European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS 2004
P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer (eds.)
Jyväskylä, 24–28 July 2004

TAPENADE: A TOOL FOR AUTOMATIC DIFFERENTIATION OF PROGRAMS

Laurent Hascoët*

*INRIA Sophia-Antipolis, TROPICS team
2004 Route des lucioles, BP 93, 06902 VALBONNE, FRANCE
e-mail: Laurent.Hascoet@sophia.inria.fr

Key words: Automatic Differentiation, Adjoint code, Adjoint algorithm, Gradient, Optimization

Abstract. *We present* TAPENADE*, a tool for Automatic Differentiation (AD). AD transforms a program that computes or simulates a mathematical vector function into a new program that computes derivatives of this function. Specifically,* TAPENADE *can produce* tangent *programs that compute directional derivatives, and* adjoint *programs that compute gradients. Gradients and Adjoints are probably the most promising derivatives, as they are required in optimization. Therefore, they receive a particular attention and development effort in* TAPENADE*. This paper presents the AD principles behind* TAPENADE*, and shows how they are reflected in* TAPENADE*'s differentiation model, internal algorithms and output. We show the program analysis techniques that make AD-generated tangents or adjoints perform comparably to good hand-written tangents or adjoints, at a cheaper development cost.* TAPENADE *is available at no cost, either as a server on our web site* `http://www-sop.inria.fr/tropics`*, or downloaded locally and called from the command line or from a makefile.*

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 is in the latest collection of articles [3] and in the monograph [8].

AD is a program transformation, and is therefore performed by software tools similar to compilers or parallelizers. This article presents TAPENADE, an AD tool with a strong focus on the "*reverse*" mode, that computes *gradients*. Our guideline in this work 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.

After a brief description in section 2 of the theoretical basis of AD, section 3 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 4 focuses on refinements to the AD model for adjoint codes. Section 5 presents the user interface of TAPENADE, and gives pointers to further documentation. Section 6 concludes with next developments to come in TAPENADE.

2 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 $X \in \mathbb{R}^n$, computes some vector function $Y = F(X) \in \mathbb{R}^m$. The AD tool generates a new source program that, given the argument X, computes some derivatives of F. In short, AD first 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 reproduce 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. Thus, for a given control:

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

where each f_k is the elementary function implemented by instruction I_k. 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

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

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

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 $m \times n$. Moreover, some problems are solved using only some projections of $F'(X)$. For example, one may need only *sensitivities*, which are $F'(X).\dot{X}$ for a given direction $\dot{X}$ in the input space. Using equation (2), sensitivity is

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

which is easily computed from right to left, 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.

However in optimization, data assimilation [10], inverse problems, or adjoint problems [7], the appropriate derivative is the *gradient* $F'^*(X).\overline{Y}$, where F'^* is the *transposed* Jacobian. Using equation (2), the gradient writes

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

which is most efficiently computed 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.

This turns out to make a very efficient program, at least theoretically [8, Section 3.4]. The computation time required for the gradient is only a small multiple of the run time of P, multiplied by the number of outputs m, which is usually small for optimization or inverse problems. It is independent from the number of parameters n, which can be very large.

However, we observe 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 $f'^*_{k+1}(X_k)$. There are two strategies for that:

- **Recompute All (RA):** the X_k is recomputed when needed, restarting P on input X_0 until instruction I_k. Brute-force RA strategy has a quadratic time cost with respect to the total number of run-time instructions p. The TAF [6] tool uses this strategy, together with *checkpointing* to reduce its time complexity.

- **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. Brute-force SA strategy has a linear memory cost with respect to p. The ADIFOR [2] and TAPENADE tools use this strategy.

Practically, both RA and SA strategies need a special storage/recomputation trade-off in order to be really profitable, and this makes them become very similar. This trade-off is

called *checkpointing*. Since TAPENADE uses the SA strategy and applies checkpointing to procedure calls, we will describe checkpointing in this context.

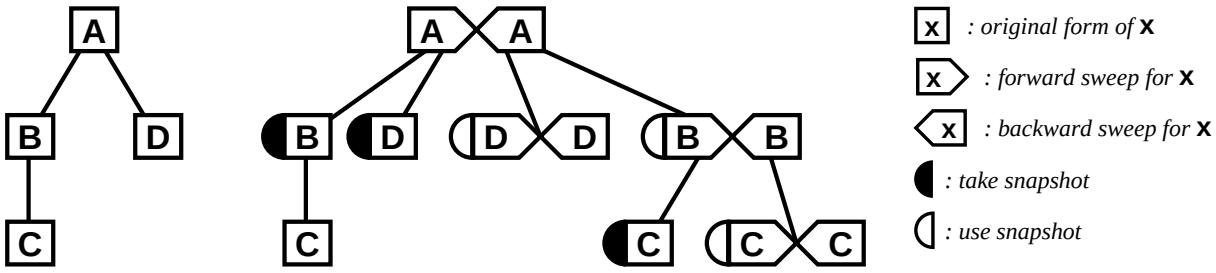

Figure 1: Reverse AD storage/recomputation tradeoff on the Call Tree

Let us define some vocabulary and graphical notations. Execution of a subroutine `A` in its original form is shown as [A]. Execution of `A` augmented with storage of variables on the stack, just before they are overwritten, is called the "*forward sweep*", shown as [A>. Actual computation of the gradient of `A`, that pops values from the stack when they are needed to restore the X_k's, is called the "*backward sweep*", shown as <A]. With no checkpointing, plain reverse differentiation of `A` is just [A><A]. Checkpointing consists in choosing a part `B` (a procedure call in TAPENADE) in `A`, which will be run *without* storage during [A>. When the backward sweep <A] reaches `B`, it runs `B` again, this time with storage, followed by the backward sweep of `B` and the rest of <A]. Duplicate execution of `B` requires that some variables used by `B` (a "*snapshot*") be stored. In TAPENADE, checkpointing is applied at each procedure call. Figure 1 shows the resulting differentiated call tree for an example initial program call tree. If the program's call tree is well balanced, 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 THE DIFFERENTIATION MODEL OF TAPENADE

The previous section showed the theoretical basis of Automatic Differentiation, emphasizing the reverse mode. 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.

3.1 Symbol names

First consider symbol names. If a variable `v` is of differentiable type, and currently has a non-trivial derivative (see *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 procedures and `COMMONS` are built appending "`_D`" in *tangent* mode and "`_B`" in reverse mode. The following table summarizes that:

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

`SUBROUTINE T1(a)`
`REAL a(10)`
`REAL b(5)`
	`SUBROUTINE T1_D(a,ad)`
	`REAL a(10),ad(10)`
	`REAL b(5),bd(5)`
		`SUBROUTINE T1_B(a,ab)`
		`REAL a(10),ab(10)`
		`REAL b(5),bb(5)`

TAPENADE checks for possible conflicts with names already used in the program, in which case it appends `0`, then `1`, etc after the derivative name until conflicts disappear. Suffixes can be changed via command line options.

3.2 Simple instructions

Now consider an assignment I_k. In *tangent* mode (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 (equation (4)), derivative instruction(s) $\overline{I}_k$ implement $\overline{Y}_{k-1} = f'^*_k(X_{k-1}).\overline{Y}_k$, with initial $\overline{Y}_p = \overline{Y}$. 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))`,

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

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

and therefore TAPENADE produces the following derivative instructions:

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)`

Other simple instructions may have side-effects that affect derivatives. For example a `READ` I-O into a variable `v` forces the derivative of `v` to be reset to zero. TAPENADE automatically inserts these reset instructions. However, the end-user should check that this is the behavior wanted.

3.3 Activity of variables

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. A variable is said "*active*" if it depends on some independent *and* some dependent depends on it. Only the derivatives of the active variables need be computed. If variable `v` depends on no independent, then `vd` is certainly null and the value of `vb` does not matter. Conversely, if no dependent depends on `v`, then the value of `vd` does not matter, and `vb` is certainly null. TAPENADE automatically detects active variables and simplifies the differentiated program accordingly.

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`

In this example, `x` does not depend any more on the independent, and `t` has no influence on any dependent. Therefore, TAPENADE knows that `xd` and `tb` are null: they can be simplified and never computed. We shall say that these derivatives are *implicit-null*. Symmetrically, `td` and `xb` are non-null but useless, and therefore need not be evaluated. Nevertheless, there are two special cases where TAPENADE explicitly resets implicit-null variables: (1) when the control flow merges and the other incoming flow has an explicit non-null derivative for this variable, and (2) when the end of the differentiated program is reached and the derivative is an output. Notice also that some of the user-given independent and dependent variables may turn out to be inactive. If so, TAPENADE removes them automatically.

3.4 Control structure

The following example 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(adTo)
DO i=adTo,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
```

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 $\dot{I}_k$. The control structures are unchanged. In *reverse* mode, TAPENADE applies the *Store All* strategy (*cf* section 2), resulting in a *forward sweep* followed by a *backward sweep*. The forward sweep runs the original procedure, storing into a stack the variables potentially required by the derivatives. In addition, the forward sweep stores into the same stack the *control* information, used by the backward sweep to reproduce between the $\overline{I}_k$ the reverse of the original control flow. The stack is used classically through several `PUSH` and `POP` subroutines, according to the type of the 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 (`GOTO`'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*.

3.5 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. The following example illustrates this. In *tangent mode*, a call to `SUB` just gives a call to the differentiated `SUB_D`. 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 1).

original program	TAPENADE **reverse: forward sweep**

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

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)`
`xd = y*xd`
`x = x*y`
	`xb = y*xb`
	`CALL POPREAL4(x)`
	`CALL SUB_B(a, ab, x, xb,`
	`1.5, arg2b, z)`
	`CALL POPREAL4(x)`
	`xb = 3*x**2*xb`

One principle of TAPENADE is procedure *generalization*, as opposed to *specialization*. 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 useless derivative result. Suppose `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 "`arg2b`" in reverse.

In the reverse mode, checkpointing requires taking a *snapshot*. TAPENADE runs a preliminary In-Out analysis to find a minimal snapshot, made of variables that are both *used* by the procedure and *overwritten* before the differentiated procedure is called. On the example, the In-Out analysis could prove that this is only the case for `x`.

4 SPECIFIC IMPROVEMENTS 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 cheaper gradients. 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.

4.1 *To Be Restored* analysis

We saw that intermediate values need to be stored 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) [5, 11] does this in TAPENADE. In the following example, 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`

4.2 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. TAPENADE implements a *data-dependency* analysis that allows it to safely move and gather initializations and increments of the same differentiated variable, whenever possible. 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. We illustrate the effect of this improvement 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`

4.3 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 to the user. This message should not be ignored, because it may point to a future problem in the differentiated code, especially in the reverse mode.

There is another form of aliasing, local to an instruction, where an assigned variable may or may not be the same as 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. The following example illustrates this: there is a local aliasing in the third instruction, because equality between `i` and `n-i` could not be decided.

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)`

4.4 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. Expressions are not split during the forward sweep, to avoid intermediate variables that might need to be `PUSH`'ed and `POP`'ed. Splitting occurs only in the reverse sweep, and occurs only at carefully selected places in the differentiated expressions. The following example illustrates this on a "not so long" expression, to keep things readable. 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`

4.5 Dead adjoint code

Reverse differentiation of the program P that computes function F yields program $\overline{P}$ that computes the gradient of F. The original results of P , which are also computed by the forward sweep of $\overline{P}$, are *not* a result of $\overline{P}$. Only the gradient is useful. Moreover in most implementations the original results will be overwritten and lost during the backward sweep of $\overline{P}$. Therefore some of the last instructions of the forward sweep of $\overline{P}$ are actually dead code. TAPENADE has a prototype mechanism to remove this dead code, and a more complete implementation is under way. The following example shows the effect on a small program which terminates on a test, with some dead adjoint code at the end of each branch.

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`

5 USING THE TAPENADE TOOL

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

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

Alternatively, the TAPENADE web server

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

requires no installation and of course always runs the latest version. It can be triggered in a few clicks from most web browsers. All TAPENADE documentation, with tutorial and an ever-growing reference manual, is available at:

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

User input to TAPENADE consists in command-line options, directives in the original code, as well as configuration files. Consider for instance *black-box* procedures, i.e. procedures eventually called by the code to be differentiated, whose source is hidden (e.g. libraries). If nothing is known about a *black-box* procedure, the inter-procedural analyses of TAPENADE will make conservative assumptions, and the code produced will be less efficient. TAPENADE lets the user specify in a configuration file useful summarized information about *black-box* procedures, about parameters read and written, and about their derivatives. This is most useful in large industrial codes. Black-box routines are used to correctly differentiate procedures that use complex libraries, such as MPI. Alternatively, even procedures whose source is available are sometimes better differentiated by hand. By declaring them as black-box, the user tells TAPENADE to differentiate the rest automatically and then use the user-defined differentiated procedure.

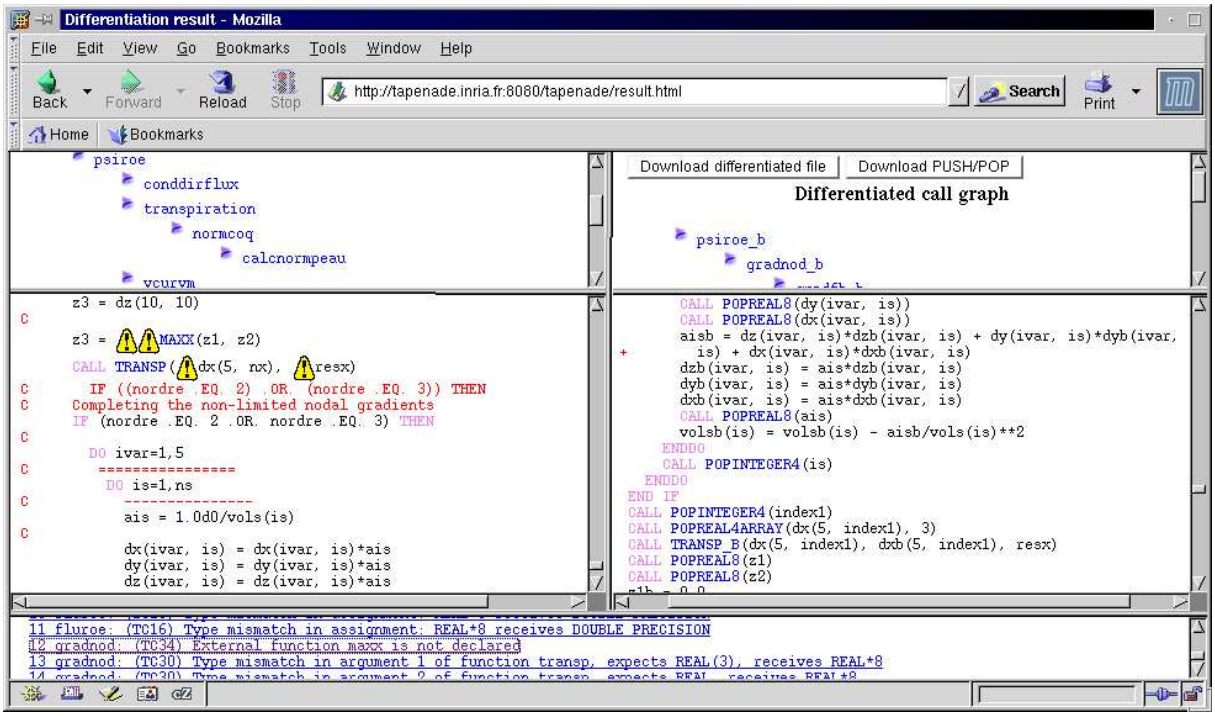

Figure 2: HTML interface for TAPENADE output

The graphical user interface shown on figure 2 helps examine TAPENADE output, exhibiting correspondence between original and differentiated code. This user interface consists of HTML files, and is therefore accessible from the web server as well as from a local installation. In its bottom frame, the interface also lists messages issued by TAPENADE, with location in the source. There are many types of messages, such as type conflicts, wrong number of arguments or dimensions, aliasing, or variables used before initialized. Although the temptation is strong, these messages should not be ignored right away. Especially when reverse AD is concerned, these messages may indicate 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.

6 CONCLUSION

We have presented the AD tool TAPENADE. We gave a basic theoretical understanding of AD and then showed how it relates to actual program transformation. We examined TAPENADE output in detail to gain precise understanding and confidence into the internal analyses and decisions made by this tool.

Our goal is to promote the use of TAPENADE in the scientific computing community, and more importantly the use of the reverse mode of AD for optimization [4, 7, 9] and inverse problems [10]. Discussion with end-users drives our research very strongly.

We are currently extending TAPENADE in several directions. A new version that fully accepts FORTRAN95 is coming soon, and C in next on the list. Program static analyses will be developed further, particularly pointer analysis. There is also work to be done in the definition of directives that drive AD efficiently. Our research work, focused on the reverse mode, both justifies existing refinements of TAPENADE and leads to new improvements.

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. *Optimization Methods and Software*, 18(5):615–627, 2003.

[5] C. Faure and U. Naumann. The taping problem in automatic differentiation. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 293–298, 2001.

[6] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997. [www http://www.autodiff.com/tamc].

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

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

[9] L. Hascoët, M. Vázquez, and A. Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In *V.Kumar et al., editors, Proceedings of ICCSA'03, Montreal, Canada, LNCS 2668*, pages 85–94. Springer, 2003.

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

[11] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In *Proceedings of the ICCS 2000 Conference on Computational Science, Part II*, LNCS. Springer, 2002.