Computing Adjoints by Automatic Differentiation with TAPENADE

Laurent Hascoët[*], Rose-Marie Greborio[*], Valérie Pascual[*]

November 18, 2002

Abstract

We present the Automatic Differentiation (AD) tool TAPENADE, with emphasis on the so-called *reverse mode* which computes *gradients*. Computing gradients with the reverse mode is the discrete equivalent of writing and then solving the *adjoint* equations. We present the main usages of AD, and its fundamental model, based on the *chain rule*. We detail the architecture and algorithms used in TAPENADE, highlighting the strategies used for an efficient reverse mode. We present the user interface of TAPENADE and give pointers to the on-line documentation. We describe the difficulties that AD tools still have to overcome to reach a larger audience, and how we plan to address them.

1 Introduction

In Scientific Computing, the domains of *simulation*, *optimization*, and *inverse problems*, are developing rapidly due to the increase in computational power. These domains widely use *derivatives*. These derivatives can be written in equations and then discretized and solved. This is for example the case for the *adjoint* equations used in inverse problems. Alternatively, when these are derivatives of functions already discretized and solved, one can use Automatic Differentiation (AD) to get the derivatives without going back to the discretization step. The technique of AD just transforms a given program, so that the new program computes some derivatives of the original program [10]. In particular the *reverse mode* of AD computes the "transposed Jacobian times vector" products that appear in the *adjoint* equations.

[*]Projet Tropics, INRIA, France

AD is based on some simple assumptions, such as identification of programs with compositions of functions, one per instruction. AD uses the *chain rule* to compute analytical derivatives, and thus gets rid of approximation errors. This also leads to *limitations* of AD, in particular due to analytical non-differentiability when the program's control switches.

While AD is driven by its applications, mostly from scientific computing, the software engineering concepts that are used inside AD tools are similar to those used inside compilers. In particular *data-flow analyses*, *data-dependence analysis* [1] are of central importance. Our research team develops and distributes an AD tool, TAPENADE, that has a strong focus on the reverse mode of AD. TAPENADE uses the above concepts of compilation to produce more efficient differentiated programs. In the current state of the art, it is necessary to understand these concepts to gain confidence in the Automatic Differentiation process.

The goal of this paper is to present the basic ideas of AD, and the central algorithms of TAPENADE, to promote the use of this tool to compute gradients of functions implemented by programs.

In section 2, we present the context of AD, the fundamental idea of differentiating *programs*, and the AD model, especially for the reverse mode. Section 3 describes in greater detail the architecture, functionalities, and algorithms of TAPENADE. Section 4 rapidly describes the practical utilization of TAPENADE, and two aspects of the user interface: the diagnostics messages and the way to handle external ("black-box") subroutines. Section 5 concludes with some perspectives on future research in AD.

2 What is Automatic Differentiation?

Automatic Differentiation is a technique to evaluate derivatives of a function $F : X \in \mathbb{R}^m \mapsto Y \in \mathbb{R}^n$ defined by a computer program `P`. In AD, the original program is automatically transformed into a new program `P'` that computes the derivatives *analytically*. For further reference, see the monography [10], selected articles of a recent conference [3], or the AD community website at `www.autodiff.org`.

2.1 What is AD for?

Let us first recall what derivatives may be needed, and for what usage. Here is a non-exhaustive list:

- The so-called *tangent* (or *forward*) derivatives compute the first-order variation of the program's outputs Y, for a given small variation of X

following a given direction $\dot{X} \in I\!R^m$. Formally, tangent derivatives are the product of the Jacobian F' of F by $\dot{X}$. Tangent derivatives are used for quick approximate simulation of the behavior of a system, for small variations of its inputs along a fixed direction. Tangent derivatives are also used to evaluate the sensitivity of the program's results with respect to variations of some of its arguments. They are also needed to study the influence of truncation errors.

- The so-called *reverse* (or *gradient*) derivatives compute the relative influence of the program's inputs X on some combination of its outputs Y given by a weight vector $\overline{Y} \in I\!R^n$. Formally, this is the product of the *transposed* Jacobian of F by $\overline{Y}$. Reverse derivatives are used in optimization problems, to implement gradient descent algorithms. Reverse derivatives also play a central role in inverse problems, such as data assimilation or parameter estimation. Reverse differentiation is the discrete equivalent of the *adjoint equations* technique.
- One can think of another mode of differentiation, that explicitly computes the Jacobian matrix F', either by an ad-hoc transformation, or by a clever usage of tangent or reverse differentiation. Among many other utilizations, Jacobian matrices provide a complete first-order simulation of a complex system, for any small variations of its inputs.
- The above modes of differentiation can be extended to higher degrees of differentiation. For example, experiments were made that yield Taylor expansions in a given direction, or also Hessian tensors. These higher-order derivatives are used for better simulation and optimization, and for quicker convergence of optimization loops.

2.2 How to compute derivatives

How can we obtain these derivatives? Automatic Differentiation need not go back to the equations that define the function F. AD only needs the source of the program `P`, and builds a new, augmented program `P'`, that computes the analytical derivatives along with the original F. Basically, each time the original program holds some variable `v`, the differentiated program holds an additional variable with same shape, that we call the *differential* of `v`. Moreover, for each operation in the original program, the differentiated program performs additional operations dealing with the differential variables. For example, suppose that the original program comes to executing the following instruction on variables `a`, `b`, `c`, and array `T`:

```
a = b*T(10) + c                                   (1)
```

Suppose also that variables $\dot{\texttt{b}}$, $\dot{\texttt{c}}$, and array $\dot{\texttt{T}}$ are available and contain one particular sort of differential: the tangent derivatives, i.e. the first-order variation of `b`, `c`, and `T` for a given variation of the input. Then the differentiated program must execute additional operations that compute $\dot{\texttt{a}}$, using `b`, `c`, `T` and their differentials $\dot{\texttt{b}}$, $\dot{\texttt{c}}$, and $\dot{\texttt{T}}$. These must somehow amount to:

$$\dot{\texttt{a}} = \dot{\texttt{b}}*\texttt{T(10)} + \texttt{b}*\dot{\texttt{T}}\texttt{(10)} + \dot{\texttt{c}} \quad (2)$$

The derivatives are computed analytically, using the well known formulas on derivation of elementary operations. Approximation errors, which are a problem if using Divided Differences, have just vanished.

2.3 Overloading

At this point, let us mention an elegant manner to implement the above transformation: *overloading.* Overloading is a programming technique, available in several languages, where one can redefine the semantics of basic functions (such as arithmetic operations), according to the type of their arguments. For example, instruction (1) can easily subsume instruction (2), if only the type of variables is changed from `REAL` to pairs of `REAL`, and the semantics of `+`, `*`, etc, are augmented to compute the derivatives into, say, the second component of the above pairs of `REAL`. The advantage of overloading is that it requires very little code transformation. Drawbacks are that not all languages support overloading, that overloaded programs are poorly optimized by the compiler, and more importantly that overloading is not suitable for the *reverse* mode of AD. We shall not consider overloading in the sequel.

2.4 Discontinuities due to Control Flow

Control structures, such as conditionals or loops, introduces discontinuities in the program. A small modification of the inputs sometimes makes the control switch. In this case, the true derivatives may be undefined. This is a problem, because the differentiated program supposes the control is fixed, and therefore actually computes a derivative, which may be meaningless. In real codes however, this problem seldom arises. Moreover, for important cases, such as iterative loops, some results [8] guarantee that the AD loop actually converges towards the derivative of the original loop (but maybe not at the same speed). Until tools exist that detect these discontinuities, AD derivatives still have to be validated by comparison with Divided Differences. This warning must be kept in mind: AD tools suppose that, in a neighborhood of the original inputs, the program control is constant and therefore the program is identified with a fixed sequence of instructions.

2.5 General model for A.D.

We are now able to describe formally the general model of Automatic Differentiation: In some neighborhood of the input X, we assume as stated above that P is identified with a sequence of assignments $I_k, k \in [1..p]$, and therefore is identified with the composition of mathematical functions

$$F = f_p \circ f_{p-1} \circ \ldots \circ f_1$$

where each f_k is the elementary function implemented by instruction I_k. Each f_k operates on a vector of variables V_k, that contains the values of variables (input, intermediate, and output) just after execution of the first k instructions. We set $V_0 = X$. From the chain rule, and writing f' for the derivative (Jacobian matrix) of f, we get:

$$\begin{aligned} F'(X) &= (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(X)) \\ &\quad .\, (f'_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(X)) \\ &\quad .\, \ldots \\ &\quad .\, (f'_1(X)) \\ &= f'_p(V_{p-1}).f'_{p-1}(V_{p-2}) \ldots f'_1(V_0) \end{aligned} \tag{3}$$

Automatic Differentiation builds a program that computes these elementary Jacobian matrices $f'_k(V_{k-1})$, and combines them efficiently to get the desired derivatives.

2.6 The model of reverse A.D.

Let us now focus on the *reverse mode*. The goal is to compute a gradient, i.e. the product $\overline{X} = F'^t(X).\overline{Y}$. Using the above model (3), this writes:

$$\overline{X} = F'^t(X).\overline{Y} = f'^t_1(V_0) \ldots f'^t_{p-1}(V_{p-2}).f'^t_p(V_{p-1}).\overline{Y} \tag{4}$$

If the products are done from left to right, one multiplies matrices with matrices, ends up with the complete Jacobian matrix $F'^t(X)$, and finally multiplies it by $\overline{Y}$. This is very expensive in time and space, and should be avoided. In contrast, from right to left, one constantly multiplies a matrix with a vector, which is far less expensive. This is how the reverse mode works, and it is the key to its efficiency. The *reverse program* thus performs a sequence of (composite) instructions $\overline{I}_k, k \in [p..1]$, where each $\overline{I}_k$ operates on a vector of new variables $\overline{V}_k$, with same length as V_k. Vector $\overline{V}_p$ is initially set to $\overline{Y}$. Each $\overline{I}_k$, for $k = p$ down to 1, performs the matrix times vector product:

$$\overline{V}_{k-1} = f'^t_k(V_{k-1}).\overline{V}_k \tag{5}$$

The result $\overline{X}$ is eventually found in $\overline{V}_0$. There is an intuitive meaning to these $\overline{V}_k$: just like $\overline{X}$ is the gradient of the output with respect to X, each $\overline{V}_k$ is the gradient of the output with respect to the intermediate variables V_k. Yet in other words, $\overline{V}_k$ measures the *influence* of the intermediate V_k on the output objective function $(\overline{Y}|Y)$.

Notice that the model of the reverse mode implies that $f_p'^t(V_{p-1})$ is used before $f_{p-1}'^t(V_{p-2})$. Therefore, the values of the intermediate variables V_k are needed in the *reverse* order of their normal computation order. This inversion has a cost, that is the main drawback of the reverse mode. One extreme way to solve this inversion problem is to recompute each needed V_k from V_0, by restarting the program until instruction I_k. This is the basic tactic of the AD tool TAMC/TAF [7]. The cost is extra execution time, essentially proportional to the square of the number of run-time instructions p. Figure 1 summarizes this tactic graphically. Left-to-right arrows represent execution of original instructions I_k, right-to-left arrows represent the execution of the reverse instructions $\overline{I}_k$. The big dot represents the storage of all variables needed to restart execution from a given point, which is called a *snapshot*.

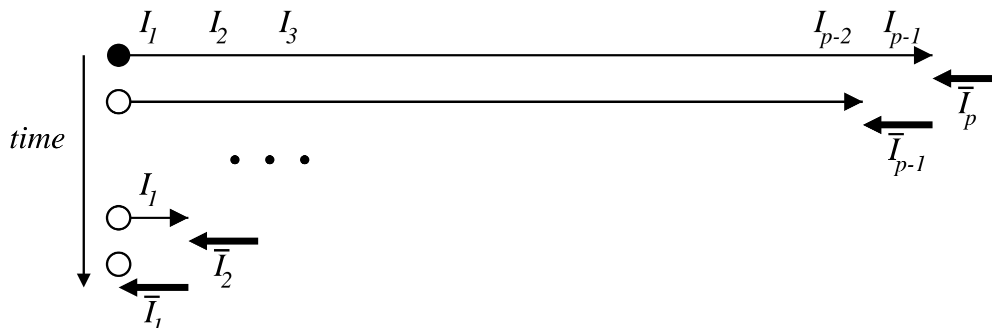

Figure 1: The "recompute-all" tactic

The other extreme solution is to store each V_k in memory, onto a stack, during a preliminary execution of the program P. Then the computation of the $f_k'^t(V_{k-1})$ pops the V_{k-1} from this stack upon demand. This is the basic tactic in ADIFOR [2] and TAPENADE. The cost is memory space, essentially proportional to the number of run-time instructions p. Figure 2 summarizes this tactic graphically. Vertical bars represent storage of the V_k on a stack, and on the way back their retrieval from this stack. Trade-offs exist between these two extremes. The idea is to select a part of the program, e.g. a subroutine, for which one accepts repeated execution. This costs one snapshot. In return, one spares a part of the memory needed for the storage of intermediate values. In AD jargon, this is called *checkpointing*. For example on figure 3, one can check that, at the cost of one repeated execution of part D,

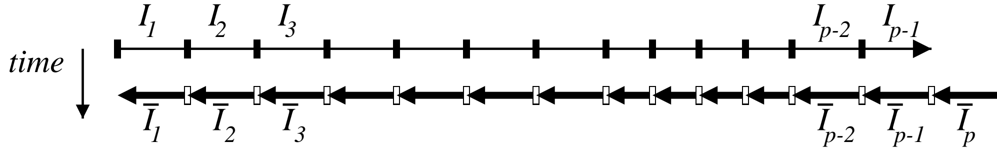

Figure 2: The "store-all" tactic

the maximum size of the stack is reduced of approximately one third. On

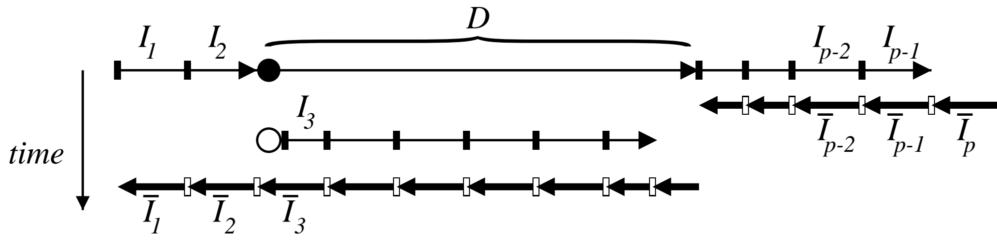

Figure 3: Checkpointing program part "D"

some model cases, for example a loop with a known number of iterations N, there exists a checkpointing strategy that can be proved optimal [9], where both the maximum number of recomputations and the maximum size of the stack grow only like $Log(N)$.

2.7 A small illustration example

Let us illustrate the application of the A.D. models to a small subroutine, that measures a sort of discrepancy between two given arrays `T` and `U`. The original subroutine is shown on the left column of figure 4.

- The **tangent mode** produces the right column of figure 4. Each original instruction is preceded by its differentiated instruction, which computes tangent derivatives, conventionally shown with a *dot* above. For example, the differentiated instruction that precedes the last original instruction `e = SQRT(e2)`, implements the following vector assignment that multiplies the instruction's elementary Jacobian by the vector of tangent derivatives:

$$\begin{bmatrix} \dot{\mathtt{e2}} \\ \dot{\mathtt{e}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0.5/\sqrt{\mathtt{e2}} & 0 \end{bmatrix} . \begin{bmatrix} \dot{\mathtt{e2}} \\ \dot{\mathtt{e}} \end{bmatrix}$$

original: T,U ↦ e	**tangent mode:** T,Ṫ,U,U̇ ↦ e,ė
	ė2 = 0.0
e2 = 0.0	e2 = 0.0
do i=1,n	do i=1,n
	ė1 = Ṫ(i)-U̇(i)
e1 = T(i)-U(i)	e1 = T(i)-U(i)
	ė2 = ė2 + 2.0*e1*ė1
e2 = e2 + e1*e1	e2 = e2 + e1*e1
end do	end do
	ė = 0.5*ė2/SQRT(e2)
e = SQRT(e2)	e = SQRT(e2)

Figure 4: A.D. in tangent mode

- The **reverse mode** produces the program on figure 5. It computes reverse derivatives, conventionally shown with a *bar* above, using the *store-all* tactic of figure 2. Therefore the differentiated program is made of two successive parts, known as the *forward* and *backward sweeps*. The forward sweep is a copy of the original program, plus instructions to store the intermediate values of variables. The backward sweep, after initialization of local reverse derivatives $\overline{\texttt{e1}}$ and $\overline{\texttt{e2}}$, evaluates equation (5), for each original instruction, but in the reverse order ($k = p$ down to 1). Thus the `do` loop now runs from `i=n` down to `1`. Considering again instruction `e = SQRT(e2)`, differentiation produces the following vector assignment that multiplies the instruction's *transposed* elementary Jacobian by the vector of reverse derivatives:

$$\begin{bmatrix} \overline{\texttt{e2}} \\ \overline{\texttt{e}} \end{bmatrix} = \begin{bmatrix} 1 & 0.5/\sqrt{\texttt{e2}} \\ 0 & 0 \end{bmatrix} . \begin{bmatrix} \overline{\texttt{e2}} \\ \overline{\texttt{e}} \end{bmatrix}$$

  This takes two derivative instructions, because $\overline{\texttt{e}}$ must be reset to zero. Notice furthermore the calls to `PUSH` and `POP`, that store and retrieve the intermediate values of variables `e1` and `e2`: in practice, not *all* values need be stored before each instruction (cf section 3.2). Only the value(s) that are going to be overwritten need be stored.

3 The TAPENADE AD tool

TAPENADE is an Automatic Differentiation tool, based on the models of sections 2.5 and 2.6. Given the source of an original program, plus a description of which output variables must be differentiated, and with respect

reverse mode: `T,U,`$\overline{\texttt{e}}$ $\mapsto$ $\overline{\texttt{T}}$`,`$\overline{\texttt{U}}$	
forward sweep:	*backward sweep:*
`e2 = 0.0`	$\overline{\texttt{e2}}$ `= 0.0`
`do i=1,n`	$\overline{\texttt{e1}}$ `= 0.0`
`PUSH(e1)`	$\overline{\texttt{e2}}$ `=` $\overline{\texttt{e2}}$ `+ 0.5*`$\overline{\texttt{e}}$`/SQRT(e2)`
`e1 = T(i)-U(i)`	$\overline{\texttt{e}}$ `= 0.0`
`PUSH(e2)`	`do i=n,1,-1`
`e2 = e2 + e1*e1`	`POP(e2)`
`end do`	$\overline{\texttt{e1}}$ `=` $\overline{\texttt{e1}}$ `+ 2*e1*`$\overline{\texttt{e2}}$
`e = SQRT(e2)`	`POP(e1)`
	$\overline{\texttt{T}}$`(i) =` $\overline{\texttt{T}}$`(i) +` $\overline{\texttt{e1}}$
	$\overline{\texttt{U}}$`(i) =` $\overline{\texttt{U}}$`(i) -` $\overline{\texttt{e1}}$
	$\overline{\texttt{e1}}$ `= 0.0`
	`end do`
	$\overline{\texttt{e2}}$ `= 0.0`

Figure 5: A.D. in reverse mode

to which input variables, TAPENADE produces a new source program that computes the requested derivatives. Figure 6 shows the overall architecture of TAPENADE. One motivation is independence from the language of the given programs. To this end, we designed an internal Intermediate Language (IL), that contains all the syntactic elements of usual imperative languages, FORTRAN, C,... Sources in FORTRAN are first translated into IL, then sent to TAPENADE for differentiation, and the result is translated from IL back into FORTRAN. Another motivation is that many analyses (type-checking, in-out...) are not specific to AD. Therefore they are gathered into a separate layer: an Imperative Language Analyzer. Above are specialized layers, such as the present AD engine. TAPENADE is implemented mostly in JAVA.

Since AD tools are large systems and require complex static analyses, it was of highest importance to choose a powerful internal representation of programs. Therefore, in complement to Abstract Syntax Trees, TAPENADE uses standard models from compiler theory [1]. Each program is kept internally as a Call Graph. Each node of the Call Graph, representing one subroutine, is kept as a Flow Graph (*cf* figure 9). Information about types, variables, and other declared objects, is kept in Symbol Tables, which are nested to support scoping. The nodes of the Flow Graph, called Basic Blocks, contain lists of instructions, each one kept as an IL Abstract Syntax Tree. All static analyses run on Flow Graphs, and most of them are fixed point iterations so that TAPENADE accepts any flow of control, even unstructured.

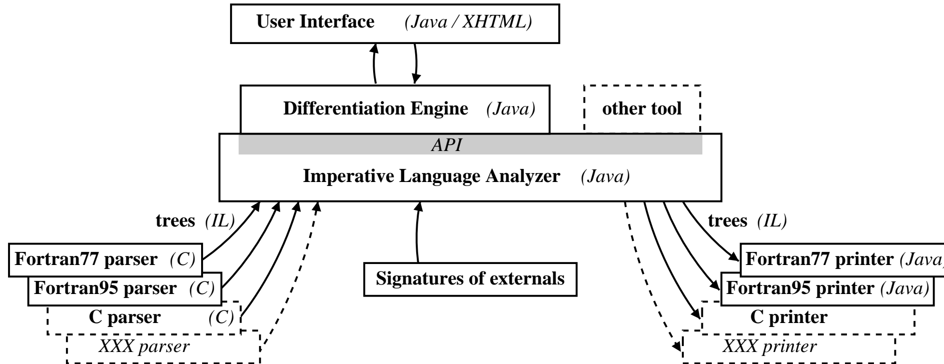

Figure 6: Overall architecture of TAPENADE

Flow Graphs permit remarkable efficiency of these static analyses, because a lot of constant work can be precomputed at the level of each Basic Block.

The version 2.0 of TAPENADE provides 3 modes of differentiation: the **tangent** mode, the **reverse** mode, and a **multi-directional tangent** mode. This last mode computes the directional tangent derivatives, but in many directions simultaneously. It is therefore equivalent to running the tangent mode repeatedly, but the evaluation of the original function is done only once. TAPENADE 2.0 accepts languages FORTRAN77 and a subset of FORTRAN95. The version for C is still under development.

There are other differentiation modes that TAPENADE does not provide, which exist in other tools. In particular, there is no mode that explicitly computes the Jacobian Matrix, although this can be achieved with a clever use of the multi-directional tangent mode. There is nothing about higher-order derivatives. To retrieve intermediate values in the reverse mode, only the "store-all" tactic of figure 2, and the checkpointing scheme of figure 3 are available, but the "recompute-all" tactic of figure 1 is not supported.

The next subsections focus on the most important tactics and algorithms in TAPENADE, and especially those for the reverse mode.

3.1 Activity analysis

TAPENADE, like most other AD tools, allows the end-user to restrict differentiation to the derivatives of *some* of the output variables Y, called the *dependent* Y_D, with respect to *some* of the input variables X, called the *independent* X_I. This information is propagated inside the program and as a result, many derivative variables can be erased from the differentiated program. Let us say that a variable instance v_j *depends in a differentiable*

way, or *depends*, on v_k, and we write $v_k \prec^* v_j$, iff the partial derivative of v_j with respect to v_k is defined and not trivially null. Obviously, relation $\prec^*$ is the transitive closure of relation $\prec$, that states the dependency between the arguments of arithmetic operations and their result. We say that a variable instance v is *varied* iff $\exists x \in X_I : x \prec^* v$, and that v is *useful* iff $\exists y \in Y_D : v \prec^* y$. Only variable instances v that are *active*, i.e. both varied and useful, need to have an associated derivative (named $\dot{v}$ in tangent mode or $\bar{v}$ in reverse mode). Here is why:

- In tangent mode, if v is not varied, then $\dot{v} = 0$ and all occurrences of $\dot{v}$ can be explicitly replaced by zero. If v is not useful, then we are sure no use of $\dot{v}$ will occur later, and therefore $\dot{v}$ need not be computed.
- In reverse mode, if v is not varied, then the value of $\bar{v}$ will not be used in the reverse instructions that follow, and therefore need not be computed. If v is not useful, then the value of v does not influence the outputs Y_D, therefore $\bar{v} = 0$, and all occurrences of $\bar{v}$ can be explicitly replaced by zero.

In AD jargon, *activity analysis* is the analysis that detects active instances of variables. Activity analysis is the most time-consuming analysis in TAPENADE. To prevent combinatorial explosion, it is done in two sweeps on the Call Graph. The first sweep, bottom-up on the Call Graph, computes explicitly relation $\prec^*$ between the inputs and the outputs of each subroutine. The second sweep, top-down on the Call Graph, propagates varied and useful variables through each subroutine. Initially, for the top routine, these are set respectively to X_I and Y_D. At the end, for each point in the program, i.e. between each two successive instructions, we know which variables are active.

On the example of figure 5, activity analysis tells us that `e2` is not active at the beginning of the original program, because it is a local variable. Therefore it is no use to reset $\overline{\texttt{e2}}$ to zero at the end of the backward sweep. Similarly, `e1` is inactive just before instruction `e1 = T(i)-U(i)`, because it is going to be overwritten. Therefore it is no use to reset $\overline{\texttt{e1}}$ to zero, and $\overline{\texttt{e1}}$ is certainly always equal to zero before instruction $\overline{\texttt{e1}}$ `=` $\overline{\texttt{e1}}$ `+ 2*e1*`$\overline{\texttt{e2}}$, which can therefore be simplified.

The usual remarks about static analyses apply here. First, it often happens that some variable cannot be proved certainly active or not. This is the classical undecidability problem. In this case, we make the *conservative assumption* that the variable *is* active. Second, it may be profitable, in some cases, to distinguish each element of arrays, because some may be active, and some not. There is some fundamental research going on about this so-called

array region analysis [5]. The current version of TAPENADE lacks this array region analysis. Last, we made the choice of *generalization* versus *specialization*. This means that when a given subroutine is called many times, in many different analysis contexts, all these contexts are fused into a single, generalized one. The activity analysis of this subroutine is therefore done only once, for the generalized context. This ensures that the analysis time remains proportional to the program's length, at the expense of a possible loss in precision.

3.2 TBR: a specific analysis for the reverse mode

The main drawback of the reverse mode is the need to *restore* the intermediate values in the reverse of their normal computation order. Whatever the method used, this has a cost. For the "store-all" tactic, with or without checkpointing, this means storing an retrieving values using a stack. But we observe that not all values need be stored before each instruction.

First, consider a variable `x`. At some time during execution of the original program, `x` is assigned. Then during a number of instructions, `x` is not modified, only used occasionally. And finally either it is overwritten, or it reaches the end of the program unchanged. It is not necessary to store `x` before each instruction: during the *forward sweep*, `x` need be stored only just before instructions that overwrite it. Thus, during the backward sweep, the old value `x` is popped before the derivative of the instructions that overwrite `x`, and `x` holds the correct value for the derivatives of all instructions before, until one reaches another instruction that overwrites `x`. This tactic was already applied on the example of figure 5. This explains why only `e1` and `e2` are pushed on the stack. Variables `e2` and `e` are not stored, because their value before assignment is undefined.

We can refine this further. Consider again this variable `x`. Before two successive assignments to `x`, there are a number of instructions that possibly use `x`. If `x` is used in a linear expression, like in `z = 2*x + y`, we know that `x` does not appear in the elementary Jacobian matrix of this instruction. In other words, `x` need not be restored for the derivative of this instruction. Therefore, we defined the "*To Be Restored*" (TBR) static analysis [6]: for each variable `x`, before each instruction I that overwrites `x`, TBR detects whether the value of `x` is needed by the derivative of some instruction caught between the previous overwriting of `x` and I. The value of `x` is stored before I only if this TBR status is true. On the example of figure 5, TBR analysis tells us that between two successive assignments to `e2`, `e2` is not used by the derivative instructions. Therefore, instructions `PUSH(e2)` and `POP(e2)` are not necessary.

3.3 Data-dependency analysis

TAPENADE implements data-dependency analysis to improve the derivative program. In the reverse mode, all instructions of the backward sweep are created as nodes of a data-dependency graph. It turns out that a large majority of reverse differentiated instructions either set differential variables to zero, or increment these differential variables. In many cases, data-dependency analysis allows TAPENADE to safely merge these kinds of differentiated instructions into a single instruction. This improves data locality. Although compilers usually can do this kind of improvement between instructions not too far apart ("peep-hole" optimization), we believe that doing it at the differentiation level is more systematic and powerful.

On the example of figure 5, data-dependency analysis tells us that instructions $\overline{\texttt{e2}}\texttt{ = 0.0}$ and $\overline{\texttt{e2}}\texttt{ = }\overline{\texttt{e2}}\texttt{ + 0.5*}\overline{\texttt{e}}\texttt{/SQRT(e2)}$ can be merged.

Using activity analysis, TBR analysis, and data-dependency analysis, the reverse differentiated program of figure 5 boils down to figure 7, which is what TAPENADE produces.

reverse mode: $\texttt{T,U,}\overline{\texttt{e}} \mapsto \overline{\texttt{T}}\texttt{,}\overline{\texttt{U}}$	
forward sweep:	*backward sweep:*
`e2 = 0.0`	$\overline{\texttt{e2}}\texttt{ = 0.5*}\overline{\texttt{e}}\texttt{/SQRT(e2)}$
`do i=1,n`	$\overline{\texttt{e}}\texttt{ = 0.0}$
`PUSH(e1)`	`do i=n,1,-1`
`e1 = T(i)-U(i)`	$\overline{\texttt{e1}}\texttt{ = 2*e1*}\overline{\texttt{e2}}$
`e2 = e2 + e1*e1`	`POP(e1)`
`end do`	$\overline{\texttt{T}}\texttt{(i) = }\overline{\texttt{T}}\texttt{(i) + }\overline{\texttt{e1}}$
`e = SQRT(e2)`	$\overline{\texttt{U}}\texttt{(i) = }\overline{\texttt{U}}\texttt{(i) - }\overline{\texttt{e1}}$
	`end do`

Figure 7: Reverse differentiation by TAPENADE of the example on figure 4

3.4 Control inversion in the reverse mode

Derivative instructions must be executed in an order that mimics the original program's. This is straightforward in **tangent** mode, **multi-directional tangent** mode, and also in the forward sweep of the **reverse mode**, since we just keep the original program's control structure. Things are not so easy for the backward sweep of the **reverse mode**, because the control, which may contain `goto` jumps, `while` loops, etc, is reversed. Even for well structured conditionals, we must devise the strategy carefully: simply duplicating the

original test does not work, because variables in the test expressions might need to be stored, and to preserve the stack structure, they must be stored at the *exit* of the conditional. Unfortunately, the values we need to store are the values at the *beginning* of the conditional.

Therefore, the correct strategy is to store, upon *exit* from the conditional, the branch that *was* actually chosen (as an integer). This is best defined on the Flow Graph: each time more than one flow arrows converge on a Basic Block, the forward sweep must remember which arriving arrow was taken, and the backward sweep must use this to decide where to go back to. Specialized reversal patterns also exist for loops. This strategy generates a backward sweep which is as much structured as possible, and makes the reverse code both more readable and better compiled. Figure 8 shows an example, where the original program contains a `while` loop and a jump outside a test. We added blank lines to emphasize correspondence between the original program and the differentiated program. Figure 9 shows on the left the corresponding Flow Graph, emphasizing the places where TAPENADE decides to store the control information, and on the right the reversed Flow Graph, emphasizing the places where control information is retrieved and used. In the loop body, there are two places where the control flow converges: at the end of the conditional, and at label `5`. At each such place, converging arrows are numbered arbitrarily (*0, 1, ...*), and this number is stored forwards and used backwards. Also, since the while loop is dynamic, TAPENADE inserts a loop `counter`, and stores successive values of i.

3.5 Checkpointing on calls using In-Out analysis

Checkpointing is absolutely necessary on real programs, because the number of values that must be stored by a "store-all" tactic often exceeds the largest available disk space. Generally speaking, checkpointing (*cf* figure 3) means designating one (or many) part "D" of the original program, and executing "D" twice, the first time without storage of intermediate values, and the second time with storage. This part "D" may be a subroutine call, like in TAPENADE, or some selected iterations of a loop, like in [9].

To implement checkpointing, we must take a *snapshot* of all variables needed to restart execution of "D". This is where In-Out analysis comes in. In-Out analysis is a classical static analysis that determines, for a given part P of a program, which entry values of variables are used in P (*"In"*), and which variables are overwritten in P (*"Out"*). Clearly, only variables which are *In* for part "D" need to be taken in the snapshot for "D". Figure 10 shows a small example. The original program calls function `G`. Suppose we know, either from the analysis of the source of `G` or from user-given information if

original: T,U $\mapsto$ T,U	**reverse mode:** T,$\overline{\mathtt{T}}$,U,$\overline{\mathtt{U}}$ $\mapsto$ $\overline{\mathtt{T}}$,$\overline{\mathtt{U}}$	
	forward sweep:	*backward sweep:*
	counter=0	POP(counter)
do while(T(i)>0.0)	do while (T(i)>0.0)	do i0=1,counter
if (T(i)>1.0) then	if (T(i)>1.0) then	POP(T(i))
U(i)=U(i)+log(T(i))	U(i)=U(i)+log(T(i))	$\overline{\mathtt{T}}$(i)=2*$\overline{\mathtt{T}}$(i)
	PUSH(0)	POP(i)
else	else	POP(arrow)
U(i)=U(i)+T(i-5)	U(i)=U(i)+T(i-5)	if (arrow=1) then
if(U(i)<0.0) goto 5	if (U(i)<0.0) then	POP(T(i))
	PUSH(0)	$\overline{\mathtt{U}}$(i)=$\overline{\mathtt{U}}$(i)+3*$\overline{\mathtt{T}}$(i)
	goto 5	$\overline{\mathtt{T}}$(i)=0.0
	endif	POP(arrow)
	PUSH(1)	if (arrow=0) then
endif	endif	$\overline{\mathtt{T}}$(i)=$\overline{\mathtt{T}}$(i)+$\overline{\mathtt{U}}$(i)/T(i)
	PUSH(T(i))	goto 6
T(i)=3*U(i)	T(i)=3*U(i)	endif
	PUSH(1)	endif
5 i=i+5	5 PUSH(i)	$\overline{\mathtt{T}}$(i-5)=$\overline{\mathtt{T}}$(i-5)+$\overline{\mathtt{U}}$(i)
	i=i+5	6 continue
T(i)=2*T(i)+1	PUSH(T(i))	
	T(i)=2*T(i)+1	
	counter=counter+1	
enddo	enddo	enddo
	PUSH(counter)	

Figure 8: TAPENADE reverse differentiation on complex control

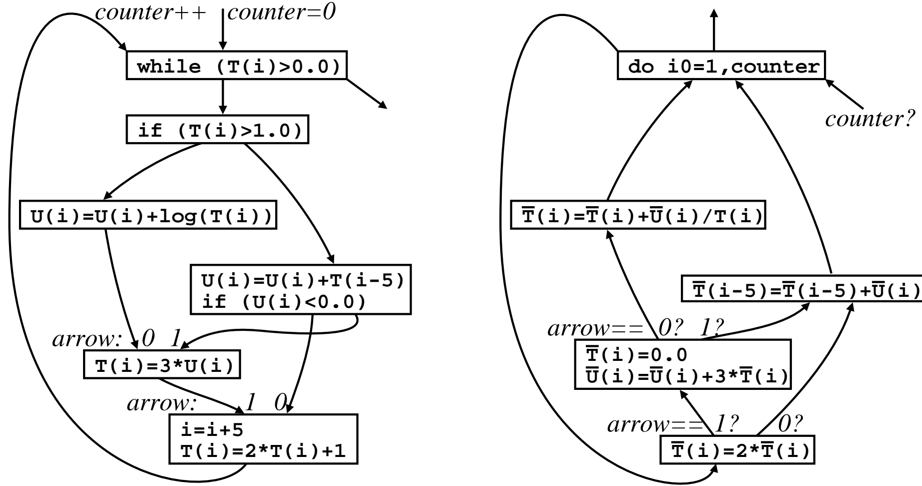

Figure 9: Control reversal on the Flow Graph

this source is hidden (see section 4.2), that `G` only uses its 1st and 2nd arguments and some global variable `x`, and overwrites its 1st and 3rd argument. Suppose also, to make things a little bit more complex, that `G`, although called with a scalar 2nd argument `T(k)`, actually expects a 2nd argument of dimension 7. This is a widespread practice in Fortran77, to emulate pointers. Then the forward sweep takes a snapshot of the *In* arguments of `G`: `x`, `a`, and the 7 elements of array `T` from rank `k`. This requires that `k` be taken in the snapshot too. During the backward sweep, the snapshot is used to restore the necessary values, before the call to the differentiated routine $\overline{\mathtt{G}}$. $\overline{\mathtt{G}}$ performs the forward sweep for `G`, immediately followed by the corresponding backward sweep. As a result, execution follows the pattern of Figure 3 with "D" defined to be `G`.

Notice that snapshots can be reduced further: when a variable is *In* for "D", and not *Out* from the beginning of "D" until the end of the program (or in the case of nested checkpoints, until the end of the surrounding checkpointing part). Then the value of the variable is never modified between the first and second execution of "D", and therefore it need not be taken by the snapshot. This refinement is not yet implemented in TAPENADE.

4 Practical utilization of TAPENADE

There are basically two ways to use TAPENADE. One can install a copy on the local computer, and then call it with a simple command, possibly from a Makefile, just like a compiler. Options are available to specify the head

original:	**reverse mode:** $a,\overline{a},T,\overline{T},x,\overline{x},\overline{r},\overline{r2} \mapsto \overline{a},\overline{T},\overline{x},\overline{r},\overline{r2}$	
$a,T,x \mapsto a,r,r2$	*forward sweep:*	*backward sweep:*
...	...	...
a = a + 3*T(2)	a = a + 3 * T(2)	POP(k)
r = G(a,T(k),r2)	PUSH(x)	POP(a)
...	PUSH(T(k:k+6))	POP(T(k:k+6))
	PUSH(a)	POP(x)
	PUSH(k)	$\overline{G}(a,\overline{a},T(k),\overline{T}(k),r2,\overline{r2},\overline{r})$
	r = G(a,T(k),r2)	$\overline{r}$ = 0.0
	...	$\overline{r2}$ = 0.0
		$\overline{T}(2) = \overline{T}(2) + 3 * \overline{a}$
		...

Figure 10: TAPENADE reverse differentiation of a function call

routine to differentiate, the dependent and independent variables, and other useful parametrizations. TAPENADE creates new files that contain the differentiated program. A typical command line is:

```
$> tapenade -reverse -head func -vars "x z" file1.f file2.f
```

Alternatively, one can use the TAPENADE web server that we maintain, at the URL: `http://tapenade.inria.fr:8080/tapenade/index.jsp`. In a few simple steps, one can upload a file and specify which differentiation is wanted. Then the results are displayed and can be downloaded. Detailed documentation on TAPENADE, in particular a tutorial, is available on our team's web site: `http://www-sop.inria.fr/tropics/`.

There is a graphical interface to display the results of differentiation. This interface comes systematically when using the server. It is also available as an option of the command-line invocation. The graphical interface consists of a set of HTML pages, and can be displayed by most web browsers. This interface uses HTML links to implement a (rudimentary) line-to-line correspondence between the original program and the differentiated program. Figure 11 shows this graphical interface on the example of figure 8. In the reverse program in the lower-right frame, the "`b`" suffix is used to replace the bar above differential variables.

4.1 Messages from TAPENADE analyses

During the execution of TAPENADE, several analysis modules might detect problems in the source files, such as type conflicts, wrong number of argu-

```
sub2
DO WHILE (t(i) .GT. 0.0)
  IF (t(i) .GT. 1.0) THEN
    u(i) = u(i) + LOG(t(i))
  ELSE
    u(i) = u(i) + t(i-5)
    IF (u(i) .LT. 0.0) GOTO 5
  END IF
  t(i) = 3 * u(i)
  i = i + 5
  t(i) = 2 * t(i) + 1
ENDDO
```

```
sub2_b
ENDDO
CALL PUSHINTEGER4(adCount)
CALL POPINTEGER4(adCount)
DO i0=1,adCount
  CALL POPREAL8(t(i))
  tb(i) = 2 * tb(i)
  CALL POPINTEGER4(i)
  CALL POPINTEGER4(branch)
  IF (.NOT.branch .LT. 1) THEN
    CALL POPREAL8(t(i))
    ub(i) = ub(i) + 3 * tb(i)
```

Figure 11: TAPENADE graphical interface

ments, aliasing, or variables used before initialized. All these messages are collected and displayed in the user interface, with HTML links to the corresponding places in the source.

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. For example, TAPENADE implements a limited version of detection of *aliasing*. Aliasing is when two different arguments of a given subroutine may in fact be the same memory location. Aliasing is in general a dangerous programming practice, but it may very well cause no harm to the original program. Nevertheless, this aliasing may often introduce errors into the program differentiated in reverse mode.

In some cases, TAPENADE needs some help from the user. Suppose for example that an array T must be differentiated, and the differentiated array $\dot{T}$ (or $\overline{T}$) must be declared as a local variable. If T itself is a formal argument, it may be of unknown size, or even of dynamic size. FORTRAN77 does not permit declaration of local $\dot{T}$ of dynamic size. There is no systematic solution to this very practical problem. Therefore, TAPENADE issues a message to the end-user, which requests for hand-modification of the generated program.

4.2 External routines

When a program uses "black-box" routines, e.g. libraries or routines with source hidden for any reason, TAPENADE makes some worst-case assumptions to differentiate the program safely. For example, it assumes that any argument of an external subroutine are inputs *and* outputs, and that the Jacobian matrix of this subroutine is full, i.e. each output depends on all inputs. This has negative effects on the performance of the generated code.

The user can improve this, by providing information on the hidden rou-

tines. TAPENADE defines a concise notation to put this information into a special specification file. Consider for example the function G of section 3.5. The specification file first enumerates the formal arguments expected by G, including globals, which we call the *shape* of G. Then, following this shape, the following information may be given: argument type, argument read or not, argument written or not, or dependency matrix, which is the sparsity pattern of the Jacobian matrix. For G, this is specified as:

```
function G: external :
    shape:  (param 1, param 2, param 3, result, common /globals/x)
    type:  (real, real(1:7), real, real, real)
    R: (1 1 0 0 1)
    W: (1 0 1 1 0)
    deps:( 0 0 0 0 1
           0 1 0 0 0
           1 0 0 0 1
           1 1 0 0 0
           0 0 0 0 1)
```

In any case, it is the responsibility of the user to provide the code for the differentiated routine.

5 Perspectives and conclusion

We have presented the AD tool TAPENADE. Apart from the fundamental *tangent* mode, the strong point of TAPENADE is the *reverse* mode, which computes gradients. There is also a more experimental *multi-directional* tangent mode, that computes simultaneously many tangent modes, following different directions in the input space. The novelty brought by TAPENADE in this 3rd mode is the use of data-dependency analysis to enable automatic fusion of loops on differentiation directions.

In this presentation, we put particular emphasis on the reverse mode, because gradients are the discrete equivalent of the *adjoint* equations found in inverse problems and optimization. We tried to describe the underlying model of reverse AD, and showed how it implies a particular architecture of reverse-differentiated programs. We also described in some detail the concrete and practical techniques implemented in TAPENADE to improve these reverse-differentiated programs. With these techniques in mind, it should be easier to understand a program differentiated in the reverse mode, and therefore one can grow more confident in AD tools such as TAPENADE.

The degree of maturity of AD is hard to assess. Probably a good measure is its usage in the applications community, which is steadily growing. Applications in meteorology, oceanography [11], are frequent. Large applications in optimal shape design also start to appear [12] [4]. With TAPENADE, we have differentiated codes up to 130,000 lines in tangent mode, or up to 10,000 lines in reverse mode. Obviously, maturity of the reverse mode still lags behind the tangent mode. This comes of course from the difficulty of storing/recomputing the intermediate values. On large programs, checkpointing is the only solution to this storage problem, and this checkpointing needs to be optimized, largely by hand. There is still research work ahead, to bridge the gap from optimal checkpointing strategies on special cases, to good automatic checkpointing strategies on arbitrary programs.

Utilization of AD largely depends on the degree of confidence of users. This confidence is good enough for the *tangent* mode, because the underlying model is simple and the differentiated program is still understandable. On the other hand, confidence in the reverse mode still needs to be improved. Generated programs are harder to read, even if we believe TAPENADE makes a step in this direction. Confidence will also improve when differentiated programs can detect the cases when the returned derivatives are *not* valid, or valid in a small domain which is limited by control. We believe this is an interesting direction for research. Practically, validation and debug of differentiated programs is still a difficult task. A set of tools to facilitate debugging would be helpful. For example we developed a systematic "on the fly" comparison of the tangent mode with divided differences, along with program's execution. We also try to build debugging tools for the reverse mode.

In the years to come, TAPENADE will be progressively extended to accept more and more of FORTRAN95, and also C. Right from the beginning, the architecture of TAPENADE was devised to ease this. Another perspective is to open the programming interface between the general-purpose analyses and the AD tool, to let other tools plug in at this level.

References

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

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

[3] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann(editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization.* Springer, 2001. Selected proceedings of AD2000, Nice, France.

[4] F. Courty, A. Dervieux, B. Koobus, and L. Hascoët. Reverse Automatic Differentiation for Optimum Design: from Adjoint State assembly to Gradient Computation. *submitted*, 2002.

[5] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.

[6] C. Faure and U. Naumann. Minimizing the Tape Size. *in [3]*, pages 293–298, 2000.

[7] R. Giering. Tangent linear and Adjoint Model Compiler, Users manual. Technical report, 1997. http://www.autodiff.com/tamc.

[8] J.C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

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

[10] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Frontiers in Applied Mathematics, 2000.

[11] P. Heimbach, C. Hill, and R. Giering. Automatic Generation of Efficient Adjoint Code for a Parallel Navier-Stokes Solver. In *International Conference on Computational Science (Part II), Amsterdam, The Netherlands.* Springer, 2002.

[12] P. Hovland, B. Mohammadi, and C. Bischof. Automatic Differentiation of Navier-Stokes computations. Technical report, Argonne National Laboratory MCS-P687-0997, 1997.