mode:**	TAPENADE **reverse: dead adjoint code removed**

`IF (a.GT.0.0) THEN`
`  a = LOG(a)`
`ELSE`
`  a = LOG(c)`
`  CALL SUB(a)`
`ENDIF`
`END`
	`IF (a .GT. 0.0) THEN`
	`  CALL PUSHREAL4(a)`
	`  a = LOG(a)`
	`  CALL POPREAL4(a)`
	`  ab = ab/a`
	`ELSE`
	`  a = LOG(c)`
	`  CALL PUSHREAL4(a)`
	`  CALL SUB(a)`
	`  CALL POPREAL4(a)`
	`  CALL SUB_B(a, ab)`
	`  cb = cb + ab/c`
	`  ab = 0.0`
	`END IF`
		`IF (a .GT. 0.0) THEN`
		
		`  ab = ab/a`
		`ELSE`
		`  a = LOG(c)`
		
		`  CALL SUB_B(a, ab)`
		`  cb = cb + ab/c`
		`  ab = 0.0`
		`END IF`

Figure 22: Removing useless dead adjoint code

most implementations the original results will be overwritten and lost during the backward sweep of $\overline{\mathtt{P}}$. Therefore some of the last instructions of the forward sweep of $\overline{\mathtt{P}}$ are actually dead code. TAPENADE implements a slicing analysis, called *Adjoint Liveness analysis*, to remove this dead adjoint code. Incidentally, as a side-effect of this slicing analysis, TAPENADE is also able to detect which arguments of a procedure are effectively read and effectively overwritten, not by the procedure `P` itself, but by its adjoint $\overline{\mathtt{P}}$. This is the *Adjoint Written* analysis, already mentioned in section 4.8, which is used to reduce the snapshot taken before calling $\overrightarrow{\mathtt{P}}$. The example on figure 22 shows the effect of detection of Adjoint Liveness analysis on a small program which terminates on a test, with some dead adjoint code at the end of each branch.

6 USING TAPENADE FROM THE COMMAND LINE

TAPENADE can be installed on the local computer and run from the command line or from a `Makefile`, just like a compiler. A typical call to TAPENADE is:

```
#> tapenade -reverse -root func -vars "x z" file1.f file2.f
```

TAPENADE may be downloaded from the FTP address:

`ftp://ftp-sop.inria.fr/tropics/tapenade/`

All TAPENADE documentation, with a tutorial and an ever-growing reference manual, is available at:

`http://www-sop.inria.fr/tropics/tapenade.html`

Section 6.1 describes the command line options, section 6.2 describes the files where the user can give information on black-box procedures, section 6.3 explains the messages that TAPENADE sends to the user during differentiation, and section 6.4 describes how the user can provide the variable size information requested by TAPENADE.

6.1 Available TAPENADE options

TAPENADE offers two main ways to drive differentiation: command-line parameters and configuration files. Directives put into the source files are a third way which is not available yet, and which will be available in the next versions.

Schematically, the command line has the following simple syntax:

`tapenade` *<options_list> <file_list>*

- The *<file_list>* is an unordered list of source files that contain the program to be differentiated. File names must be separated by white space. The `INCLUDE` files must not be given in the *<file_list>*: TAPENADE will look for them automatically when meeting an `INCLUDE` instruction in some source file. `INCLUDE` files can recursively contain `INCLUDE` files. By default, the suffix of each source file indicates its language, `.f` `.for` or `.fortran` for FORTRAN77, `.f90` or `.f95` for FORTRAN95, and `.c` for C.
- The *<options_list>* is an unordered list of options, possibly empty, separated by white space. Options belong to several categories, described in the next sections

6.1.1 Fundamental options

These options give the basic data on which F to differentiate, which dependent Y with respect to which independent X, and with which AD mode.

`-root <name>` `-head <name>`	Specifies the name of the "root" procedure that defines the function F to differentiate. TAPENADE only allows you to differentiate complete procedures. If you want F to be a part of a procedure, you must first split this part as a new procedure. If this option is not used, the default root procedure is chosen in the maximal elements of the call graph, i.e. any procedure which is not called by another procedure.
`-outvars <vars>`	Specifies the dependents, i.e. the list of the output variables of the root procedure for which the derivative is requested. Variables which are not output (local variables or constants) are just ignored. `<vars>` is a list of variable names, between double quotes (`"`). The double quotes may be omitted when there is only one variable in the list. If this option is not used, the default is the list of all output variables of the root procedure.
`-vars <vars>`	Specifies the independents, i.e. the list of the input variables of the root procedure with respect to which the derivative is requested. Variables which are not input (local variables or constants or function return) are just ignored. `<vars>` is a list of variable names, between double quotes (`"`). The double quotes may be omitted when there is only one variable in the list. If this option is not used, the default is the list of all input variables of the root procedure.
`-d` `-tangent`	Calls for differentiation in the tangent mode. The short name `-d` is a reminder for "dot" or the deprecated name "direct". This option is the default if neither `-d`, `-b`, nor `-p` option is used.
`-b` `-reverse`	Calls for differentiation in the reverse mode. The short name `-b` is a reminder for "bar" or the deprecated name "backward".
`-multi`	Modifies the current differentiation mode to turn it into multi-directional (*cf* section 4.11). In the current version of TAPENADE, this option applies only to the tangent mode.
`-p` `-preprocess`	Does no differentiation at all, and therefore ignores the `-outvars` and `-vars` options. This mode produces a program which is equivalent to the input program, but analyzed and restructured by TAPENADE. This option is also useful for debugging purposes.

6.1.2 File information options

These options control the style and location of several files that TAPENADE will look for.

`-inputlanguage <lang>`	Specifies the language used by all the files in the *<file_ list>*. `<lang>` may be `fortran`, `fortran90`, `fortran95`, or `c`. This option overrides the default mechanism that deduces the language of each file from its suffix.
`-outputlanguage <lang>`	Specifies the language in which the output differentiated files must be produced. `<lang>` may be `fortran`, `fortran90`, `fortran95`, or `c`. This option overrides the default mechanism that produces each differentiated procedure in the same language as its original procedure.
`-o <file>` `-output <file>`	Specifies that the complete differentiated program must be written into a single file, named `<file>_<mode>.<ext>`. Extension `<ext>` reflects the language. This option overrides the default mechanism that writes each differentiated top-level object (program, subroutine, function, or module) into a separate file named after the name of this object.
`-O <dir>` `-outputdirectory <dir>`	Specifies that all produced files must be put into directory <dir>. If this option is not used, all files are created in the current directory.
`-ext <file>`	Adds `<file>` to the list of files that contain general specifications of *black-box* procedures. Initially, this list contains one predefined file, named `F77GeneralLib`, which is found in the `lib` directory of the TAPENADE installation.
`-nolib`	Empties the list of files searched for general specifications of *black-box* procedures. This even removes the initial `F77GeneralLib` file from the list.
`-extAD <file>`	Adds `<file>` to the list of files that contain AD-related specifications of *black-box* procedures. Initially, this list contains one predefined file, named `F77ADLib`, which is found in the `lib` directory of the TAPENADE installation.
`-noADlib`	Empties the list of files searched for AD-related specifications of *black-box* procedures. This even removes the initial `F77ADLib` file from the list.

6.1.3 Differentiation options

These options affect the differentiation process itself.

`-diffvarname <str>`	Specifies how the name of a differentiated variable should be built from the name of the original variable. If this option is not used, the default appends `d` in tangent mode and `b` in reverse mode.
`-difffuncname <str>`	Specifies how the name of a differentiated procedure, module, `COMMON` name, or type name should be built from the name of the original object. If this option is not used, the default appends `_D` in tangent mode, `_DV` in multi-directional tangent mode, and `_B` in reverse mode.
`-nooptim <optim_name>`	Specifies that optimization `<optim_name>` must be disactivated for this differentiation process. This might be useful for instance for teaching or debugging. The optimization `<optim_name>` can be one of the following: • `spareinit` suppresses unnecessary computation or (re)initialization of derivatives when they are *implicit-null* or useless (*cf* section 4.5). • `diffarguments` refuses to insert a differentiated argument after a procedure argument which is never active, neither before nor after any call (*cf* section 4.8). • `splitdiff` finds a minimal split of expressions, so as to reduce common subexpressions in the derivatives (*cf* section 5.4). • `mergediff` merges differentiated instructions to reduce overhead and improve data locality. (*cf* section 4.11 for multi-directional mode and section 5.2 for the reverse mode). • `tbr` uses *To Be Restored* analysis to reduce memory consumption in the reverse mode (*cf* section 5.1). • `adjointliveness` removes instructions that are dead in the adjoint code (*cf* section 5.5). • `deadcontrol` removes control which is dead in the adjoint code. • `snapshots` reduces the snapshot taken before each procedure call in the reverse mode, using *adjoint Read-Written analysis* (*cf* sections 4.8 and 5.5).

6.1.4 System options

These options specify system information. Sizes of primitive types may also influence differentiation, because they influence the way `EQUIVALENCE`s are solved, as well as the memory size passed to `PUSH*` and `POP*` subroutines in the reverse mode.

`-java_home <dir>`	Specifies the directory `<dir>` where the JAVA interpreter is. If this option is not used, the default is `/usr/local/jdk1.4.1`
`-javaheapsize <size>`	Specifies the maximum java heap size for this command. If this option is not used, the default is `-mx256m`, but a larger heap may be necessary for very large programs.
`-i<n>`	Specifies the size in bytes of integer numbers on the current target architecture. `<n>` may be 2, 4, or 8. If this option is not used, the default is 4.
`-r<n>`	Specifies the size in bytes of real numbers on the current target architecture. `<n>` may be 4 or 8. If this option is not used, the default is 4.
`-dr<n>`	Specifies the size in bytes of double precision real numbers on the current target architecture. `<n>` may be 4, 8, or 16. If this option is not used, the default is twice the size of real numbers..

6.1.5 Display and Debugging options

These options control display of differentiation output, including some outputs that are used for debugging purpose.

`-msglevel <n>`	Sets the maximum detail level of the messages that show progress of differentiation. If this option is not used, the default is 5. Larger `<n>` prints more.
`-html`	Displays the differentiation result into a web page, similar to the output of the TAPENADE web server described in section 7. If your web browser doesn't display it automatically, the page to load is in `./tapenadedir/tapenade.html`, or in `<dir>/tapenadedir/tapenade.html` if the option `-O <dir>` is used.
`-view`	Opens windows that display graphically the internal representation of the original program and of the differentiated program, which are each a Call Graph of Flow Graphs.
`-dump <file>`	Writes internal representation of the program, differentiated program, and analyses results into a (large) dump file named `<file>`.

6.2 Giving information on black-box procedures

In addition to command-line options, the user can (actually *should*) give information on each black-box procedure used by the program.

We call black-box any procedure whose source is not given to TAPENADE, either because it comes from a compiled library, or because its source language is not understood by TAPENADE, or even because the user prefers to differentiate the procedure by hand and therefore does not want TAPENADE to differentiate it.

Black-box procedures cannot be analyzed normally by TAPENADE, for example for type-checking, read-written, or activity analysis 3.3. If nothing is known on the black-box procedure, TAPENADE makes conservative assumptions. For example that each output may depend on each input. This has dramatic consequences: nearly all variables downstream become "varied", nearly all variables upstream become "useful", many variables become active and the differentiated code is still correct but less efficient.

TAPENADE allows the user to give precise information on black-box procedures in special configuration files, one for the AD-independent information (e.g. `MyGeneralLib`) and one for the AD-related information (e.g. `MyADLib`).

Each entry in `MyGeneralLib` is about one procedure, and gives info about number, type, and intent of arguments. For example you may type for a `"BBOX"` subroutine

```
subroutine bbox:
  external:
  shape:(param 1,
         param 2,
         common /c1/[0,*[,
         common /c2/[0,16[,
         common /c2/[16,176[,
         param 3,
         param 4,
         param 5)
  type :(ident real,
         ident real,
         arrayType(ident real,dimColons(dimColon(none(),none()))),
         modifiedTypeName(modifiers(ident double), ident real),
         arrayType(modifiedTypeName(modifiers(ident double), ident real),
                   dimColons(dimColon(none(),none()))),
         ident boolean,
         arrayType(ident character,dimColons(dimColon(none(),none()))),
         ident character)
  R:    (1, 1, 0, 1, 1, 1, 1, 1)
  W:    (0, 1, 1, 1, 1, 1, 0, 0)
```

Similarly for a function called `"FBBOX"`, you may type:

```
function fbbox:
  intrinsic:
  shape: (param 1, param 2, result)
  type: (ident real, ident real, modifiedTypeName(modifiers(ident double), ident real))
  R: (1,1,0)
  W: (0,0,1)
```

The `external:` line states that the procedure is external. For an intrinsic, just replace this line by `intrinsic:`.
The `shape:` part is necessary, because it defines the (arbitrary) order in which arguments will be referred to in the sequel. Its syntax is either `param N` that designates the N-th formal parameter, or `result` for the returned value if BBOX is a function, or `common /xx/[0,8[` for an argument which is found from offset 0 to offset 8 in common `/xx/`. All following lines are optional. If they are missing, the obvious conservative assumptions are made.
The `type:` part gives the types of each argument, in the same order. The syntax of a type is rather heavy. It should be cleaned up in next versions. For example, `real*8` is written `modifiedTypeName(modifiers(intCst 8), ident real)` Arrays are built with the combinator `arrayType`, that binds the array elements' type with a list of dimensions (`dimColon`) which are pairs of the lower and upper bound. So far, these bounds are not very useful, so you might just as well leave them as `none()`. Take a look at the default `F77GeneralLib` to find more examples.
The `R:` part gives the read signature of each argument. A 1 means the argument may be used, at least partly, inside the procedure (not necessarily in differentiable expressions).
The `W:` part gives the written signature of each argument. A 1 means the argument may be overwritten, at least partly, inside the procedure.

Each entry in `MyADLib` is about one procedure, and gives info related to AD. This can be the differentiable dependency matrix, or also the partial derivatives for an intrinsic function. These properties are declared using the conventions defined in the corresponding `*GeneralLib` file, about the rank assigned to each argument. For example, consider the following entry about subroutine BBOX:

```
subroutine bbox:
  deps: (id,
         1, 1, 0, 0, 0, 0, 0, 0,
         1, 0, 0, 1, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0,
         0, 1, 0, 1, 1, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0,
         id,
         id)
```

The `deps:` entry gives the differentiable dependency matrix, of each output value with respect to input values, for each argument. For example here, the result in parameter 2 depends on the input parameters 1 and 2, the result in the first variable in common `/c2/` doesn't depend on any input, and the result in the tail array of `/c2/` depends on the inputs

in parameter 2 and in the whole of `/c2/`. An `id` entry instead of a complete line states that the argument is left unmodified, which is a stronger information than a line with just one 1, because it implies that the partial derivative is certainly `1.0`. This is the case here for parameters 1, 4, and 5. On the other hand, parameter 3 is overwritten, but of course depends on nobody, since it has a non-differentiable type. For the same reason, nobody depends on parameters 3, 4, and 5. Notice also that nobody depends on the input in `/c1/`: it is an output only.

File `MyADLib` can also specify the partial derivatives of black-box functions such as `FBBOX`. This uses the `derivative:` keyword. Here again, the syntax is ugly and will be cleaned up soon. Please refer to the TAPENADE web site for information on the current syntax, and on the cleaner syntax when it is available.

Let us underline an interesting utilization of the black-box mechanism: to declare hidden side-effects between communicating procedures. Suppose that two procedures `SEND` and `RECEIVE` are used, e.g. in the context of a MPI-like parallel program. These two black-box procedures communicate through a side-effect which must be declared somehow to TAPENADE. This can be achieved by introducing a dummy global communication channel (here a dummy `COMMON`), and using it in the black-box signatures of `SEND` and `RECEIVE`. This way, sending an active variable will make the received variable active too. To this end, one may insert into the `MyGeneralLib` file:

```
subroutine SEND
  shape:(param 1, common/CHANNEL/[0,*[
  type :(ident real, ident real)
  R    :(1,1)
  W    :(0,1)

subroutine RECEIVE
  shape:(param 1, common/CHANNEL/[0,*[
  type :(ident real, ident real)
  R    :(0,1)
  W    :(1,1)
```

and into the `MyADLib` file:

```
subroutine SEND
  deps:(id,
        1,1)

subroutine RECEIVE
  deps:(0,1
        0,1)
```

The end-user is responsible for providing the appropriate differentiated procedures for `SEND` and `RECEIVE`.

6.3 Messages issued by TAPENADE

TAPENADE issues a number of messages resulting from the preliminary analyses done before actual differentiation. Although the temptation is strong, these messages should not be ignored right away. Especially when AD is concerned, these messages can be the indication that the program runs into one limitation of the AD technology. Generally speaking, compilers often permit to go against the standard with no visible harm, but this often introduces errors into the program differentiated in reverse mode. So even if the original program compiles and runs correctly with your compiler, these messages warn you that differentiation may produce a faulty program.

Some messages are requests for help: when TAPENADE needs some help from the user, e.g. because it needs and does not know the size of an array, it issues a message that asks you to give this size after differentiation is done. Otherwise the resulting program will not run.

In the following sections, we present all the messages that TAPENADE may issue when differentiating a file. These messages are gathered into a number of general categories. For each category, after a general comment, you will find a **"Why should you care?"** paragraph that emphasizes the nasty things that may happen if this message is disregarded, and a **"What can you do?"** paragraph that gives hints about how the problem may be solved.

In the text of each message, symbols in italics are place holders for symbols from the program being analyzed. For example *V*, *V1*, *V2* represent variable names, *T*, *T1*, *T2* represent type names, *C*, *C1*, *C2* represent names of `COMMON`s, *F*, *F1*, *F2* represent procedure names, *K1*, *K2* represent procedure kinds (`EXTERNAL`, `INTRINSIC`, or `USER`). *Exp* represents an expression, *File* a file, *I* an interface name, *L* a label identifier, *N* a number, and *S* is any symbol.

6.3.1 Declarations problems

DD01	Cannot combine successive declarations of variable *V*: *T1* and *T2* (ignored new)
DD02	Cannot combine successive declarations of character array: *T1* and *T2* (ignored new)
DD04	Double definition of procedure *F* (ignored new)
DD05	Cannot combine successive declarations of procedure *F* (ignored new)
DD06	Cannot combine successive declarations of function *F* return type: *T1* and *T2* (overwritten previous)
DD07	Cannot combine successive declarations of procedure *F* kind: *K1* and *K2* (ignored new)
DD08	Double declaration of *K* procedure *F*
DD10	Double declaration of the main program: *F1* and *F2* (overwritten previous)
DD11	Double declaration of procedure *F* in library file *File*
DD12	Cannot combine successive declarations of type *T* (ignored new)
DD13	Double definition of label *L*
DD14	Undefined constant value *Exp*
DD15	Module *M* is not defined
DD16	Double declaration of module *M* (ignored new)
DD17	Symbol *S* is not defined in module *M*
DD18	Incorrect public/private declaration
DD20	Cannot combine successive declarations of interface *I* (ignored new)

In general these messages indicate that there are two successive definitions or partial declarations of some symbol, as a variable, constant, type, label or procedure or module name. These successive declarations occur in the same scope and do not combine into a coherent declaration for the symbol. TAPENADE therefore chooses either to ignore the new piece of declaration, or to overwrite the previous piece of declaration, as indicated by the message. For (DD08) the two choices are equivalent.

Why should you care? TAPENADE chooses to ignore the indicated declaration or part of your original program. Therefore the types considered by TAPENADE may not be the ones you expect, or the program actually differentiated may differ from the program you think. In particular when types are concerned, remember that differentiation occurs only on `REAL` (or `COMPLEX`) typed variables. Therefore some variable may have no derivative whereas you think it should have one. Also for (DD10) or (DD13), the flow of control may be different from what you think, or from what your local compiler used to build. Also for (DD11), some information that you give about an external procedure may be ignored and replaced by another one.

What can you do? Remove the declaration or definition in excess, or modify conflicting declarations so that they combine well into the type that you expect. For (DD11), clean up the library file by merging the data about a given procedure into one single entry. Use the `-nolib` and `-noADlib` options if you need to disactivate loading of `F77GeneralLib` and `F77ADLib` respectively.

6.3.2 Type problems

TC01	Expression is expected to be numeric, and is here *T*
TC02	Expression is expected to be boolean, and is here *T*
TC03	Expression is expected to be a record, and is here *T*
TC04	Expression is expected to be a pointer, and is here *T*
TC05	Expression *Exp* is expected to be an array, and is here *T*
TC06	Loop index is expected to be integer, and is here *T*
TC07	Loop bound is expected to be integer, and is here *T*
TC08	Loop times expression is expected to be integer, and is here *T*
TC09	Real part of complex constructor is expected to be numeric, and is here *T*
TC10	Imaginary part of complex constructor is expected to be numeric, and is here *T*
TC11	Argument of arithmetic operator is expected to be numeric, and is here *T*
TC12	Array index is expected to be numeric, and is here *T*
TC13	Triplet element is expected to be numeric, and is here *T*
TC14	Substring index is expected to be numeric, and is here *T*
TC15	Argument of logical operator is expected to be boolean, and is here *T*

The indicated expression is used in a context which requires some type. Using the available declarations, the type-checker found that this expression actually has a different type, indicated as *T* in the message.

Why should you care? This may be the sign of an error in the program. Even if this original program actually compiles and runs well, remember that differentiation occurs only on `REAL` (or `COMPLEX`) typed variables. If some sub-expression becomes of another type, there will be no derivative associated, even if you think there should be one. Remember also that these problems make the program non-portable.

What can you do? Check the types. You may also use the appropriate conversion functions. Be aware that when you convert a `REAL` into an `INTEGER`, the differentials are reset to zero.

TC16	Type mismatch in assignment: *T1* receives *T2*
TC17	Type mismatch in case: *T1* compared to *T2*
TC18	Type mismatch in comparison: *T1* compared to *T2*

The two terms in the assignment or comparison, or the switch and case expressions, must have compatible types. Otherwise, the meaning of the instruction is not well defined, and not portable.

Why should you care? This can be the indication for a standard type error and, like above, this can lose derivatives.

What can you do? Check the types. Use conversion functions when appropriate.

TC19	Equality test on reals is not reliable

Equality and non-equality are not well defined between REALs, because they are defined only up to a given "machine precision". Equality test can yield different results on different machines.

Why should you care? As far as AD is concerned, equality tests on REALs are sometimes used as the stopping criterion for some iterative process. Then this relates to a well-known problem: do the derivatives converge when the function does, and if yes, do they converge at the same speed?

What can you do? Check that this does not impact differentiation. Take care to perform the usual validation tests on the computed derivatives.

TC20	Variable *V* is not declared
TC21	No implicit rule available to complete the declaration of variable *V*
TC22	Variable *V*, declared implicitly as *T1*, is used as *T2*
TC23	Symbol *S*, formerly used as a variable, now used for another object
TC24	Wrong number of dimensions for array *V*: *N2* expected, and *N1* given
TC25	Dimensions mismatch in array expression, *T1* combined with *T2*
TC26	Incorrect equivalence
TC27	End of common */C/* reached before variable *V*.
TC28	Variable *V* cannot be added to common */C1/* because it is already in common */C2/*
TC29	Common */C/* declared with two different sizes: *N1* vs *N2*

These messages indicate conflicts between declaration and usage of some variable. Message (TC25) occurs in the context of FORTRAN95 array expressions. Messages (TC26) to (TC29) indicate a problem in EQUIVALENCEs and COMMONs. Recall that variables in a COMMON are stored contiguously, so that their relative order cannot be changed, and extra variables cannot be inserted occasionally. Also a variable cannot be in two different COMMONs, and one cannot set an EQUIVALENCE between two variables that already have their own, different, memory space allocated.

Why should you care? In general, if you let the system choose the type of a variable, you run the risk that it becomes REAL while you don't want, or the other way round. And this in turn impacts differentiation. Messages (TC24), (TC26), (TC27), and (TC29) are even more serious: They can indicate that the code relies on FORTRAN weak typing to perform hidden EQUIVALENCEs between two variables, or to resize, reshape, or split an array into pieces. Since TAPENADE cannot possibly understand what is intended, it might not propagate differentiability and derivatives from the old array shape to the new array shape.

What can you do? Declare variables explicitly. Try to avoid the "implicit" declaration facility, for example by setting explicitly an IMPLICIT NONE declaration. Declare arrays with their actual size (not "1"). Never resize, reshape, or split arrays implicitly through parameter passing, and avoid to do so across COMMONs

TC30	Type mismatch in argument *N* of procedure *F*, was *T1*, is here *T2*
TC31	Type mismatch in result of function *F*, was *T1*, is here *T2*
TC32	Conflicting numbers of arguments for procedure *F*, was *N1*, is here *N2*
TC33	*K* procedure *F* is not declared
TC35	Subroutine *F* used as a function
TC36	*T* function *F* used as a subroutine
TC37	Return type of function *F* set by implicit rule to *T*
TC38	Undefined overloaded operator *T1 Op T2*
TC39	Overloaded operator *Op* undefined for types *T1 T2*

These messages indicate conflicts between declaration and usage of some procedure. For most of these messages, programs which raise them should not compile. Messages (TC30) to (TC32) indicate a mismatch between the formal and actual parameters of a procedure, or between two declarations. Recall that the standards require that formal and actual parameters match in number and type.

Why should you care? In general, type mismatches mean that some variables may not get the type you intended. and this in turn impacts differentiation. Messages (TC30) to (TC32) may indicate that the code relies on FORTRAN weak typing to perform hidden EQUIVALENCEs between two variables, or to resize, reshape, or split an array into pieces. Since TAPENADE cannot possibly understand what is intended, it might not propagate differentiability and derivatives from the old array shape to the new array shape.

What can you do? Insert the correct declarations and make sure each procedure usage matches its definition.

TC42	Recursivity: *F1–>F2–>...–>Fn*

Recursivity means that a procedure *F1* calls a procedure *F2*, which itself calls a procedure *F3*, and so on, and this eventually leads to calling *F1* again. In other words this is a cycle in the call graph, Basically, this is not an error, although FORTRAN77 used to forbid recursive programs.

Why should you care? It is just a matter of time. Recursive programs are harder to analyze. We are progressively updating TAPENADE analyses so that they cope with recursivity.

What can you do? In the meantime, check that the resulting derivatives are correct, using the classical tests.

TC43	Could not find type of argument *N* of procedure *F*, please check
TC44	Could not find type of result of function *F*, please check

During differentiation, TAPENADE had to split an expression near a procedure call, either precomputing some argument of the procedure, or the function's result, into a temporary variable. Sometimes TAPENADE cannot guess the type of this temporary variable. This is probably a temporary bug of TAPENADE.

Why should you care? The temporary variable may be of a wrong type and this may loose activity.

What can you do? Insert the temporary variable by hand into the source program, and declare it of the correct type.

TC45	No field named S in T
TC46	Illegal recursive type definition

These are illegal uses of types. (TC45) says the program looks for a field name which does not exist, and (TC46) that some type recursively contains a field of the same type, thus yielding an infinite size!

Why should you care? It is surprising if the original program actually compiles! Differentiation is undefined.

What can you do? Rewrite the definitions of these types.

TC47	Definition of type T comes a little too late

The definition of type T came after it was used. This is just a warning, because TAPENADE will do as if the definition was done in due time. But you should better fix this.

TC50	Label L is not defined
TC51	Label variable V must be written by an ASSIGN-TO construct
TC52	Variable V used in assigned-GOTO is not a label variable

These are illegal uses of labels, or label variables.

Why should you care? If labels are misunderstood, the final flow of control that TAPENADE understands and regenerates might not be the one you expect.

What can you do? Define the labels you use. Use the correct constructs to manipulate label variables.

6.3.3 Control flow problems

CF01	Irreducible entry into loop
CF02	Computed-GOTO without legal destinations list
CF03	This assigned-GOTO to V goes nowhere
CF04	This assigned-GOTO to V only goes to label L
CF05	This assigned-GOTO to V is possibly undefined

These messages indicate problems in the flow of control of the program. Message (CF01) comes from jumps that go from outside to inside a loop, without going through the loop control header. The meaning of the program is not well defined and is probably implementation dependent.

Why should you care? Of course a bad control of a program yields a bad control of its differentiated programs.

What can you do? Avoid irreducible loops. Avoid jumps into loops and into branches of a conditional structure. Try to use structured programming instead. If an assigned-`GOTO` goes to only one place, at least replace it by a plain `GOTO`.

6.3.4 Data flow problems

`DF01`	Illegal non-reference value for output argument *N* of procedure *F*
`DF02`	Potential aliasing in calling function *F*, between arguments *Args*
`DF03`	Variable *V* is used before initialized
`DF04`	Variable *V* may be used before initialized

These problems are detected through the built-in inter-procedural data-flow analysis of TAPENADE.

Why should you care? Programs with (`DF01`) usually provoke segmentation faults. Aliasing (`DF02`) is a major source of errors for Differentiation in the reverse mode, as illustrated in section 5.3. Uninitialized variables lead to uninitialized derivatives.

What can you do? Output formal arguments of a procedure must be passed reference (i.e. writable) actual arguments. It is better to avoid aliasing. When aliasing is really a problem, one can easily avoid it by introducing temporary variables, so that memory locations of the parameters do not overlap. Variables should be initialized before they are used.

6.3.5 Automatic Differentiation problems

`AD01`	Actual argument *N* of *F* is active while formal argument is non-differentiable
`AD02`	Actual output *N* of *F* is useful while formal result is non-differentiable

These two messages are usually associated. They indicate the following situation: Some programs implicitly convert `REAL`s to `INTEGER`s during a function call, then these `INTEGER`s follow their way through the code, and finally are converted back into `REAL`s during another function call. This is a sort of side-effect communication of the `REAL` value

Why should you care? If the original `REAL`s are active, i.e. depend on the independent input variables, and the final `REAL`s are useful, i.e. influence the dependent output variables, then this temporary conversion into `INTEGER`s has lost activity and derivatives. Differentiability is lost, derivatives are wrong!

What can you do? `REAL`s must remain `REAL`s. If absolutely necessary, there is a "secret" option in TAPENADE to actually differentiate these strange `REAL-INTEGER`s... but we won't write it in this manual!

`AD03`	Active variable *V* written by I-O to file *File*
`AD04`	Useful variable *V* read by I-O from file *File*

This is comparable to the above (`AD01`) and (`AD02`). These two messages are usually associated. They indicate the following side-effect situation: Some programs write interme-

diate `REAL` values into a file, and later on read back these `REAL` values from this file. This process looses differentiability (*cf* section 4.7).

Why should you care? If the written value is active, i.e. depends on the independent input variables, and the corresponding (equal) value read is useful, i.e. influences the dependent output variables, then this temporary storage into a file has lost activity and derivatives. Differentiability is lost, derivatives are wrong!

What can you do? Do not use files as temporary storage. Use arrays instead. You may also use TAPENADE black-box mechanism to disconnect differentiation on the calling procedures, and differentiate them by hand, with an ad-hoc mechanism to propagate derivatives.

`AD05`	Active variable *V* overwritten by I-O

This reminds you that a variable that was active has been overwritten by an I-O operation (*cf* section 4.7).

Why should you care? The derivative is of course reset to zero. Maybe this is not what you expected?

What can you do? Check that it is OK. Otherwise try to put all I-O operations outside the program part that you are actually differentiating.

`AD06`	Differentiate of function *F* needs to save undeclared side-effect variables: *V**
`AD07`	This call needs to save undeclared side-effect variable: *V*

Sometimes, the checkpointing mechanism of the reverse mode requests to save some variables, which are declared in some called procedure, but not in the current calling procedure. Think for example of a `SAVE` variable, or a `COMMON` which is not declared at the calling procedure level.

Why should you care? The instruction that saves this variable cannot be put here, since this variable is not declared here in the calling procedure. The differentiated program is therefore wrong.

What can you do? Insert the `COMMON` declaration at the level of the calling procedure. Do something similar for `SAVE` variables.

`AD08`	Don't know the size of dimension *N* of array *V*, which must be saved

In some cases, for example in reverse-mode AD, the differentiated function must save some variables, to restore them later. If one such variable is an array, and the size of this array is dynamic, TAPENADE must know this size to actually do the save.

Why should you care? If you don't give this size, the differentiated program will not compile.

What can you do? Define the value of the constants (`PARAMETER`s) that TAPENADE requests, in the special include file named `DIFFSIZES.inc`. You may also explicitly give this size in the original file, so that the message does not show up!

AD09	Please provide a differential of function F for arguments *Signature*

You must provide the differentiated program with a new function, differentiated from F with respect to the input and output parameters of F specified in the *Signature* part. Probably F is a black-box function, for which TAPENADE has no source, and therefore it couldn't differentiate it.

Why should you care? If you don't give this new function, the differentiated program will probably not compile.

What can you do? Define this new function. You have the choice of the method. Maybe you know explicitly the derivative mathematical function, which can be implemented efficiently. Maybe you have the source and you may use AD again, with TAPENADE or with another tool. Maybe you just want to run a local "divided differences" method, knowing that this might degrade the precision of the derivatives. In any case, make sure that you provide a differentiated procedure in the correct mode (tangent, reverse...), and which adheres to the conventions of TAPENADE about the name of the procedure and the order of its parameters (*cf* section 4.8).

AD10	User help requested: constant V must hold the maximum number of differentiation directions
AD11	User help requested: constant *V1* must hold the size of dimension N of V (should be *Hint*)
AD12	User help requested: unknown dimension N of array V

During differentiation, TAPENADE must declare new arrays whose size can be a dynamic number, unknown during differentiation. Such declarations are impossible in FORTRAN77. Therefore TAPENADE declares these arrays with a size which is a constant, and asks you to define this constant before you compile the differentiated program. Similarly, `(AD10)` requires you to give the size of the new dimension that holds the array of derivatives in multi-directional differentiation (*cf* section 4.11).

Why should you care? The differentiated program will simply not compile otherwise!

What can you do? In this case, TAPENADE automatically inserts an `INCLUDE` command in the differentiated files, and the include file is named `DIFFSIZES.inc`. Create this file and fill it with all the required definitions of constants. TAPENADE gives you hints about the values you should put. Be careful to define these sizes as constants (`PARAMETER`s) and not as variables (*cf* section 4.11). Again, FORTRAN77 does not allow you to declare a local array with a size which is not known at compile time.

AD13	Differentiation root procedure F has no active input nor output

After activity analysis, it may turn out that the root differentiation procedure has no dependent which actually depends on an independent (*cf* section 4.5). Then after simplification, no active input nor output remain, and there is nothing left to differentiate.

Why should you care? The differentiated program is just empty!

What can you do? Check the dependents and independents that you specified, so that they effectively depend on one another.

6.4 Specifying required size information

Messages (AD10), (AD11), and (AD12) require that the end user defines the value of some constants in a special include file named DIFFSIZES.inc. This is necessary to compile the differentiated program, which uses these constants. There are two sorts of such constants:

- the constant that defines the maximum number of simultaneous differentiation directions in multi-directional mode (*cf* section 4.11). This constant is named usually nbdirsmax, unless the name is already used in the program. Of course the user is free to choose the value.
- the constants that give the size of some dimensions of some new arrays introduced by differentiation. These constants are usually named SIZE*n*OF*v*IN*f*, where n is the rank of the dimension whose size is missing, v is the name of the differentiated array for which the size is missing, and f is the name of the procedure in which the problem occurs. TAPENADE usually prints a hint about the actual value that should be given to these constants.

If you don't know exactly the values of these constants, you should *over-approximate* them! To define these constants in the DIFFSIZES.inc, we recommend that you use the syntax of the following FORTRAN77 example, or its FORTRAN95 equivalent:

```
INTEGER nbdirsmax
PARAMETER (nbdirsmax=25)
INTEGER SIZE2OFarraybINmyfunc
PARAMETER (SIZE2OFarraybINmyfunc=150)
```

6.5 Compiling the differentiated code

The differentiated code produced by TAPENADE consists of ordinary FORTRAN source. In most cases, a FORTRAN77 compiler is appropriate to compile it. However, when the original files use some specific FORTRAN95 constructs, or when the user required it using the -outputlanguage command-line option (*cf* section 6.1.2), the output code uses the FORTRAN95 syntax and must be compiled with a FORTRAN95 compiler.

When the differentiated code required the user to provide some array dimension sizes, it contains the line

```
INCLUDE 'DIFFSIZES.inc'
```

In this case you must create this file and put it in the directory where compilation is made.

To compile files differentiated in the reverse mode, you need to compile and link the definition of the PUSH* and POP* subroutines. They are provided in the standard TAPENADE distribution, or can be downloaded from the TAPENADE web server. They consist of a FORTRAN77 file adBuffer.f and a C file adStack.c. Both files must be compiled and linked to get the final executable.

7 USING TAPENADE FROM THE WEB INTERFACE

As an alternative to a local installation, one can also use the TAPENADE web server. It requires no installation and of course always runs the latest version of TAPENADE. It can be triggered in a few clicks from most web browsers.

7.1 The TAPENADE input form web page

To begin an AD session, the URL to get started is

`http://tapenade.inria.fr:8080/tapenade/index.jsp`

which takes you to the TAPENADE input page or form, shown on figure 23. Notice that this interface gives the user access only to the fundamental options of the command-line TAPENADE. Specifically, the web interface lets the user specify the source and include files, their language, the root procedure, the dependents, the independents, and finally the differentiation mode. On the other hand, library files about black-box procedures, output language, suffixes of differentiated variables, and optimization levels cannot be specified and therefore retain their default values.

Filling the TAPENADE request form consists of six steps, visible on figure 23.

1. A radio button selects the input language then,
2. the next step allows the user to upload all necessary source files, *this time also uploading the include files*. Notice that for security reasons, each file must be uploaded separately, which may unfortunately make the process tedious. Source files and include files must be uploaded in two separate boxes.
3. The third step is to fill in the name of the root procedure,
4. The fourth step is to fill in the names of all dependent output variables separated by white space
5. The fifth step is to fill in the names of all independent input variables separated by white space
6. The sixth and last step selects the differentiation mode and triggers differentiation.

Steps 3, 4, and 5 can be skipped, in which case the usual default answers are taken, respectively the topmost procedure in the call graph, the list of all outputs of this procedure with a differentiable type, and the list of all inputs of this procedure with a differentiable type.

During the uploading step 2, keep in mind that all files travel on the Internet to our server in Sophia-Antipolis. This connection is not encrypted nor protected. We will keep your files confidential, but we cannot guarantee they will not be spied on the web. If security of your files matters, download TAPENADE on your local system.

On the input page (figure 23), we encourage the user to register to the *tapenade-users* mailing list, to be kept informed about improvements and new developments. This is not compulsory to use the server, but it helps us to know our users better. There is also a link to the on-line documentation of the tool.

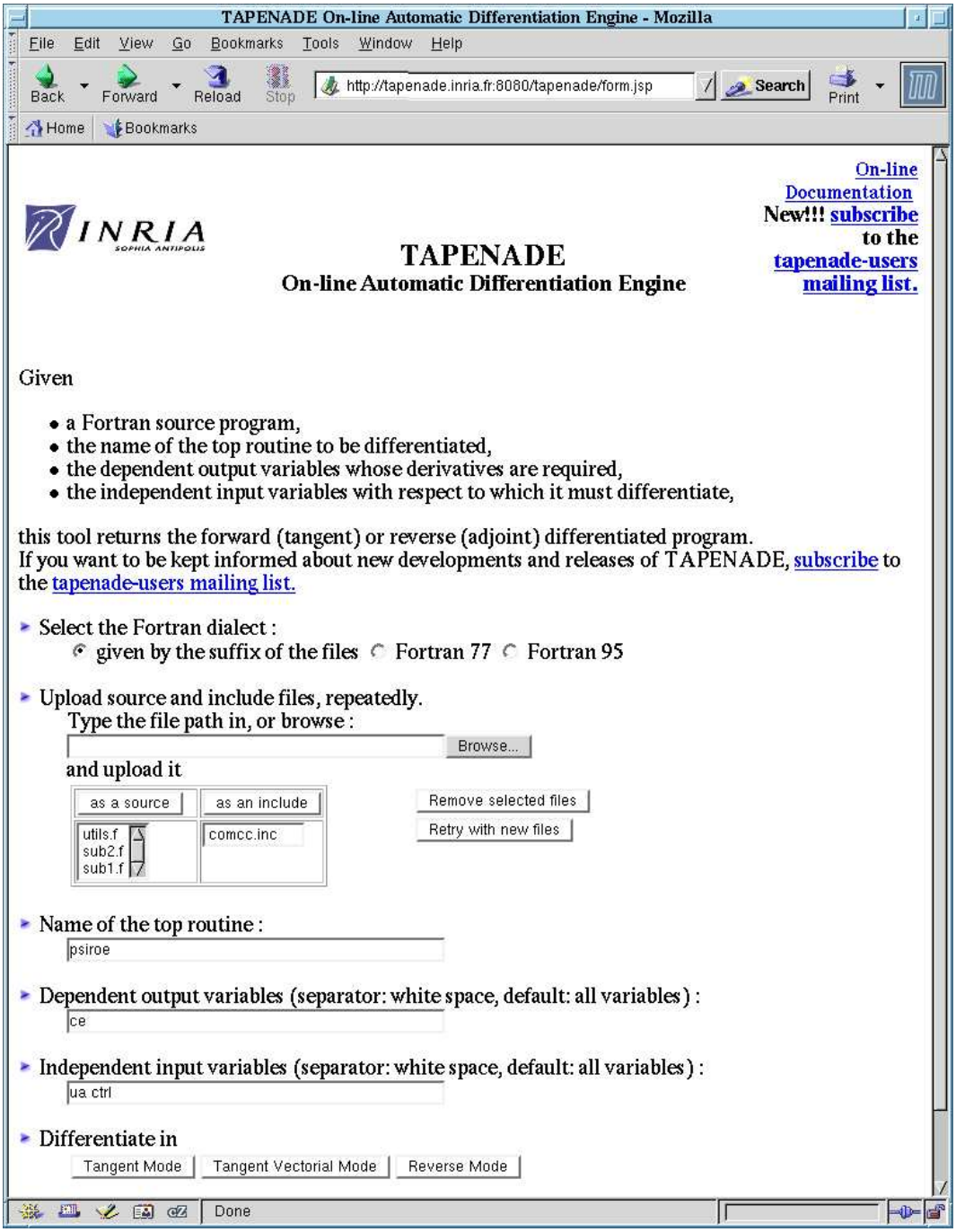


Figure 23: HTML interface for TAPENADE input

7.2 The TAPENADE output result web page

A moment after the user clicked on one AD mode button which triggered differentiation (step 6), the TAPENADE server sends the differentiation output in a new web page, shown on figure 24. This graphical user interface helps examine TAPENADE output, exhibiting

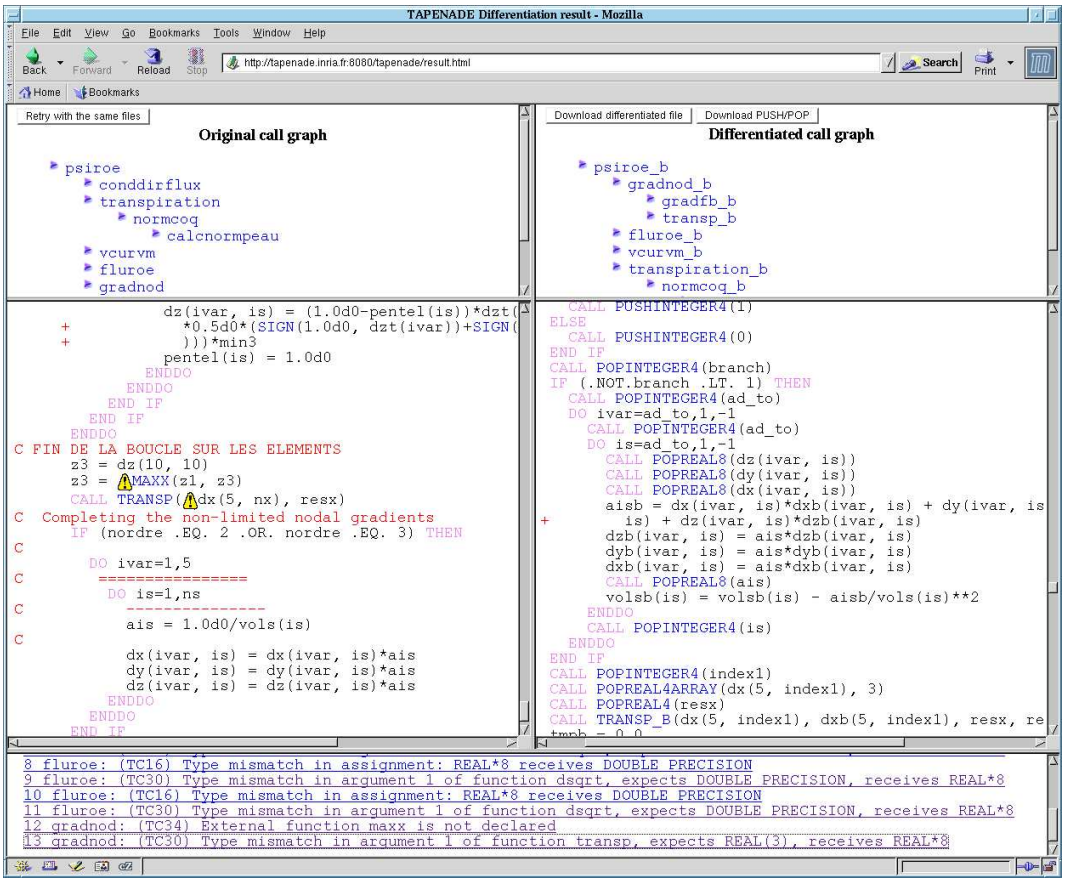


Figure 24: HTML interface for TAPENADE output

correspondence between original and differentiated code, as well as warning messages. The same interface is used when the user adds the `-html` option to the command-line `tapenade`.

The four top frames show the original program on the left, and the differentiated program on the right. The two top frames show the call graphs, and the middle frames show the FORTRAN source of the selected subroutine. Selecting a procedure name in a call graph frame triggers display of this procedure's source in the frame below. Selecting an instruction in

either source frame makes the other source frame scroll, to show the corresponding area. Danger signs ⚠ in the source frames are links to warning messages, which are displayed in the bottom frame, and reciprocal links go from the message text to the location in the source frame.

In addition, the output window has links to download the resulting differentiated program to your system, to download the source for the `PUSH*` and `POP*` utility subroutines for the reverse mode.

The "Back" button of your browser will take you to a new TAPENADE input web page, which may have lost all updated files. To avoid that, use the "Retry with same files" button from the output interface page, to go back to the input interface page without loosing the data you filled in.

8 ARCHITECTURE AND PERFORMANCES

Figure 25 summarizes the architecture of TAPENADE. At the source language level, the input parsers and the output "pretty-printers" are clearly separated from the kernel, which only knows an abstract language called IL (for "Imperative Language"). Ideally, IL contains all the semantic constructs that exist in the most common imperative languages we are targeting at, i.e. the FORTRANs and the Cs. The kernel of TAPENADE is an "Imperative Language Analyzer", that builds a Call Graph of the program, with a Flow Graph for each procedure. This kernel is responsible for all general purpose analyzes, such as Read-Written analysis or pointer Analysis.

Through an API on top of this kernel (which is maybe not as clear-cut as it should), is built the AD engine strictly speaking. It implements the AD-specific analyses, and builds the differentiated call graph and flow graphs back into the kernel. We claim that other tools could be built this way on top of the same kernel. The kernel is then able to send the differentiated IL programs to the pretty-printer of the appropriate language. Notice also in the middle bottom of figure 25 the "black-box signatures", which stands for the user-written library files described in section 6.2

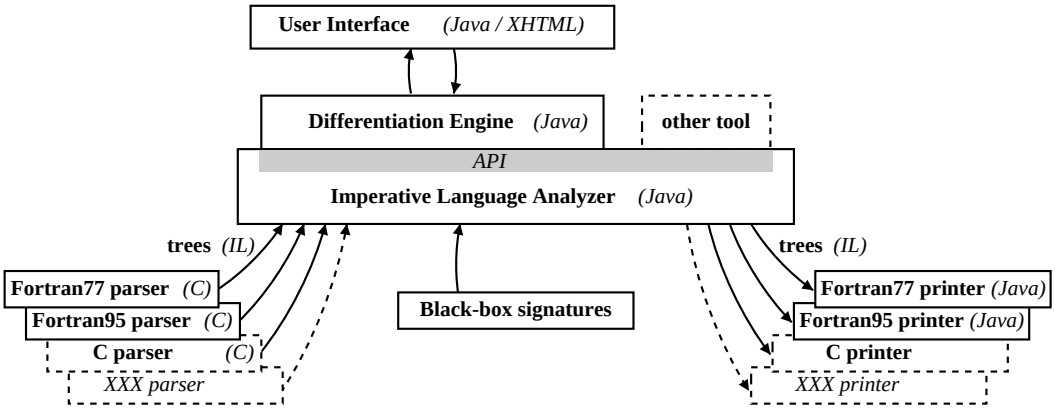


Figure 25: Overall architecture of TAPENADE

The chain of analyses that lead to differentiated programs is shown on figure 27. The topmost half is done inside the AD-independent kernel, and the bottom half is done inside the AD engine. Figure 27 illustrates the partial order coming from the dependencies between the analyses.

Some of these analyses are more time consuming than others: Pointers, Dependency, and Differentiable Dependency. All these analyses actually propagate through the program a piece of data-flow information. At each location in the program, i.e. before or after each instruction, this data-flow information associates an elementary piece of information to each accessible variable. For most analyses (Read-Written, Activity, Adjoint Liveness, TBR, and

Adjoint Written), this elementary piece of information is a handful of booleans. On the other hand for Pointers, Dependency, and Differentiable Dependency this elementary piece of information is itself a vector of booleans, one for each accessible variable. The reason is that dependency analysis, for instance, computes for each variable, the list of all variables it depends on. The propagated pieces of information are larger, their manipulation is slower. The differentiation time is therefore dominated by these three analyses, whose complexity is grossly the number of instructions in the program, times the square of the number of declared variables.

To terminate this section, let us illustrate the performances of the differentiated code itself, on several examples coming from large real applications. In theory, the tangent differentiated code should take a small multiple (3 to 4) of the time of the original code, and the reverse differentiated code should also take 4 to 5 times the time of the original code. In practice, the reverse code may take much longer because these estimations do not take into account the cost of storing intermediate variables, let alone the cost of checkpointing, which is nearly always necessary (*cf* section 3.2). Nevertheless, the reverse mode remains a very interesting choice to obtain the derivatives of a small number of outputs with respect to a large number of inputs.

Name:	ALYA	UNS2D	THYC	LIDAR	STICS	MARGARET
Domain:	*CFD*	*CFD*	*Thermo*	*Optics*	*Agronomy*	*Nuclear*
t(P) :	0.85	2.39	2.67	11.22	1.80	0.00027
t($\dot{P}$) :	1.69	7.17	5.57	12.23	3.80	0.00130
t($\dot{P}$)/t(P) :	2.0	3.0	2.1	1.1	2.1	4.8
t($\overline{P}$) :	4.62	24.78	10.99	22.99	35.70	0.00271
t($\overline{P}$)/t(P) :	5.4	10.4	4.1	2.0	19.8	10.0
Memory($\overline{P}$) :	9.4	259	3334	16.5	230	0.1

Figure 26: Time and memory consumption on six validation codes

Figure 26 shows the times in seconds and the memory consumption for intermediate variables and snapshots in megabytes. Measurements were made on a PC running Linux, with GNU or PGF compilers, and/or on a SUN workstation with the SUN `f77` compiler. Changing the compiler doesn't change the ratios significantly.

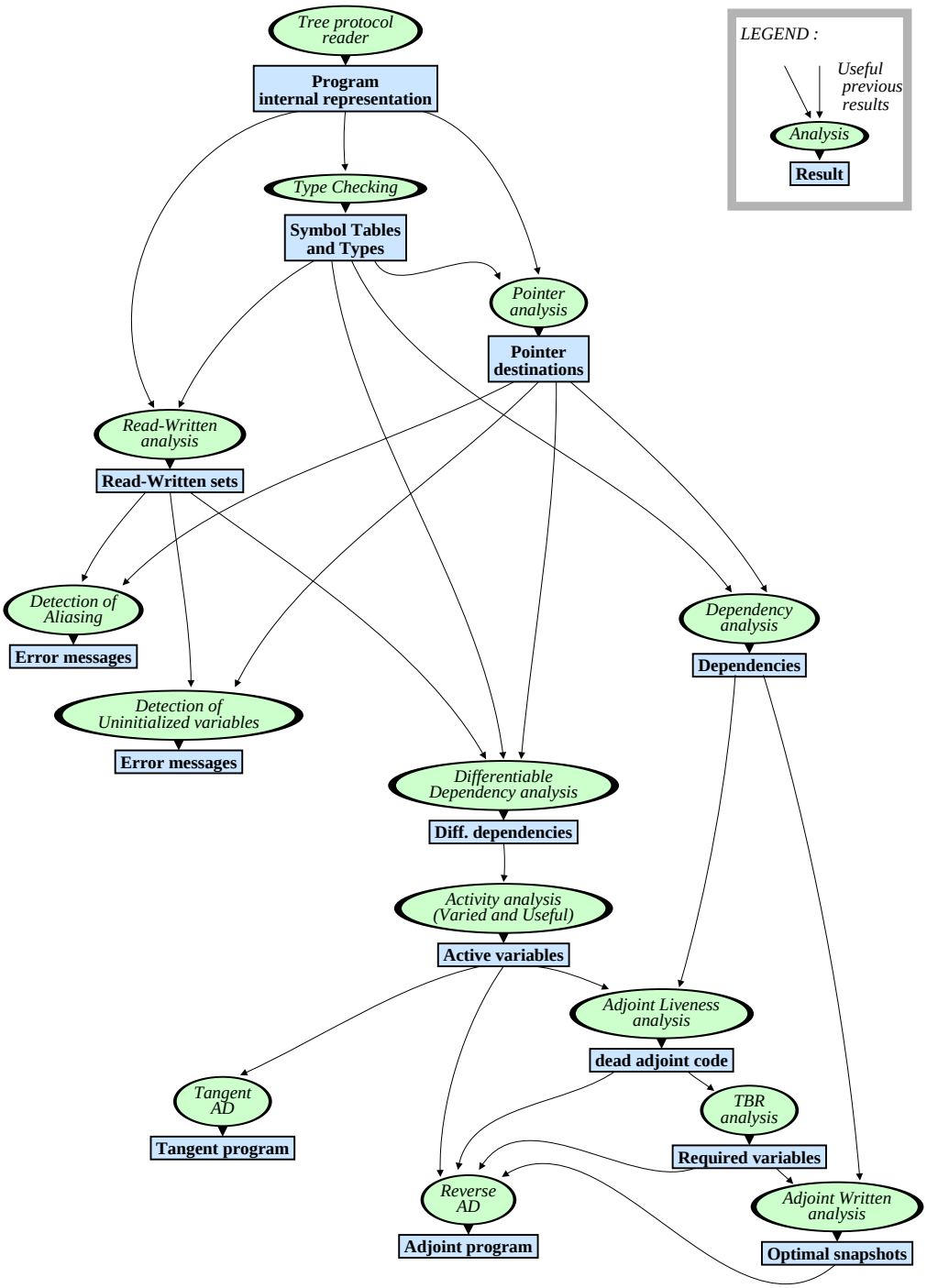


Figure 27: Internal Data Flow analyzes needed by AD in TAPENADE

9 VALIDATION AND TUNING

Let's face it, TAPENADE will probably not produce a perfect differentiated program at first try. After Automatic Differentiation, a validation step is necessary to check the derivatives computed. Section 9.1 proposes a framework to perform this validation. As long as validation doesn't work, the end-user must identify the problem and if possible modify the input program to solve the problem. Section 9.2 describes the most frequent problems found. It may happen that you find an unknown problem in the source produced by TAPENADE: in this case, please keep us informed by sending a mail to

`tapenade@lists-sop.inria.fr`

When the computed derivatives are finally correct, it may be good to check whether some well-known improvements are applicable to the differentiated code. This is described in section 9.3.

9.1 A classical framework for validation

Recalling the notations of section 3, we differentiate a program `P` that computes a function F, with input $X \in I\!R^n$ and output $Y = F(X) \in I\!R^m$.

Classically, one validates the results of the tangent mode by comparing them with divided differences, i.e. by applying the well-known formula for a differentiable function f of a scalar variable $x \in I\!R$:

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{5}$$

We recall that for a given direction $\dot{X}$ in the input space, the output of the tangent mode should be $\dot{Y} = F'(X) \times \dot{X}$. Introducing function g of scalar variable h: $g(h) = F(X + h \times \dot{X})$, expanding g' at input $h = 0$ with equation (5) (on the left hand side) and with the chain rule (on the right hand side) tell us that

$$\lim_{\varepsilon \to 0} \frac{F(X + \varepsilon \times \dot{X}) - F(X)}{\varepsilon} = g'(0) = F'(X) \times \dot{X} = \dot{Y} \tag{6}$$

so that we can approximate $\dot{Y}$ by running F twice, on X and on $X + \varepsilon \times \dot{X}$

The results of the reverse mode are validated in turn by using the validated tangent mode. This is called the *"dot product"* test. It relies on the observation that for any given $\dot{X}$, the result $\dot{Y}$ of the tangent mode can be taken as the input $\overline{Y}$ of the reverse mode, yielding a result $\overline{X}$. We then develop the dot product of $\overline{X}$ and $\dot{X}$:

$$(\overline{X} \cdot \dot{X}) = (F'^*(X) \times \dot{Y} \cdot \dot{X}) = \dot{Y}^* \times F'(X) \times \dot{X} = \dot{Y}^* \times \dot{Y} = (\dot{Y} \cdot \dot{Y}) \tag{7}$$

so that we can compare $\overline{X}$ returned by the adjoint code with the $\dot{Y}$ returned by the tangent mode.

This results in the following framework to validate the code produced by TAPENADE, shown on figure 28. Suppose the root differentiated procedure that implements the function

```
      SUBROUTINE VALIDATION(X,Y,n,m)
      INTEGER n,m
      REAL X(n), Y(m)
      REAL Xsave(n), Xdsave(n), Xd(n), Xb(n)
      REAL Yeps(m), Ydeps(m), Yd(m), Yb(m)
      REAL eps, normEps, normTgt, normAdj

!  Choose epsilon for divided differences
      eps = 1.e-10
!  Save the initial input X
      Xsave(:) = X(:)
!  Create the input differentiation direction Xd
      Xdsave(:) = 1.0
!  Prepare the X+epsilon*Xd input for F
      X(:) = Xsave(:) + eps*Xdsave(:)
!  Run Yeps = F(X+epsilon*Xd)
      call FOO(X,Y)
      Yeps(:) = Y(:)
!  Restore the initial input X
      X(:) = Xsave(:)
!  Restore the differentiation direction Xd
      Xd(:) = Xdsave(:)
!  Run the tangent code Y;Yd = F_D(X, Xd)
      call FOO_D(X, Xd, Y, Yd)
!  Compute the square norm of Yd from divided differences
      Ydeps(:) = (Yeps(:)-Y(:))/eps
      normEps = SUM(Ydeps(:)*Ydeps(:))
      print *, 'Divided Differences : ',normEps
!  Compute the square norm of Yd from tangent AD
      normTgt = SUM(Yd(:)*Yd(:))
      print *, 'AD Tangent mode     : ',normTgt
!  Restore the initial input X
      X(:) = Xsave(:)
!  Initialize Yb = Yd
      Yb(:) = Yd(:)
!  Run the adjoint code Xb = F_B(X, Yb)
      call FOO_B(X, Xb, Y, Yb)
!  Compute the dot product (Xb . Xd)
      normAdj = SUM(Xb(:)*Xd(:))
      print *, 'AD Reverse mode     : ',normTgt

      END
```

Figure 28: Subroutine framework for validation of derivatives

F is the subroutine `FOO(X,Y)`, where `X` is an array of n real numbers, which are the inputs X of F, and `Y` is an array of m real numbers, which are the outputs Y of F. This validation method requires us to call `FOO` repeatedly, and its results must be reproducible. Therefore `X` must really contain *all* inputs to F. There must be no hidden side-effect inputs on which we have no control.

In reality, if the root procedure has many different inputs and outputs of different shapes and types, it is absolutely not required that they are gathered in one array such as `X` and `Y`. We do this here only to make the explanation simpler. Variables `X` and `Y` may even overlap if some variables are at the same time "in" and "out". Subroutines `FOO_D` and `FOO_B` are produced by differentiation with TAPENADE, with the following simplifying assumption which can be lifted easily: we set the independents to all `X` and the dependents to all `Y`.

If the computed derivatives are correct, then the three numbers printed by the procedure on figure 28 must be roughly the same. Usually, the tangent norm and the adjoint norm match very well, up to the last few digits. The norm obtained with Divided Differences matches only to half the machine precision, sometimes after some adaption of the value of `eps`. In general, this is enough to be convinced that the derivatives are correct.

9.2 What if validation fails?

If validation fails, we must look in more detail at the differentiated programs. First, one must make the tangent norm match the Divided Differences norm, and then the adjoint norm must match the tangent norm.

Even worse, validation may fail simply because of arithmetic exceptions, returning results like `NaN` or `Inf`. Actually differentiation can introduce NaN's in the resulting program, because programs are sometimes non differentiable, and AD is then in trouble, for example for `SQRT` on zero or exponentiation `x**y` when `x` is zero. `NaN`s can also appear for a slightly different reason: for some given inputs, some functions have an output which is large, but not yet in overflow, whereas the derivative is definitely in overflow. Think of `1/x`, whose derivative is `-1/x**2`. Again this requires a special treatment, although a brute-force solution could be to switch to double precision. However, there are situations where TAPENADE can do a slightly better job automatically. For `SQRT` in the tangent mode, instruction:

```
a = SQRT(b)
```

generates differentiated instruction:

```
ad = bd/(2.0*SQRT(b))
```

which returns `NaN` if `b` is zero. But if `bd` is also zero, we feel it is safe to return zero in `ad` too. Thus in this case, TAPENADE avoids the above instruction and sets `ad` to zero instead. Something similar will be done in the reverse mode in the future versions.

Another general remark is that TAPENADE emits a number of warning messages during differentiation. In section 6.3, we describe some messages that may actually indicate why the derivatives are invalid. Think of all programming techniques where `REAL` values are stored temporarily into `INTEGER`s or into files, therefore loosing their derivatives. Also, TAPENADE performs a number of optimizations on the differentiated program, which can be delicate and certainly not bug-free. A good strategy can be to retry differentiation, this time after

disactivating optimizations with the `-nooptim` option (*cf* section 6.1). If the problem appears only with some optimization, it is a precious indication you can give us!

A last general remark before we examine specifically debugging of the tangent mode and of the reverse mode is that TAPENADE makes hypotheses on the memory size occupied by primitive data types. For example a `DOUBLE PRECISION` number is supposed to take 8 bytes. This is crucial for solving equivalences an different splittings of a given `COMMON`, and in the reverse mode this conditions the way a variable is `PUSH`ed and `POP`ped. If TAPENADE size hypotheses are different from the actual sizes on the target architecture, then the differentiated program is likely to crash (segmentation fault) or give wrong results. In the `ADFirstAidKit` directory of the standard installation, the `testMemSize` program can help you find out these actual sizes, and the `-i`, `-r`, and `-dr` command line options must be used to modify the default choices of TAPENADE accordingly.

9.2.1 Debugging tools for the tangent mode

For the tangent mode, we devised a way to locate the origin place of the mismatch. The general idea is to run two computations of derivatives, using Divided Differences on one hand, using TAPENADE's tangent code on the other hand, checking intermediate derivatives on the fly at places selected by the user. The first difference probably indicates where the bug lies. This is documented in the "Validation" section of the FAQ page in TAPENADE's web page, as it may evolve with time. It is necessary to modify the test-bed of figure 28 as explained in the "Validation" section, yielding what is shown on figure 29. Computing the Divided differences requires as usual a first run of procedure `FOO`, but we call `FOO_D` instead, which anyway also computes the output of `FOO`, but in addition stores the intermediate values selected by the user into a file called `epsvalues`. The second run of `FOO_D` effectively computes the tangent derivatives, and in addition to that progressively recovers each value stored in `epsvalues`, uses it together with the present intermediate value to compute the derivative with Divided Differences, and immediately compares it with the tangent derivative and complain when the difference is more than `epszero`. Distinction between these two behaviors of `FOO_D` is made through the global `phase` integer. At any place in the source of `FOO_D`, you may select a value so that its derivative is checked with the above mechanism, simply by inserting procedure calls like the following:

```
call TRACEACTIVEARRAY1REAL("T",t,td,1,10)
```

which will compare at run-time the Divided Differences and the tangent derivatives of elements 1 to 10 of array `T` and print a message if they differ. These procedures are provided in the `ADFirstAidKit` directory of the standard installation.

9.2.2 Debugging tools for the reverse mode

As usual, things are not so easy for the reverse mode, but we also devised a rudimentary divide-and-conquer approach to run the dot-product test on parts of the procedure `FOO`. Bugs can thus be located precisely, in a relatively small part of the program, and may be tracked down more easily. Suppose we choose locations $\mathtt{L}_j, j = 0, p$ in "the middle of" `FOO_D`,

```
      SUBROUTINE VALIDATION(X,Y,n,m)
      ...
      real*8 ddeps, epszero
      integer phase, ddfile
      common /comphase/ phase, ddfile, ddeps, epszero

      ...
!  Run Yeps = F(X+epsilon*Xd)
      Xd(:) = Xdsave(:)
      ddeps = eps
      epszero = 0.00000001
      phase = 1
      ddfile = 37
      OPEN(37, FILE='epsvalues')
      call FOO_D(X, Xd, Y, Yd)
      Yd(:) = 0.0
      CLOSE(37)
      Yeps(:) = Y(:)
!  Restore the initial input X
      ...
!  Run the tangent code Y;Yd = F_D(X, Xd)
      phase = 2
      OPEN(37, FILE='epsvalues')
      CALL FOO_D(X, Xd, Y, Yd)
      CLOSE(37)
!  Compute the square norm of Yd from divided differences
      ...
```

Figure 29: Subroutine framework for debugging the tangent code

with obvious corresponding points inside the backward sweep of `FOO_B`. The constraint is that at each $\mathtt{L}_j$, all active variables are visible: in other words, no active variable can be in a common which is not declared and visible for $\mathtt{L}_j$. This is the most stringent restriction on the approach. Call these variables X_j, and we will assume they are the elements of an array `Xj` for simplicity in the description only. Set $\mathtt{L}_0$ to the very beginning of `FOO`, and $\mathtt{L}_p$ to its very end. We are going to run the dot-product test on each piece of `FOO` that goes from a location $\mathtt{L}_j$ to $\mathtt{L}_{j+1}$. Each time execution of `FOO_D` reaches a location $\mathtt{L}_j$, all derivatives $\dot{X}_j$ are stored, and the dot product $(\dot{X}_j \cdot \dot{X}_j)$ is computed and printed. Conversely, each time execution of the backward sweep of the adjoint `FOO_B` reaches the corresponding location $\mathtt{L}_j$, the stored value of $\dot{X}_j$ is retrieved, its dot product with the present $\overline{X}_j$ (which is going upstream with the adjoint code) is computed and printed, and last but not least $\overline{X}_j$ is overwritten with the contents of $\dot{X}_j$. Figure 30 illustrates the mechanism: Call F_j the function implemented

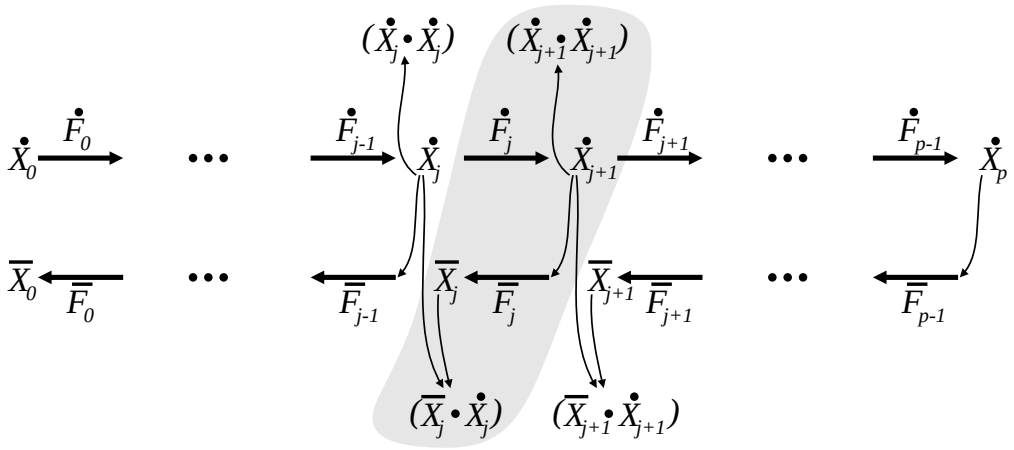


Figure 30: Split execution of the dot product test

by the fragment of `FOO` between $\mathtt{L}_j$ and $\mathtt{L}_{j+1}$. During `FOO_D`, the code fragment $\dot{F}_j$ from $\mathtt{L}_j$ to $\mathtt{L}_{j+1}$ computes, from its input tangent direction $\dot{X}_j$, its output tangent derivative $\dot{X}_{j+1} = F'_j(X_j) \times \dot{X}_j$. During `FOO_B`, the code fragment $\overline{F}_j$ from $\mathtt{L}_{j+1}$ to $\mathtt{L}_j$ computes, from its input $\overline{X}_{j+1}$ which has just been completely overwritten to contain $\dot{X}_{j+1}$, the gradient $\overline{X}_j = F'^*_j(X_j) \times \overline{X}_{j+1}$. For F_j just like for the complete F, it is then easy to check that:

$$(\overline{X}_j \cdot \dot{X}_j) = (\dot{X}_{j+1} \cdot \dot{X}_{j+1}) \tag{8}$$

The right hand side of (8) is what is printed by `FOO_D` at location $\mathtt{L}_{j+1}$, and the left hand side of (8) is what is printed by `FOO_B` at location $\mathtt{L}_j$. If the two numbers don't match, then the problem with the adjoint code lies between these two locations. One can then choose new locations to further reduce the place where the problem lies.

In practice, one needs to start from the initial test-bed shown on figure 28, and insert the appropriate subroutine calls inside `FOO_D` and `FOO_B`, at all locations $L_j, j = 0, p$, not forgetting L_0 at the very beginning nor L_p at the very end. Figure 31 shows what must be inserted at each location L_j, supposing that the active variables X_j at this location are exactly `u`, `v`, and array `t` (whose dimension is e.g. `t(3,s)`, therefore equivalent to a mono-dimensional array of size `3*s`). Keep in mind that the $\dot{X}_j$ are stored into a stack, and therefore must be popped from the stack in the reverse order. Subroutines

original program	TAPENADE **tangent**	TAPENADE **reverse (backward sweep)**
`...`	`...`	`...`
	`ud = 3.0*ud`	`ub = ub + v*vb`
`u = 3.0*u + 2.0`	`u = 3.0*u + 2.0`	`vb = u*vb`
	`call DPFWDREAL8(ud)`	`call DPREVREAL8ARRAY(tb,3*s)`
	`call DPFWDREAL8(vd)`	`call DPREVREAL8(vb)`
	`call DPFWDREAL8ARRAY(td,3*s)`	`call DPREVREAL8(ub)`
	`call DPFWDDISPLAY('Mid point')`	`call DPREVDISPLAY('Mid point')`
	`vd = u*vd + ud*v`	
`v = u*v`	`v = u*v`	`ub = 3.0*ub`
`...`	`...`	`...`

Figure 31: Procedure calls inserted for split dot product test

`DPFWDREAL8`, `DPFWDREAL8ARRAY`, `DPFWDDISPLAY` and the corresponding reverse `DPREVREAL8`, `DPREVREAL8ARRAY`, `DPREVDISPLAY` are all provided in the `ADFirstAidKit` directory.

For each location L_{j+1} for $j = 0$ up to $p-1$, `FOO_D` prints the dot product $(\dot{X}_{j+1} \cdot \dot{X}_{j+1})$ followed by the name of L_{j+1}. For each location L_j for $j = p$ down to 0, `FOO_B` prints the dot product $(\overline{X}_j \cdot \dot{X}_j)$ (equation (8) states it should be the same) followed by the name of L_j. With correct derivatives, one gets this sort of output (locations go from L_0 to L_3; numbers symbolized here by `nnnnn` and `mmmmm` are useless for the test and must be ignored):

```
>> DotProduct = nnnnn
Location L_0
>> DotProduct = 7827.3
Location L_1
>> DotProduct = 2.543E-02
Location L_2
>> DotProduct = 9.233E+07
Location L_3
<< DotProduct = mmmmm
Location L_3
<< DotProduct = 9.233E+07
Location L_2
<< DotProduct = 2.543E-02
```

```
Location L_1
<< DotProduct = 7827.3
Location L_0
```

9.3 Improving differentiated programs

This section is about modifying the differentiated program to improve its performances. Please bear in mind that this is a dangerous activity if done only by hand on the differentiated code: all modifications will be lost if the program is differentiated again. Therefore we insist that all hand-made modifications on the differentiated code should remain simple and of limited extent, and next developments of TAPENADE will automate these improvements during AD itself, probably with the help of directives inserted by the end user into the source program.

Despite the analysis effort done during differentiation, there will always remain unnecessary instructions in the differentiated code. There is even a theoretical justification for that: static analyses (compile-time analyses of programs) are non-decidable. In other words, there will always be programs on which the answer to a data-flow question is unknown. The tool that uses the analysis must therefore make a conservative choice, i.e. generate a code that copes with the worst-case scenario, and this has a cost.

For instance in all modes of AD, there may remain unnecessary initializations of derivative variables, because analyses couldn't tell whether this derivative will be used before it is next overwritten. Also, specifically in the reverse mode, some storage may have been inserted to write intermediate values on the tape or to take a snapshot, although the user knows that it is not necessary. Obviously, removing this unnecessary storage will improve the program in execution time as well as memory usage. However this must be done with care until a directive is available.

In the reverse mode, another source of inefficiency is a poor choice of checkpoints. Figure 5 summarizes where TAPENADE puts checkpoints. This choice is in general efficient when the call graph is well balanced. In other cases, it may be wise not to checkpoint some very small procedures, or to checkpoint selected parts of time-consuming procedures, especially for loops (*cf* [14]). Again, this should be done during AD, driven by user-given directives. Until this mechanism exists in TAPENADE, this can be achieved by changing the call graph of the original program, introducing new procedures or inlining existing procedures.

However, the most profitable improvements come from the user's knowledge of the original program's algorithm and meaning, rather than from a generic, short-sighted, static analysis of the program's source. A variety of directives may be very helpful to give this knowledge to TAPENADE. Some of these improvements may be automated partly in the meantime, using a combination of black-box mechanism and limited hand modification. We shall discuss below a number of such improvements, namely about *indeed inactive variables*, *Linear code fragments*, *auto-adjoint code fragments*, *iterative resolutions*, and *loops with independent iterations*.

9.3.1 Indeed inactive variables

Consider for example the computation of the time-step during the iterative resolution of a steady-state problem. Strictly speaking, this time step is computed at each iteration and depends on the current iterated state. Conversely this time step clearly influences the next iterated state, and therefore the final steady state. Therefore an AD tool will probably declare this time-step as an active variable.

However, the end-user knows that the time step should not influence the final result more than through minor approximation effects. It may be appropriate to force the AD tool to consider this time-step inactive. As a consequence, the whole piece of code that computes this time-step does not need to be differentiated, saving actually a lot of derivative computations.

Until a directive exists, one way to cheat TAPENADE could be to introduce an extra assignment:

```
time_step = DISACTIVATE(time_step)
```

where the black-box function `DISACTIVATE` essentially copies its input to its output, but whose signature in the `*ADLib` file specifies that the output depends on nothing and is therefore inactive.

9.3.2 linear code fragments

A code fragment may perform a linear operation. Consider the case where the active output Y of this fragment is a linear function of its active input X, i.e. $Y = M \times X$, where M itself is not active. This information can be hidden very deeply inside the fragment, and actually the fragment may perform a lot of non-linear intermediate operations. This is the case for example for `GMRES` [24] solvers.

An AD tool has no hope of detecting this situation. Therefore the differentiated code will probably differentiate the whole computation sequence, even if the user knows most of it is useless. Not only will the differentiated code perform useless operations, but moreover the principle of AD will use the iterative control that solves for Y to solve for $\dot{Y}$ or $\overline{X}$ too. This may prove dangerous, because the solver takes care of instabilities in the intermediate values that lead to Y, and the differentiated solver will not do the same for the intermediate differentiated values.

Therefore the user certainly knows better than the AD tool how this code fragment should be differentiated, in tangent mode as well as in reverse mode. This special case can be flagged by a directive. Until this exists, the obvious solution is again to use the black-box mechanism, and to provide a hand implementation of the differentiated fragment. This is quite easy in the case of a linear operation done by a procedure call:

```
call LINEAROP(X,Y)
```

One just needs to define the differentiated subroutine `LINEAROP_D(X,Xd,Y,Yd)` as:

```
call LINEAROP(Xd,Yd)
call LINEAROP(X,Y)
```

9.3.3 Auto-adjoint code fragments

Consider a linear code fragment like above $Y = M \times X$, and suppose in addition that $M^* = M$. To make the example more interesting, suppose also that M is not explicit: it is the inverse of another matrix N. Of course we don't want to compute N^{-1} explicitly, and we prefer to factorize N instead as $L \times U = N$. In the program, this may be gathered in a single call:

```
call AUTOADJ(X,Y,N,L,U)
```

Here again, reverse AD of `AUTOADJ` probably does not generate a very efficient code, because TAPENADE cannot guess that the adjoint `AUTOADJ_B` is equivalent to the original `AUTOADJ` itself. This special case can be flagged by a directive. Until this exists, the solution is again to make `AUTOADJ` a black-box subroutine.

```
AUTOADJ_B(X,Xb,Y,Yb,N,L,U)
```

can be defined simply by hand as:

```
Ab = 0.0
call AUTOADJ(Yb, Ab, N, L, U)
Xb = Xb + Ab
Yb = 0
```

With a little more effort, one can even put in common the factorization effort, which was probably already done during the forward sweep of the reverse code, in the first call to `AUTOADJ(X,Y,N,L,U)`.

9.3.4 Iterative resolutions

Consider an iterative resolution where only the final state matters. All the intermediate states, starting from the initial first guess, do not really matter for differentiation. Actually, there are many clever strategies to compute the derivatives, and especially the gradient, of an iterative resolution. One way to view it is to consider that only the last, converged state matters. If resolution iteratively goes from initial guess state s_0 to converged state s_x, the forward sweep of the reverse mode does not need to keep track of all intermediate values during steps s_0 to s_{x-1}. It can keep the last state s_x only, and the backward sweep will compute the adjoints iteratively staying on state s_x.

This is not the only strategy available, and there are still research problems in this question [17, 19]. For example the resolution that takes place during each iteration step is probably well adapted for solving the state s, but nothing proves its adjoint is the best for solving the adjoint state $\overline{x}$. Other strategies use a different solver for the adjoint state.

In any case, this decision is up to the end-user. An AD tool must help the user by providing the adjoints of basic bricks of the iterative program, and by letting open the choice of the strategy to glue these adjoint bricks together. This is slightly more than what the black-box mechanism can provide. Therefore, until directives exist for that, some hand-coding will be necessary.

9.3.5 Loops with Independent Iterations

Consider a loop whose iterations are independent, i.e. they could be done in any order. In this section, we call "iteration" each instance of the loop body. Precisely, suppose all the instructions in this loop are either *parallel* or *global sum reductions*. Call these loops *II*-loops. This case is more frequent than it seems: all *gather-scatter* loops on meshes are *II*-loops. It can be shown that an *II*-loop is parallel, and that its adjoint loop is parallel too, so that the two loops can be fused into a single loop. This is shown on figure 32. If we can

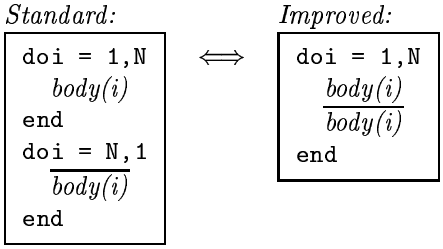


Figure 32: Optimization of adjoint II-Loops

manage to move the *II*-loop just before its adjoint loop, we reach the state on the left of figure 32. Since the two loops are parallel and operate on the same iteration space, they can be fused as shown on the right of figure 32. The advantage is that all the memory used to store the tape, i.e. the intermediate values during the forward sweep *body(i)* is immediately used during the backward sweep $\overline{body(i)}$ that follows and can thus be reused for the next iteration. The memory consumption is reduced by a factor N. The resulting code is even faster due to a better locality and a reduced swapping. Moreover, dead adjoint code can be removed from the end of *body(i)*.

Again, this is typically a job for directives. Before this is done, though, here is a reasonable partly automated implementation: in the original program, create a subroutine that contains only the *II*-loop. Inside this loop, create a subroutine that contains *body(i)*. This modified source is sketched in the left column of figure 33. Then call TAPENADE to perform reverse AD. The result is sketched in the middle column of figure 33. Observe that the loop immediately precedes its adjoint, apart from a trivial `PUSH/POP`. There could be some initializations of derivatives between the two loops: just move these initializations before the first loop. Observe also that the code for `BODY18_B` already benefits from dead adjoint code elimination, as the last instruction of `BODY18` has been removed. Then comes the manual step, which is to remove the indicated lines in subroutine `LOOP18_B`, to obtain the right column of figure 33. This is a small enough modification even if it must be repeated after each differentiation. The resulting code embeds the optimization of the adjoint *II*-loop.

original program	**reverse mode**	**reverse mode, after manual improvement**

`...`
`call LOOP18(...,N)`
`...`
 	
`SUBROUTINE LOOP18(...,N)`
`...`
`INTEGER i,N`
`DO i=1,N`
`  call BODY18(...,i)`
`ENDDO`
`END`
 	
`SUBROUTINE BODY18(...,i)`
`...`
`INTEGER i`
`...`
`res(i)= res(i) + v`
`END`
	`...`
	`call LOOP18(...,N)`
	`...`
	`call LOOP18_B(...,N)`
	`...`
	
	`SUBROUTINE LOOP18_B(...,N)`
	`...`
	`INTEGER i,N`
	`DO i=1,N`
	`  call PUSH...`
	`  call BODY18(...,i)`
	`ENDDO`
	`CALL PUSHINTEGER4(i-1)`
	`CALL POPINTEGER4(ad_to)`
	`DO i=ad_to,1,-1`
	`  call POP...`
	`  call BODY18_B(...,i)`
	`ENDDO`
	`END`
	
	`SUBROUTINE BODY18_B(...,i)`
	`...`
	`INTEGER i`
	`...`
	
	`vb = vb + resb(i)`
	`...`
	`END`
		`...`
		`call LOOP18(...,N)`
		`...`
		`call LOOP18_B(...,N)`
		`...`
		
		`SUBROUTINE LOOP18_B(...,N)`
		`...`
		`INTEGER i,N`
		`DO i=1,N`
		
		`  call BODY18_B(...,i)`
		`ENDDO`
		`END`
		
		`SUBROUTINE BODY18_B(...,i)`
		`...`
		`INTEGER i`
		`...`
		
		`vb = vb + resb(i)`
		`...`
		`END`

Figure 33: Partly automatic reverse differentiation for *II*-loops

10 KNOWN PROBLEMS AND DEVELOPMENTS TO COME

We conclude this user's guide of TAPENADE by a quick description of known problems, and how we plan to address them in the next releases. We are not talking here about the bugs in TAPENADE, which of course exist among the 70 000 lines of JAVA source. Rather, we focus on missing functionalities. We suggest you check the TAPENADE web site to know which new functionalities appeared after this guide was written.

10.1 Includes, comments, and declarations

When a FORTRAN source contains `INCLUDE`s, TAPENADE reads and uses the include files. During the analysis, the internal representation contains only the symbol tables built from the declarations and include files. The internal representation doesn't contain the textual declarations nor the `INCLUDE` commands any more. This is why the produced differentiated programs have a declaration part which is completely rewritten. For example the include files are not used there: their contents are *inlined* into the new declarations. Also the order and style of the declarations is changed according to TAPENADE's taste, which is arbitrary. Even more disturbing, the comments which were attached to these declarations are now floating at the end of the declaration section, which is inconvenient.

Although not vital for AD, this is a serious problem, and we plan to fix it by keeping a list of the original declarations lines, and using these original declarations to produce the new differentiated source program.

10.2 Directives

We mentioned several times the need for *directives*, to let the end-user give additional information to TAPENADE. Directives are interesting because they can provide information relative to a given location in the source. Furthermore, they remain in the source program when it is modified, providing cumulative AD knowledge into the source code.

We are thinking about several uses for directives, such as locating special pieces of code like iterative resolutions (*cf* section 9.3.4), *II*-loops (*cf* section 9.3.5), linear or auto-adjoint functions (*cf* section 9.3.3).

The most important use of directives will be to drive *checkpointing*. The present TAPENADE systematically checkpoints each procedure call. This choice should be modifiable by the end-user to take advantage of the relative different costs of each procedure.

Directives will be available in the next versions of TAPENADE.

10.3 Input languages

For this version on, TAPENADE accepts FORTRAN77 and FORTRAN95 as input. Work to accept C as an input language will start soon.

On the other hand, extension to object-oriented languages is not yet a priority. This is still in the research area and will not be available in a foreseeable future.

Several users asked for a version of TAPENADE running on WINDOWS. This should be available very soon.

10.4 Pointers and dynamic allocation

Full AD on FORTRAN95 supposes pointer analysis, and an extension of the AD models on programs that use dynamic allocation. This is not done yet.

Whereas the tangent mode does not pose major problems for programs with pointers and allocation, there are problems in the reverse mode. For example, how should we handle a memory deallocation in the reverse mode? During the reverse sweep, the memory must be reallocated somehow, and the pointers must point back into this reallocated memory. Finding the more efficient way to handle this is still an open problem.

10.5 Parallel programming

We made very few tests on parallel programs. Apparently, TAPENADE does not differentiate all sorts of parallel programs. This version 2.1 handles the vectorial notation of FORTRAN95 for the most frequent cases. TAPENADE can also use previous work [8, 7] with ODYSSEE [10] on differentiating programs with MPI communications. But there are plenty of cases where parallel programs are not treated well.

For example, TAPENADE does nothing to keep the differentiation of a parallel loop parallel. Consider a loop with the well known CRAY directive `IVDEP`, meaning that a loop is vectorizable. The differentiated loop in the reverse mode should be vectorizable in most cases. However, TAPENADE will probably have inserted several calls to `PUSH*` and `POP*`, which make the loop non-parallel. This problem can be solved by changing the way the tape is stored inside parallel loops.

10.6 The Recompute-All strategy

The `PUSH*` and `POP*` strategy to store the tape and the snapshots is relatively versatile and robust. However in many cases it is not the most efficient approach. At some locations in the program, other approaches can be proposed, probably triggered by directives. From more robust to more efficient, we might:

- use `PUSH*` and `POP*` in an external memory allocated dynamically
- use `PUSH*` and `POP*` to statically allocated memory.
- replace the `PUSH*` and `POP*` by assignments to local storage arrays.
- use selected variables in Single-Assignment mode, i.e. use new, different variables to avoid overwriting. If a variable is never overwritten, there is no need to save it.

More generally, the alternative Store-All approach sometimes gives better performances. In our research activity, we are investigating optimal trade-offs between RA and SA strategies, where some variables are stored and others are recomputed, according to properties of the data-flow or data-dependence graphs. This may result in new functionalities of TAPENADE in the long run.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

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

[3] M. Buecker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *AD2004 Post-Conference Special Collection.* Lecture Notes in Computational Science and Engineering. Springer, 2004. to appear.

[4] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[5] G. Corliss, Ch. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization.* Computer and Information Science. Springer, New York, NY, 2001.

[6] F. Courty, A. Dervieux, B. Koobus, and L. Hascoët. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software*, 18(5):615–627, 2003.

[7] P. Dutto, C. Faure, and Fidanova S. Automatic differentiation and parallelism. In *Proceedings of Enumath 99, Finland*, 1999.

[8] Ch. Faure and P. Dutto. Extension of Odyssée to the MPI library - Reverse mode. Technical Report 3774, INRIA, 1999.

[9] Ch. Faure and U. Naumann. Minimizing the tape size. In G. Corliss, Ch. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 34, pages 293–298. Springer, New York, NY, 2001.

[10] Ch. Faure and Y. Papegay. Odyssée User's Guide. Version 1.7. Technical Report RT–0224, INRIA, Sophia-Antipolis, France, 1998.

[11] R. Giering. *Tangent Linear and Adjoint Model Compiler, Users Manual.* Center for Global Change Sciences, Department of Earth, Atmospheric, and Planetary Science, MIT, 1997. Unpublished [url http://www.autodiff.com/tamc].

[12] R. Giering and T. Kaminski. Generating recomputations in reverse mode AD. In G. Corliss, Ch. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, chapter 33, pages 283–291. Springer Verlag, Heidelberg, 2002.

[13] M.-B. Giles. Adjoint methods for aeronautical design. In *Proceedings of the ECCOMAS CFD Conference*, 2001.

[14] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992. Also appeared as Preprint MCS–P228–0491, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991.

[15] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[16] A. Griewank and G. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, PA, 1991.

[17] A. Griewank and Ch. Faure. Reduced Gradients and Hessians from Fixed Point Iteration for State Equations. *Numerical Algorithms*, 30(2):113–139, 2002.

[18] L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. Preprint ANL-MCS/P936-0202, Argonne National Laboratory, 2002. also *Rapport de recherche number 4856*, INRIA.

[19] L. Hascoët, M. Vázquez, and A. Dervieux. Automatic Differentiation for Optimum Design, Applied to Sonic Boom Reduction. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 85–94, Berlin, 2003. Springer.

[20] INRIA Tropics team. On-line documentation of the Tapenade AD tool. Technical report. [url http://www.inria.fr/tropics].

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

[22] M. Metcalf and J. Reid. *Fortran 90/95 explained.* Oxford University Press, 1996.

[23] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1039–1048, Berlin, 2002. Springer.

[24] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *J. Sci. Comput.*, 7:856–869, 1986.

Index

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 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 - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803