

Automatic Differentiation by Program Transformation

Laurent Hascoët

INRIA Sophia-Antipolis, TROPICS team

<http://www-sop.inria.fr/tropics/>

(version: april 2007)

Contents

1	Introduction	4
2	Automatic Differentiation of Programs	4
2.1	Computer Programs and Mathematical Functions	6
2.2	The Tangent Mode of Automatic Differentiation	9
2.3	The Reverse Mode of Automatic Differentiation	11
3	Motivating Applications of Reverse AD	17
3.1	Inverse Problems and Data Assimilation	17
3.2	Optimization in Aerodynamics	22
4	Static Data Flow Analyses for a better AD	25
4.1	Activity Analysis and reinitializations of derivatives	28
4.2	TBR Analysis	30
4.3	Detection of Aliasing	30
4.4	Dead Adjoint Code	31
4.5	Undecidability	32
5	AD Tools and TAPENADE	33
5.1	TAPENADE	35
6	Validation of Differentiated Programs	44
7	Conclusion	45

List of Figures

1	<i>AD in tangent mode</i>	10
2	<i>Comparison of Tangent and Reverse AD modes</i>	11
3	<i>The “Recompute-All” tactic</i>	13
4	<i>The “Store-All” tactic</i>	13
5	<i>AD in reverse mode, Store-All tactic</i>	14
6	<i>Checkpointing on the “Recompute-All” tactic</i>	15
7	<i>Checkpointing on the “Store-All” tactic</i>	16
8	<i>Oceanography gradient by reverse AD on OPA 9.0</i>	21
9	<i>Shock wave patterns on the near and far fields.</i>	22
10	<i>Gradient of the Cost Functional on the skin of a jet</i>	24
11	<i>Pressure distribution in a plane below the aircraft</i>	25
12	<i>Names of differentiated symbols</i>	26
13	<i>Differentiation of a simple assignment</i>	27
14	<i>Instructions simplifications due to Activity Analysis</i>	29
15	<i>Removing unnecessary storage through To Be Recorded analysis</i>	30
16	<i>Detection and correction of aliasing in the reverse mode</i>	31
17	<i>Removing useless dead adjoint code</i>	32
18	<i>TAPENADE’s ID card</i>	36
19	<i>Overall Architecture of TAPENADE</i>	38
20	<i>Ordering of generic static data flow analyses in TAPENADE</i>	39
21	<i>Ordering of AD static data flow analyses in TAPENADE</i>	40
22	<i>HTML interface for TAPENADE input</i>	41
23	<i>HTML interface for TAPENADE output</i>	42
24	<i>Checkpointing on calls in TAPENADE reverse AD</i>	43
25	<i>AD and other ways to compute derivatives</i>	45

1 Introduction

We present Automatic Differentiation (AD), an innovative program transformation technique to obtain derivatives of functions implemented as programs. Unlike parallelization, AD does not only preserve the semantics of program, but rather augments it. AD adds new program outputs which are mathematical derivatives of the original results. Efficient AD relies on a range of program static analysis, some of them specific to AD.

This paper has three main goals:

- to present AD and its theoretical background (section 2)
- to present uses of AD in Scientific Computation (section 3)
- to present the compiler algorithms inside AD tools (section 4)

Section 5 does a general presentation of existing AD tools, with some emphasis on the TAPENADE tool developed by our team. Section 6 gives indications on the most popular validation methods for AD-differentiated codes. Section 7 concludes with a general picture of AD compared to alternative approaches, and the current limitations and challenges of AD tools.

2 Automatic Differentiation of Programs

Automatic Differentiation (AD) 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 reference, we recommend the monograph [15], selected articles of recent conferences [6, 4], or the AD community website at www.autodiff.org.

The goal of AD is to compute derivatives without going back to the underlying mathematical equations, considering only the source program P . This will spare a tedious extra discretization and implementation phase. Moreover, sometimes the mathematical equations are not available. How can we reach this goal? Naturally, one can do *divided differences*. For a given set of program's inputs X , program P computes a result Y . Given now some normalized direction dX in the space of the inputs, one can run program P

again on the new set of inputs $X + \varepsilon.dX$, where ε is some very small positive number. Divided Differences return an *approximation* of the derivative by the formula:

$$\frac{P(X + \varepsilon.dX) - P(X)}{\varepsilon}.$$

The *centered* divided differences give a better approximation, at the cost of an extra run of P by the formula:

$$\frac{P(X + \varepsilon.dX) - P(X - \varepsilon.dX)}{2\varepsilon}.$$

In any case, these are just approximations of the derivatives. Ideally, the exact derivative is the limit of these formulas when ε tends to zero. But this makes no sense on a real computer, since very small values of ε lead to truncation errors, and therefore to erroneous derivatives. This is the main drawback of divided differences: some trade-off must be found between truncation errors and approximation errors. Finding the best ε requires numerous executions of the program, and even then the computed derivatives are just approximations.

To get rid of approximation errors, AD computes derivatives **analytically**. Each time the original program holds some variable v , the differentiated program holds an additional variable with the 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 :

$$a = b * T(10) + c \tag{1}$$

Suppose also that variables \dot{b} , \dot{c} , and array \dot{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{a} , using b , c , T and their differentials \dot{b} , \dot{c} , and \dot{T} . These must somehow amount to:

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

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

At this point, let us mention an elegant manner to implement AD: *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`. Even when the language does not support overloading, there is an elegant workaround [20]: just replace all `REAL`'s by `COMPLEX` numbers, and put their derivative into the imaginary part! It is easy to check that the arithmetic of `COMPLEX` numbers can be a good approximation of overloaded operations on derivatives. The advantage of overloading is that it requires very little code transformation. Drawbacks are that not all languages support overloading (or `COMPLEX` numbers), that overloaded programs are poorly optimized by the compiler, and more importantly that overloading is not suitable for the *reverse* mode of AD (*cf* section 2.3). Therefore, overloading is not the choice method when efficiency is a concern. In this case, **source transformation** techniques are preferred, using technology from compilation and parallelization.

Automatic Differentiation can compute many different sorts of derivatives. At least in theory, it can yield a complete Jacobian matrix, i.e. the partial derivatives of each output with respect to each input. Higher order differentiation is possible too, e.g. computation of Hessian tensors. In practice, these mathematical objects are often too large or too expensive to compute. Therefore, AD can also return smaller objects at a cheaper cost, e.g directional derivatives, gradients, directional higher-order derivatives, or Taylor series expansions. Similar techniques apply to domains slightly outside of strict differentiation: AD techniques are used to build programs that compute on *intervals* or on *probabilistic* data. We believe the most promising of these “*modes*” is the computation of *gradients*, whereas computation of first-order directional derivatives (called *tangents*) is the most straightforward, and is therefore an interesting basic mode. Let’s look at the theoretical background that we need for these two particular modes.

2.1 Computer Programs and Mathematical Functions

Remember that we want to compute exact derivatives analytically, based on a given computer program `P` which is seen as the principal *specification* of the

mathematical function f to differentiate. Therefore, we need to introduce a general framework in which programs can be identified with functions. We first identify programs with sequences of instructions, identified in turn with composed functions.

Programs contain *control*, which is essentially discrete and therefore non-differentiable. Consider the set of all possible run-time sequences of instructions. Of course there are a lot (often an infinity!) of such sequences, and therefore we never build them explicitly! The control is just how one tells the running program to switch to one sequence or another. For example this small C program piece:

```
if (x <= 1.0)
    printf("x too small");
else {
    y = 1.0;
    while (y <= 10.0) {
        y = y*x;
        x = x+0.5;
    }
}
```

will execute according to the control as one of the following *sequences of instructions*:

```
printf("x too small");
y = 1.0;
y = 1.0; y = y*x; x = x+0.5;
y = 1.0; y = y*x; x = x+0.5; y = y*x; x = x+0.5;
y = 1.0; y = y*x; x = x+0.5; y = y*x; x = x+0.5; y = y*x; x = x+0.5;
```

and so on... Each of these sequences is differentiable. The new program generated by Automatic Differentiation uses the original program's control to guarantee it computes the differentials of the actual run-time *sequence of instructions*. This is only **piecewise** differentiation. Thus, this differentiated program will probably look like:

```

if (x <= 1.0)
  printf("x too small");
else {
  dy = 0.0;
  y = 1.0;
  while (y <= 10.0) {
    dy = dy*x + y*dx;
    y = y*x;
    x = x+0.5;
  }
}

```

However it sometimes happens, like in this example, that the control itself depends on differentiated variables. In that case, a small change of the initial values may result in a change of the control. Here, a small change of x may change the number of iterations of the while loop, and the derivative is not defined any more. Yet the new program generated by Automatic Differentiation will return a result, and using this derivative may lead to errors. In other words, the original program, with control, is only piecewise differentiable, and “state of the art” AD does not take this into account correctly. This is an open research problem. In the meantime, we simply assume that this problem happens rarely enough. Experience on real programs shows that this is a reasonable assumption. However, this problem is widely known and it limits the confidence end-users place into AD.

Now that programs are identified with sequences of instructions, these sequences are identified with composed functions. Precisely, the sequence of instructions :

$$I_1; I_2; \dots; I_{p-1}; I_p;$$

is identified to the function :

$$f = f_p \circ f_{p-1} \circ \dots \circ f_1$$

Each of these functions is naturally extended to operate on the domain of all the program variables: variables not overwritten by the instruction are just transmitted unchanged to the function’s result. We can then use the chain

rule to write formally the derivative of the program for a given input X :

$$\begin{aligned}
 f'(X) &= (f'_p \circ f'_{p-1} \circ f'_{p-2} \circ \dots \circ f'_1(X)) \\
 &\cdot (f'_{p-1} \circ f'_{p-2} \circ \dots \circ f'_1(X)) \\
 &\cdot \dots \\
 &\cdot (f'_1(X)) \\
 &= f'_p(W_{p-1}) \cdot f'_{p-1}(W_{p-2}) \cdot \dots \cdot f'_1(W_0)
 \end{aligned} \tag{3}$$

Each f'_k , derivative of function f_k , is a Jacobian matrix. For short, we defined $W_0 = X$ and $W_k = f_k(W_{k-1})$ to be the values of the variables just after executing the first k instructions. Computing the derivatives is just computing and multiplying the elementary Jacobian matrices $f'_k(W_{k-1})$.

Let us conclude this section with some bad news: for average size applications, the Jacobian matrix $f'(X)$ is often too large! Therefore it is not reasonable to compute it explicitly: the matrix-matrix products would be too expensive, and the resulting matrices would be too large to be stored. Except in special cases, one must not compute $f'(X)$. Fortunately, many uses of derivatives do not need $f'(X)$, but only a “view” of it, such as $f'(X).\dot{X}$ for a given \dot{X} or $f''(X).\bar{Y}$ for a given \bar{Y} . These two cases will be discussed in the next two sections.

2.2 The Tangent Mode of Automatic Differentiation

For some applications, what is needed is the so-called *sensitivity* of a program. For a given small variation \dot{X} in the input space, we want the corresponding first-order variation of the output, or “sensitivity”. By definition of the Jacobian matrix, this sensitivity is $\dot{Y} = f'(X).\dot{X}$. Historically, this is the first application of AD, probably because it is the easiest.

Recalling equation (3), we get:

$$\dot{Y} = f'_p(W_{p-1}) \cdot f'_{p-1}(W_{p-2}) \cdot \dots \cdot f'_1(W_0) \cdot \dot{X} \tag{4}$$

To compute \dot{Y} efficiently, one must of course do it from right to left, because Matrix×Vector products are so much cheaper than Matrix×Matrix products. This turns out to be easy, because this formula requires W_0 first, and then W_1 , and so on until W_{p-1} . In other words, the intermediate values from the original program P are used as they are computed. Differentiated instructions, that compute the Jacobian matrices and multiply them, can be

done along with the initial program. We only need to interleave the original instructions and the derivative instructions.

In the tangent mode, the differentiated program is just a copy of the given program, Additional derivative instructions are inserted just before each instruction. The control structures of the program are unchanged, i.e. the Call Graph and Flow Graph have the same shape in P and P' . Figure 1 illustrates AD in tangent mode. On the left column is an example subroutine, that measures a sort of discrepancy between two given arrays T and U . The right column shows the tangent differentiated subroutine, which computes tangent derivatives, conventionally shown with a *dot* above. For example, the differentiated instruction that precedes instruction $e = \text{SQRT}(e2)$, implements the following vector assignment that multiplies the instruction's elementary Jacobian by the vector of tangent derivatives:

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

original: $T, U \mapsto e$	tangent mode: $T, \dot{T}, U, \dot{U} \mapsto e, \dot{e}$
<code>e2 = 0.0</code>	<code>$\dot{e}2 = 0.0$</code>
<code>do i=1,n</code>	<code>do i=1,n</code>
<code>e1 = T(i)-U(i)</code>	<code>$\dot{e}1 = \dot{T}(i)-\dot{U}(i)$</code>
<code>e2 = e2 + e1*e1</code>	<code>$\dot{e}2 = \dot{e}2 + 2.0*e1*\dot{e}1$</code>
<code>end do</code>	<code>end do</code>
<code>e = SQRT(e2)</code>	<code>$\dot{e} = 0.5*\dot{e}2/\text{SQRT}(e2)$</code>
	<code>e = SQRT(e2)</code>

Figure 1: *AD in tangent mode*

The tangent mode of AD is rather straightforward. However, it can benefit from some specific optimizations based on static analysis of the program. In particular *activity* analysis (*cf* section 4.1) can vastly simplify the tangent differentiated program, by statically detecting derivatives that are always zero.

2.3 The Reverse Mode of Automatic Differentiation

Maybe the most promising application of AD is the computation of *gradients* of a program. If there is only one scalar output $f(X) = y$ ($n = 1$), the Jacobian $f'(X)$ has only one row, and it is called the *gradient* of f at point X . If $n > 1$, one can go back to the first case by defining a *weighting* \bar{Y} , such that $\langle f(X), \bar{Y} \rangle$ is now a scalar. If the components of $f(X)$ were some kind of optimization criteria, this amounts to defining a single optimization criterion $\langle f(X), \bar{Y} \rangle$, whose gradient at point X can be computed. By definition of the Jacobian matrix, this gradient is $\bar{X} = f'^t(X) \cdot \bar{Y}$. This gradient is useful in optimization problems (*cf* section 3), because it gives a descent direction in the input space, used to find the optimum.

Recalling equation (3), we get:

$$\bar{X} = f_1'^t(W_0) \cdot f_2'^t(W_1) \cdot \dots \cdot f_{p-1}'^t(W_{p-2}) \cdot f_p'^t(W_{p-1}) \cdot \bar{Y} \quad (5)$$

Here also, efficient computation of equation (5) must be done from right to left. Again, one can show that the computation cost of this gradient is only a small multiple of the computation cost of the original function f by P. Figure 2 underlines this contrast between tangent and reverse modes of AD: the program \dot{P} resulting from the *tangent mode* of AD, which also costs a small multiple of P's execution time, returns only a column of the Jacobian Matrix. To obtain the full Jacobian, or equivalently the gradient when there is only one output, one must run \dot{P} once for each element of the Cartesian basis of

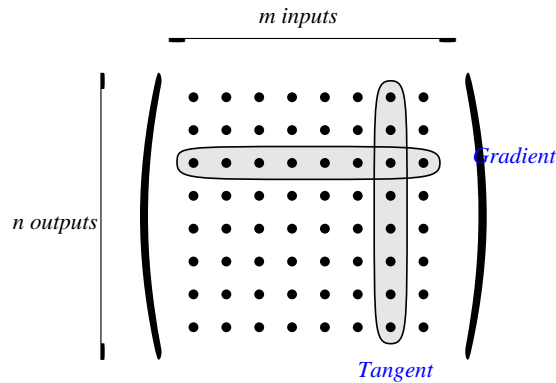


Figure 2: Comparison of Tangent and Reverse AD modes

the input space. Therefore the cost of computing the Jacobian using the

tangent mode is proportional to the dimension of the input space. Similarly, to compute the Jacobian using Divided Differences also requires a number of evaluations of P proportional to the dimension of the input space. For each basis direction e_i in the input space, i.e. for each component of vector X , one must run P at least once for $X + \varepsilon.e_i$, and in fact more than once to find a good enough ε . In contrast, the program \bar{P} resulting from the *reverse mode* of AD, which also costs a small multiple of P 's execution time, returns a row of the Jacobian Matrix. To obtain the full Jacobian, one must run \bar{P} once for each element of the Cartesian basis of the output space. If $m \gg n$ and in particular when there is only one output, the reverse mode of AD provides the gradient at a much cheaper cost.

However, there is a difficulty with the reverse mode: the intermediate values W_{p-1} are used first, and then W_{p-2} , and so on until W_0 . This is the inverse of their computation order in program P . A complete execution of P is necessary to get W_{p-1} , and only then can the Jacobian \times vector products be evaluated. But then W_{p-2} is required, whereas instruction I_{p-1} may have overwritten it! There are mainly two ways to achieve this, called the *Recompute-All* (RA) and the *Store-All* (SA) approaches.

The RA approach recomputes each needed W_k on demand, by restarting the program on input W_0 until instruction I_k . This is the fundamental tactic of the AD tool TAMC/TAF [11]. The cost is extra execution time, grossly proportional to the square of the number of run-time instructions p . Figure 3 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 \bar{I}_k which implement $\bar{W}_{k-1} = f_k^{tt}(W_{k-1}) \cdot \bar{W}_k$. The big black dot represents the storage of all variables needed to restart execution from a given point, which is called a *snapshot*, and the big white dots represent restoration of these variables from the snapshot.

The SA approach stores each W_k in memory, onto a stack, during a preliminary execution of program P , known as the *forward sweep*. Then follows the so-called *backward sweep*, which computes each $f_k^{tt}(W_{k-1})$ for $k = p$ down to 1, popping the W_{k-1} from this stack upon demand. This is the basic tactic in AD tools ADIFOR [3, 5] and TAPENADE. The cost is memory space, essentially proportional to the number of run-time instructions p . Figure 4 summarizes this tactic graphically. Small black dots represent storage of the W_k on the stack, before next instruction might overwrite them, and small white dots represent their popping from the stack when needed. We draw these dots smaller than on figure 3 because it turns out we don't need to store

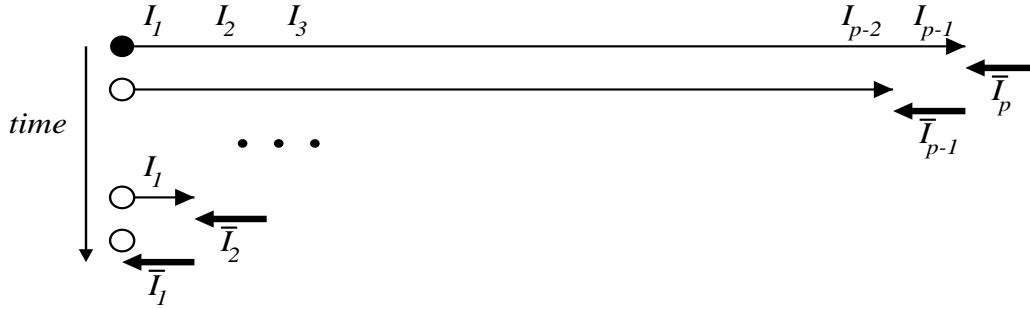


Figure 3: The “Recompute-All” tactic

all W_k , but only the variables that will be overwritten by I_{k+1} . Figure 5 il-

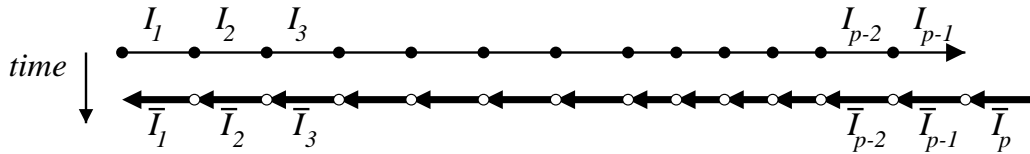


Figure 4: The “Store-All” tactic

lustrates AD in reverse mode, using the SA approach as done by TAPENADE. Actually, TAPENADE generates a simplified code, resulting from static analysis and improvements described later in this paper (section 4.2). For clarity, figure 5 does not benefit from these improvements. The original program is the one of figure 1. Reverse derivatives are conventionally shown with a *bar* above. The *forward sweep* is on the left column, and the *backward sweep* on the right. Notice that in the backward sweep, 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:

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

This takes two derivative instructions, because \overline{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

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

Figure 5: *AD in reverse mode, Store-All tactic*

be stored before each instruction (*cf* section 4.2). Only the value(s) that are going to be overwritten need be stored.

The RA and SA approaches appear very different. However, on large programs P , neither the RA nor the SA approach can work. The SA approach uses too much memory (almost proportional to the **run-time** number of instructions). The RA approach consumes computation time (it will grossly square the run-time number of instructions). Both ways need to use a special trade-off technique, known as *checkpointing*. The idea is to select one or many pieces of the run-time sequence of instructions, possibly nested. For each piece p , 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. a part of the memory state at the beginning of p . On real programs, language constraints usually force the pieces to be subroutines, loops, loop bodies, or fragments of straight-line code.

There has been little work on the evaluation and comparison of these strategies. With the notable exception of Griewank's schedule of nested checkpoints (SA strategy) [14], which was proved optimal for P being a loop with a fixed number of iterations [16], there is no known optimal checkpointing strategy for arbitrary programs. Moreover, no theoretical compari-

son between the RA and SA approaches exist, nor a common framework in which these approaches could be combined. This is an open research problem, which could yield a huge benefit for AD tools, and help disseminate AD among users.

Let us now compare checkpointing on RA and SA in the ideal case of a pure straight-line program. We claim that checkpointing makes RA and SA come closer. Figure 6 shows how the RA approach can use checkpointing for one program piece p (the first part of the program), and then for two levels of nested pieces. On very large programs, 3 or more nested levels can be useful. At the second level, the memory space of the snapshot can be reused for different program pieces. The benefit comes from the checkpointed piece being executed fewer times. The cost is memory storage of the snapshot, needed to restart the program just after the checkpointed piece. The benefit is higher when p is at the beginning of the enclosing program piece. Similarly,

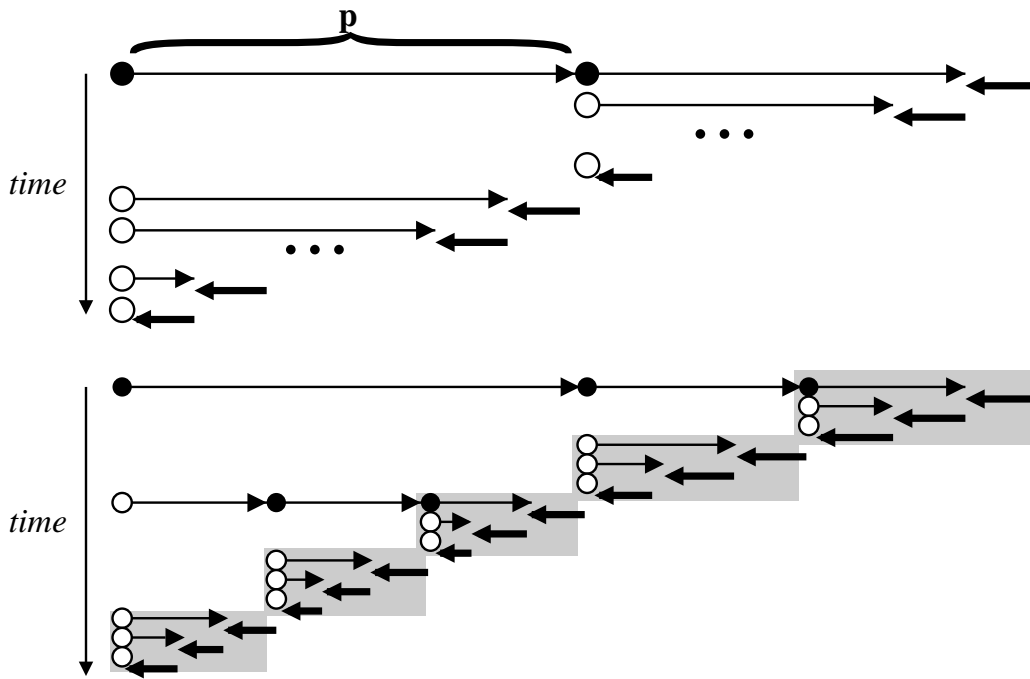


Figure 6: *Checkpointing on the “Recompute-All” tactic*

figure 7 shows how the SA approach can use the same one-level and two-levels checkpointing schemes. Again, the snapshot space used for the second

level of checkpointing is reused for two different program pieces. The benefit comes from the checkpointed piece being executed the first time without any storage of intermediate values. This divides the maximum size of the stack by 2. The cost is again the memory size of the snapshots, plus this time an extra execution of the program piece p . This makes the two schemes come closer as the number of nested checkpointing levels grow. On figure 6, the part on

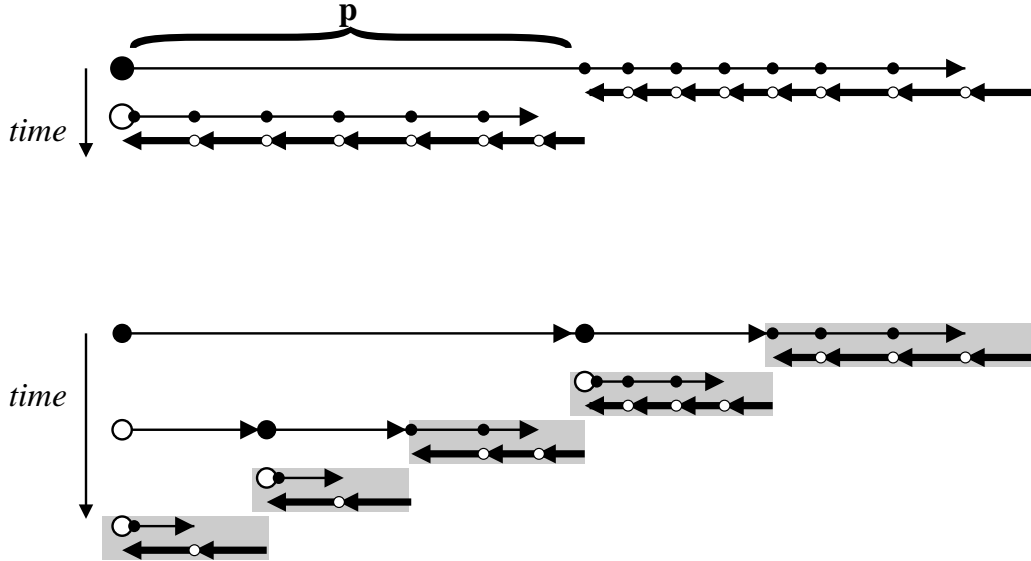


Figure 7: Checkpointing on the “Store-All” tactic

a gray background is a smaller scale reproduction of the basic RA scheme of figure 3. Similarly on figure 7, the gray box is a smaller scale reproduction of the basic SA scheme of figure 4. Apart from what happens at the “leaves” (the gray boxes), the figures 6 and 7 are identical. This shows that RA and SA with intense checkpointing can differ very little. The question remains to compare pure SA and pure RA, but it becomes less crucial as these are applied to smaller pieces of the program. However, we believe SA is more efficient, especially on small pieces of program, because the stack can stay in cache memory. We chose the SA approach for our AD tool TAPENADE. But SA could do better with some amount of recomputing!

3 Motivating Applications of Reverse AD

3.1 Inverse Problems and Data Assimilation

We call *inverse* all problems where unknown values cannot be measured directly, but instead we know some other measurable values which are *consequences of* – or *depend on* – the unknown. Given actual measures of the measurable values, the inverse problem is how to find the unknown values which are behind them.

To do this, we assume we have a physical model that we believe represents the way the unknown values determine the measurable values. When this model is complex, then the inverse problem is nontrivial. From this physical model we get a mathematical model, which is in general a set of partial differential equations. Let us also assume, but this is not absolutely necessary, that from the mathematical model, through discretization and resolution, we get a program that computes the measurable values from the unknown values.

Let us formalize the problem. This classical formalization comes from *optimal control theory* and a good reference is [21]. We are studying the state W of a given system. In general, this state is defined for every point in space, and also if time is involved for every instant in an observation period $[0, T]$. Traditionally, the time coordinate is kept apart from the others (i.e. space). The mathematical model relates the state W to a number of external parameters, which are the collection of initial conditions, boundary conditions, model parameters, etc, i.e. all the values that determine the state. Some of these parameters, that we call γ , are the unknown of our inverse problem. In general this relation between W and γ is implicit. It is a set of partial differential equations that we write:

$$\Psi(\gamma, W) = 0 \tag{6}$$

Equation (6) takes into account all external parameters, but we are only concerned here by the dependence on γ . In optimal control theory, we would call γ our control variable.

Any value of γ thus determines a state $W(\gamma)$. We can easily extract from this state the measurable values, and of course there is very little chance that these values exactly match the values actually measured W_{obs} . Therefore we start an optimization cycle to modify the unknown values γ , until the resulting measurable values match best. We thus define a *cost function* that measures the discrepancy on the measurable values in $W(\gamma)$. In practice, not

all values in $W(\gamma)$ can be measured in W_{obs} , but nevertheless we can define this cost function J as the sum at each instant of some squared norm of the discrepancy of each measured value $\|W(\gamma) - W_{obs}\|^2$.

$$j(\gamma) = J(W(\gamma)) = \frac{1}{2} \int_{t=0}^T \|W(\gamma)(t) - W_{obs}(t)\|^2 dt \quad (7)$$

Therefore the inverse problem is to find the value of γ that minimizes $j(\gamma)$, which is such that $j'(\gamma) = 0$. If we use a gradient descent algorithm to find γ , then we need at least to find the value of $j'(\gamma)$ for each γ .

Here are two illustration examples:

- If the system we study is a piece of earth crust, it is difficult to measure directly the locations of the different layers of rock. However, these locations condition shock wave propagation, and eventually condition the measurable delay after which an initial shock is received at a distant place. So γ is the location of layers of rock, Ψ models wave propagation inside the rock, W is the position of the waves at each instant, from which we deduce the theoretical reception delays at points of measurement. J is the discrepancy we must minimize to find the best estimation of rock layers location. The same method applies to find an unknown drag coefficient of the bottom of a river, given the model Ψ that captures the shape of the river bed, and measured values of the river's surface shape and input flow.
- In meteorology, the system studied is the evolution of the atmosphere. The data assimilation problem looks for the best estimation of the initial state from which the simulation will start. This initial state W_0 at $t = 0$ is largely unknown. This is our γ . All we have is measurements at various places for various times in $[0, T]$. We also know that this initial state and all later states in $[0, T]$ must obey the atmospheric equations $\Psi(\gamma, W) = 0$. The inverse problem that looks for the initial state $\gamma = W_0$ that generates the sequence of states at each time in $[0, T]$ which is closest to the observed values, using a gradient descent, is called *variational data assimilation*.

To find $j'(\gamma)$, the mathematical approach first applies the chain rule to equation (7), yielding:

$$j'(\gamma) = \frac{\partial J(W(\gamma))}{\partial \gamma} = \frac{\partial J}{\partial W} \frac{\partial W}{\partial \gamma} \quad (8)$$

The derivative of W with respect to γ comes from the state implicit equation (6), which we differentiate with respect to γ to get:

$$\frac{\partial \Psi}{\partial \gamma} + \frac{\partial \Psi}{\partial W} \frac{\partial W}{\partial \gamma} = 0 \quad (9)$$

Assuming this can be solved for $\frac{\partial W}{\partial \gamma}$, we can then replace it into equation (8) to get:

$$j'(\gamma) = -\frac{\partial J}{\partial W} \frac{\partial \Psi}{\partial W}^{-1} \frac{\partial \Psi}{\partial \gamma} \quad (10)$$

Now is the time to consider complexity of resolution. Equation (10) involves one system resolution and then one product. First notice that

$$\frac{\partial \Psi}{\partial W}$$

is definitely too large to be computed explicitly, and therefore its inverse cannot be computed and stored either. Nowadays both Ψ and W are discretized with millions of dimensions. So our choices are either to run an iterative resolution for

$$\frac{\partial \Psi}{\partial W}^{-1} \frac{\partial \Psi}{\partial \gamma} \quad (11)$$

and then multiply the result by $\frac{\partial J}{\partial W}$, or else to run an iterative resolution for

$$\frac{\partial J}{\partial W} \frac{\partial \Psi}{\partial W}^{-1} \quad (12)$$

and then multiply the result by $\frac{\partial \Psi}{\partial \gamma}$. We notice that $\frac{\partial \Psi}{\partial \gamma}$ has many columns, following the dimension of γ , which can be several thousands. Therefore computation of (11) requires as many resolutions of the linear system

$$\frac{\partial \Psi}{\partial W} x = \frac{\partial \Psi}{\partial \gamma}$$

Conversely, j is a scalar, $\frac{\partial J}{\partial W}$ is a row vector, and it takes only one resolution to compute (11), solving

$$\Pi^* \frac{\partial \Psi}{\partial W} = \frac{\partial J}{\partial W}$$

for Π , which is called the *adjoint state*. This second approach is more efficient. To summarize, this preferred *adjoint method* first solves

$$\frac{\partial \Psi^*}{\partial W} \cdot \Pi = \frac{\partial J^*}{\partial W}$$

for the adjoint state Π , then just computes

$$j'(\gamma) = -\Pi^* \frac{\partial \Psi}{\partial \gamma}$$

Suppose now that we already have a resolution program, i.e. a procedure P_Ψ which, given γ , returns W_γ . We also have a procedure P_j which, given a W , evaluates the cost function, i.e. the discrepancy between W and the observed W_{obs} . Then we can avoid all the programming step involved by the above mathematical method, using AD. Automatic Differentiation in reverse mode of the program that computes

$$j = P_j(P_\Psi(\gamma))$$

directly gives the gradient of j , i.e. the desired $j'(\gamma)$. This is indeed very close to the mathematical resolution with the adjoint state: the reverse mode actually computes a discretized adjoint on the program. One difference is that the adjoint mechanism is applied to the whole program $P_j \circ P_\Psi$, including the resolution algorithm, whereas the resolution algorithm for Π above may be different from the resolution algorithm for W .

So to get $j'(\gamma)$, we can either write the adjoint equations, then discretize them and solve them, or else we can use the reverse mode of AD on the program that computes j from γ . The first method is more difficult, because it involves a new implementation. The second method ideally doesn't require additional programming.

There is a difficulty though. The resolution for $j'(\gamma)$ by AD uses reverse differentiation of P_Ψ . P_Ψ takes γ and returns W , computed iteratively. $\overline{P_\Psi}$ takes \overline{W} and returns $\overline{\gamma}$, computed iteratively with the same number of iterations. The question is "will this second iteration converge?". Jean-Charles Gilbert has shown [12] that under some complex but widely satisfied constraints, if an iterative resolution converges to a result, then its AD derivative converges to the derivative of the result. So we are confident that $\overline{P_\Psi}$ eventually converges to the desired $\overline{\gamma}$. But we are not sure it will converge at same speed! In other words, it is not sure that an efficient algorithm to compute $W(\gamma)$ and then $j(\gamma)$ yields an efficient resolution algorithm to compute $j'(\gamma)$.

The next section illustrates an hybrid approach that uses the AD reverse derivatives for the non-iterative part of the resolution, and then solves the adjoint system by hand with an ad-hoc resolution algorithm. This may be an answer to the difficulty above.

Before that, let’s illustrate the use of reverse AD as described above, on an example taken from Oceanography. OPA 9.0 is a large oceanography code. One of its simplest “configurations” i.e. test cases, known as GYRE, simulates the behavior of a rectangular basin of water put on the tropics between latitudes 15° and 30° , with the wind blowing to the East. In order to perform data assimilation, we need a gradient of an objective function, related to the heat flux across some boundary at the northern angle, with respect to an initial parameter, which is the temperature field in the complete domain 20 days before. This system was discretized with $32 \times 22 \times 31$

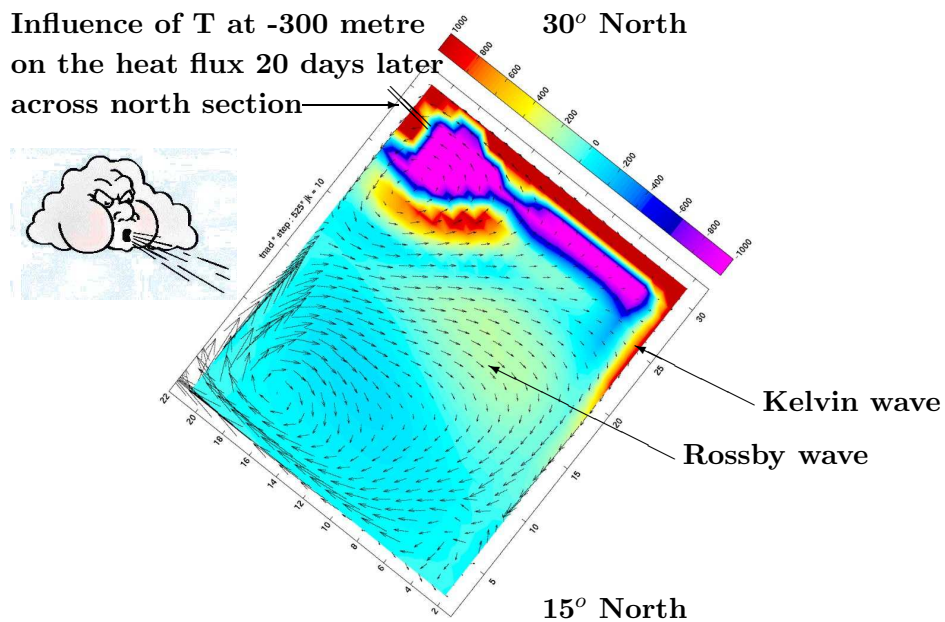


Figure 8: *Oceanography gradient by reverse AD on OPA 9.0*

nodes and 4320 time steps. The original simulation took 92 seconds, and the differentiated program computed the gradient above in 657 seconds, which is only seven times as long! Of course checkpointing was absolutely necessary to reverse this long simulation, yielding several recomputations of the same

program steps, but nevertheless the factor seven is much better than what divided differences would require.

3.2 Optimization in Aerodynamics

This application uses AD to optimize the shape of a supersonic aircraft, in order to minimize the sonic boom felt on the ground. The problem, shown on figure 9, is modeled numerically as the following chain of operations, that

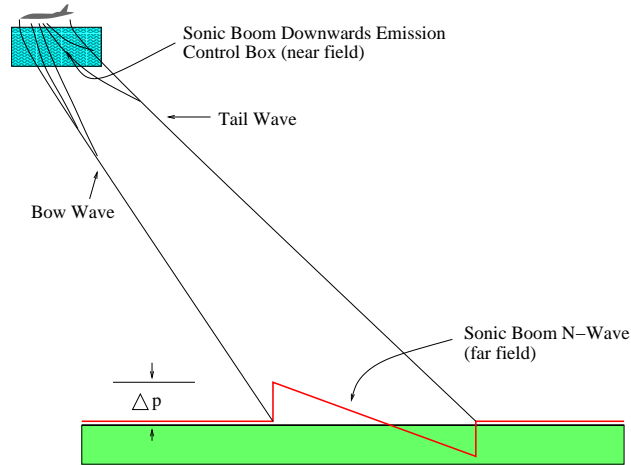


Figure 9: *Shock wave patterns on the near and far fields.*

go from the parameters γ that define the geometry of the plane to a measure of the sonic boom on the ground, the cost function $j(\gamma)$.

$$\begin{array}{cccccc}
 \text{Control} & & & \text{Euler} & \text{Pressure} & \text{Cost} \\
 \text{points} & \Rightarrow & \text{Geometry} & \Rightarrow & \text{shock} & \Rightarrow & \text{function} \\
 \gamma & & \Omega(\gamma) & & W(\gamma) & & \nabla p(W) & & j(\gamma)
 \end{array}$$

The intermediate steps are: the complete geometry $\Omega(\gamma)$ of the plane, the Euler flow $W(\gamma)$ around this geometry, and the pressure gradients $\nabla p(W)$ under the plane. The cost function $j(\gamma)$ actually combines the integral of the squared pressure gradients with the discrepancy from prescribed lift and drag coefficients.

We want to minimize $j(\gamma)$, using a gradient method, by iterative modifications of the shape parameters γ . This gradient descent is driven by $j'(\gamma)$, which we must compute.

Since the chain from γ to $j(\gamma)$ is actually a program, we can apply AD in the reverse mode to the complete program to get $j'(\gamma)$. However this is impractical due to the size of the program. Remember that the reverse mode, with the Store-All strategy, has a memory cost proportional to the execution time of the program. There are also questions about convergence: the original program solves the flow $W(\gamma)$ iteratively. It is not sure whether it makes sense to differentiate this complete iterative process.

We proposed in [19] an hybrid approach, where we only apply AD to selected parts of the original program, then use the differentiated pieces in a new, hand-coded solver, yielding the adjoint of the discretized flow equations and finally the gradient. A different approach is described in [13]. We go back to the mathematical equations. Basically, we consider a minimization problem under a particular additional constraint: the flow equation $\Psi(\gamma, W(\gamma)) = 0$, which expresses the dependence of the flow field $W(\gamma)$ on the shape γ . We thus want to find the γ_0 that minimizes the cost functional $j(\gamma) = J(\gamma, W(\gamma))$, under the constraint $\Psi(\gamma, W(\gamma)) = 0$. Here, this constraint is the compressible Euler equations, solved in a domain Ω_γ parametrized by γ . This minimization problem is solved using Lagrange multipliers. The problem's Lagrangian is

$$L(W, \gamma, \Pi) = J(\gamma, W) + \langle \Psi(\gamma, W), \Pi \rangle, \quad (13)$$

where Π is the *adjoint state*, which is a generalized Lagrange multiplier, and $\langle \cdot, \cdot \rangle$ is a suitable scalar product. Then the gradient $j'(\gamma)$ is found by solving

$$\begin{aligned} \Psi(\gamma, W(\gamma)) &= 0 \\ \nabla_W J(\gamma, W(\gamma)) - (\nabla_W \Psi(\gamma, W(\gamma)))^* \Pi(\gamma) &= 0 \\ j'(\gamma) &= \nabla_\gamma J(\gamma, W(\gamma)) - (\nabla_\gamma \Psi(\gamma, W(\gamma)))^* \Pi(\gamma). \end{aligned} \quad (14)$$

The first line gives $W(\gamma)$. The second line is the so-called *adjoint flow equation*, and gives the adjoint state $\Pi(\gamma)$. The last line gives the *gradient* $j'(\gamma)$ using $\Pi(\gamma)$ and $W(\gamma)$. This gradient will be used to update iteratively the former γ .

We then remark that, if we isolate the subprogram `Psi` of the flow solver program `P` that computes $\Psi(\gamma, W(\gamma))$, the reverse mode of AD can build automatically a new subprogram $\overline{\text{Psi}}_W$. This new subprogram, given any vector Π , returns the product $(\nabla_W \Psi(\gamma, W(\gamma)))^* \Pi$. Differentiation with respect to γ instead of W gives another subprogram $\overline{\text{Psi}}_\gamma$ that for any Π returns

$(\nabla_\gamma \Psi(\gamma, W(\gamma)))^* \Pi$. Similarly, if we isolate the subprogram J that computes the cost function $J(\gamma, W(\gamma))$, the reverse mode of AD automatically builds subprograms \bar{J}_W and \bar{J}_γ that respectively compute $\nabla_W J(\gamma, W(\gamma))$ and $\nabla_\gamma J(\gamma, W(\gamma))$.

With these subroutines generated, what remains to be done by hand to get $j'(\gamma)$ is the solver that solves the *adjoint flow equation* for Π . Notice that AD does not give us the matrix $(\nabla_W \Psi(\gamma, W(\gamma)))^*$ explicitly. Anyway this (2^{nd} -order) Jacobian matrix, although sparse, is too large for efficient storage and use. Therefore, we build a *matrix-free* linear solver. Fortunately, we just need to modify the algorithm developed for the flow solver P. P uses a simplified (1^{st} -order) Jacobian for preconditioning the pseudo-Newton time advancing. This matrix is stored. We just transpose this simplified Jacobian and reuse its Gauss-Seidel solver to build a preconditioned fixed-point iteration, that solves the adjoint flow equation. We validated the resulting gradient by direct comparison with divided differences of the cost function. The relative error is about 10^{-6} .

The overall optimization process is made of two nested loops. The outer loop evaluates the gradient, using an adjoint state as described above, then calls the inner loop which does a 1D search to get the steepest descent parameter, and finally updates the control parameters

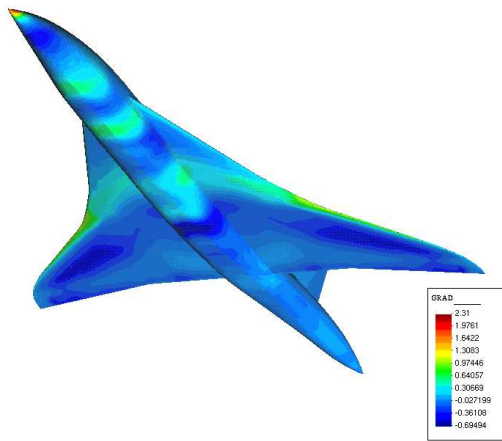


Figure 10: *Gradient of the Cost Functional on the skin of a jet*

We applied this optimization program on the shape of a Supersonic Business Jet, currently under development at Dassault Aviation. The mesh con-

sists of 173526 nodes and 981822 tetrahedra (for half of the aircraft). The inflow Mach number is 1.8 and the angle of attack is 3° . We target optimization of the wings of the aircraft only. Even then, the flow, the adjoint state, and the gradient $j'(\gamma)$ are computed taking into account the complete aircraft geometry. Figure 10 shows the gradient of our “sonic boom” cost functional j on the skin on the complete aircraft. Darker colors indicate places where modifying the shape strongly improves the sonic boom.

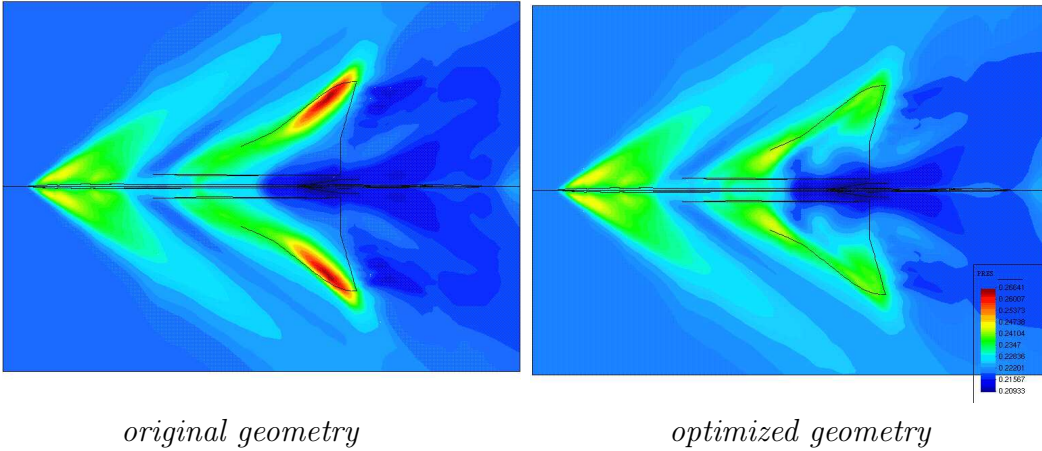


Figure 11: *Pressure distribution in a plane below the aircraft*

Figure 11 shows the evolution of the pressure on the near field, after 8 optimization cycles. We observe that the shock produced by the outboard part of the wings is dampened. However, within the Mach cone, close to the fuselage, the pressure peak has slightly increased after the optimization process. This increase is tolerable, compared to the reduction obtained on the end of the wings.

4 Static Data Flow Analyses for a better AD

There is an already very large collection of Data Flow analyses developed for compilers (standard, optimizing, or parallelizing). However, AD exhibits specific behaviors that require new data flow analyses. In this section, we describe these new analyses and show their interest. These analyses can be formalized cleanly and implemented efficiently on programs internally kept as Flow Graphs, using so-called “*data flow equations*”. These equations can even

be used to prove desirable properties of these analyses, such as termination or optimality to some extent.

We shall focus on “*activity*” analysis, which is central to all modes of AD, in section 4.1. Section 4.2 focuses on an analysis for the reverse mode, called “*TBR*”. Section 4.3 says a word about the consequences of “*aliasing*” in reverse AD. In section 4.4, we present extensions to “*dead code*” detection, that can also improve AD-generated code.

The sequel will show example pieces of differentiated code, using the notations and conventions of our AD tool TAPENADE. Let’s present these conventions briefly.

First consider symbol names. If a variable v is of differentiable type (e.g. a floating point real), and currently has a non-trivial derivative (cf *activity* 4.1), 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” in *tangent* mode and “_b” in reverse mode. Figure 12 summarizes that.

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

Figure 12: *Names of differentiated symbols*

Now consider an assignment I_k . In *tangent* mode (cf equation (4)), derivative instruction \dot{I}_k implements $\dot{W}_k = f'_k(W_{k-1}) \cdot \dot{W}_{k-1}$, with initial $\dot{W}_0 = \dot{X}$. In *reverse* mode (cf equation (5)), derivative instruction(s) \overleftarrow{T}_k implements $\overleftarrow{W}_{k-1} = f_k'^*(W_{k-1}) \cdot \overleftarrow{W}_k$, with initial $\overleftarrow{W}_p = \overleftarrow{Y}$. TAPENADE, like the other AD tools, tries to keep the original program’s structure: just like the original program *overwrites* variables, the differentiated program overwrites the differentiated variables, writing values \dot{W}_k over previous values \dot{W}_{k-1} in tangent

mode, or writing values \overline{W}_{k-1} over previous values \overline{W}_k in the reverse mode. For example, if I_k is $a(i)=x*b(j) + \text{COS}(a(i))$, it is straightforward to write the Jacobian of the corresponding function

$$f_k : \begin{matrix} \mathbb{R}^3 & \rightarrow & \mathbb{R}^3 \\ \begin{pmatrix} \dot{a}(i) \\ \dot{b}(j) \\ \dot{x} \end{pmatrix} & \mapsto & \begin{pmatrix} \dot{a}(i) \\ \dot{b}(j) \\ \dot{x} \end{pmatrix} \end{matrix}$$

which is the matrix

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

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{T}_k must implement:

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

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

and therefore an AD tool would produce the derivative instructions shown on figure 13.

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

Figure 13: *Differentiation of a simple assignment*

4.1 Activity Analysis and reinitializations of derivatives

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.

Activity analysis [17] is the specific analysis that detects these situations, therefore allowing for a better differentiated code. *Activity analysis* is present in all transformation-based AD tools.

To begin with, the end-user specifies 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 when variable v is not varied, then whatever the value of \bar{v} , this value does not matter for the final result.

Activity analysis is global, 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 , it needs 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*

original program	tangent AD	reverse AD
<code>x = 1.0</code>	<code>x = 1.0</code>	<code>x = 1.0</code>
<code>z = x*y</code>	<code>zd = x*yd</code>	<code>z = x*y</code>
<code>t = y**2</code>	<code>z = x*y</code>	<code>t = y**2</code>
<code>IF (t .GT. 100) ...</code>	<code>t = y**2</code>	<code>IF (t .GT. 100) ...</code>
	<code>IF (t .GT. 100) ...</code>	<code>...</code>
		<code>yb = yb + x*zb</code>

Figure 14: *Instructions simplifications due to Activity Analysis*

Figure 14 illustrates the benefits of activity analysis. `x` immediately becomes not varied, and `t` is useless. Therefore, the AD tool 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, the tool 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, the tool removes them automatically, which may be surprising.

4.2 TBR Analysis

In section 2.3, we saw that the main drawback of the reverse mode of AD is the memory consumption to store intermediate values before they are overwritten during the forward sweep. Although checkpointing is a general answer to this problem, it is advisable to restrict this storage to intermediate values that are *really* needed by the backward sweep. Consider for example an assignment $x = a+2*b$. The partial derivatives of the sum do not use the values of a nor b . Therefore, as far as this instruction is concerned, there is no need to store the values of a nor b in case they get overwritten in the forward sweep. This is the purpose of the TBR analysis [9, 24, 17], which analyses the program to find which variables are *To Be Recorded*, and which are *not*.

In the example of figure 15, TBR analysis could prove that neither x nor y were needed by the differentiated instructions, and therefore these variables need not be PUSH'ed on nor POP'ed from the stack.

original program	reverse mode: naive backward sweep	reverse mode: backward sweep TBR
$x = x + \text{EXP}(a)$ $y = x + a**2$ $a = 3*z$	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) $ab = ab + \text{EXP}(a)*xb$	CALL POPREAL4(a) $zb = zb + 3*ab$ $ab = 0.0$ $ab = ab + 2*a*yb$ $xb = xb + yb$ $yb = 0.0$ $ab = ab + \text{EXP}(a)*xb$

Figure 15: *Removing unnecessary storage through To Be Recorded analysis*

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*. An AD tool must detect this situation and issue a warning message. This message should not be overlooked, because it may point to a future problem in the differentiated code, especially in the reverse mode.

original program	reverse AD: forward sweep	reverse AD: 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 16: *Detection and correction of aliasing in the reverse mode*

Figure 16 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 16, there is a local aliasing in the third instruction, because equality between `i` and `n-i` could not be decided.

4.4 Dead Adjoint Code

Reverse differentiation of the program P that computes function F yields program \bar{P} that computes the gradient of F . The original results of P , which are also computed by the forward sweep of \bar{P} , are *not* a result of \bar{P} . Only the

original program	reverse AD:	reverse AD: dead adjoint removed
<pre> IF (a.GT.0.0) THEN a = LOG(a) ELSE a = LOG(c) CALL SUB(a) ENDIF END </pre>	<pre> 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 </pre>	<pre> 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 </pre>

Figure 17: *Removing useless dead adjoint code*

gradient is needed by the end-user. Moreover in most implementations the original results will be overwritten and lost during the backward sweep of \bar{P} . Therefore some of the last instructions of the forward sweep of \bar{P} are actually dead code. An AD tool may use a slicing analysis, called *Adjoint Liveness analysis*, to remove this dead adjoint code. The example on figure 17 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.

4.5 Undecidability

There is a theoretical limit to the precision of static analyses, called *undecidability*. It states that is is impossible to design a program which, given any program as an argument, will determine whether this argument program terminates or not. There exist a variety of corollaries. For instance, one can't write a software tool that finds out if two programs are equivalent, giving the same results in response to the same inputs. A program can't even decide that any two expressions of the analyzed program always evaluate to the same result.

The consequence is that most static analyses that must answer to a “boolean” question on the given program will give one of the three answers “yes”, “no”, or “*I can’t tell*”. Obviously, if the answer is “*I can’t tell*”, the tool that asked this question must be prepared to take the good decision. This decision must be *conservative*, i.e. it must produce a transformed program or an answer which is correct, whatever the run-time arguments. In other words, the tool that asked this question and got the “*I can’t tell*” answer must take no chances: it must continue with its work considering that the run-time answer can be “yes” or “no” (there is no other possibility at run-time).

Notice that undecidability is an absolute but very far barrier. In practice other barriers will arise much sooner than undecidability, which have little relation with it. For example comparison of expressions in array indexes can become very complex [7] and tools generally give up when the expressions are not linear with respect to the loop index. Array indexes can also involve indirection arrays, i.e. other arrays with unknown values. Comparison would require an enumeration of all possible cases, which is of course out of reach practically. Also, deciding whether two pointers reference the same memory location would require to enumerate all possible flows of control, and this is also practically out of reach.

Undecidability should not discourage anyone from implementing analyzers: in a sense, it just means that a given analyzer can always be improved, asymptotically!

5 AD Tools and TAPENADE

Before we present the user interface of TAPENADE, let us mention the other AD tools that we know, and our (partial) vision of how one might classify them. Maybe a better source is the www.autodiff.org site for the Automatic Differentiation community, managed by our colleagues in Aachen and Argonne.

As we saw, some AD tools rely on program overloading rather than program transformation. In general this makes the tool easier to implement. However some overloading-based AD tools can become very sophisticated and efficient, and represent a fair bit of hard work too. Overloading-based AD tools exist only for target languages that permit some form of overloading, e.g. C++ and Fortran95. Overloading-based AD-tools are particularly

adapted for differentiations that are mostly local to each statement, i.e. no fancy control flow rescheduling is allowed. On the other hand, these local operations can be very complex, more than what transformation-based AD tools generally provide. For instance, overloading-based AD-tools can generally compute not only first, but also second, third derivatives and so on, as well as Taylor expansions or interval arithmetic. **Adol-C** [1], from TU Dresden, is an excellent example of overloading-based AD tool. **FADBAD/TADIFF** are other examples.

The AD tools based on program transformation parse and analyze the original program and generate a new source program. TAPENADE is one of those. These tools share their general architecture, with a front-end very much like a compiler, followed by an analysis component, a differentiation component, and finally a back-end that regenerates the differentiated source. They differ in particular in the language that they recognize and differentiate, the AD modes that they provide. They also exhibit some differences in AD strategies mostly about the reverse mode. The best known other transformation-based AD tools are the following:

- Adifor [3, 5] differentiates Fortran77 codes in tangent mode. Adifor once was extended towards the reverse mode (Adjfor), but we believe this know-how has now been re-injected into the OpenAD framework, described below.
- Adic can be seen as the C equivalent of Adifor. However it is based on a completely different architecture, from the OpenAD framework. This framework, very similar to TAPENADE's architecture, claims that only front-end and back-end should depend on the particular language, whereas the analysis and differentiation part should work on a language-independent program representation. Adic differentiates ANSI C programs in tangent mode, with the possibility to obtain second derivatives.
- OpenAD/F [25] differentiates Fortran codes in tangent and reverse modes. Its strategy to restore intermediate values in reverse AD is extremely close to TAPENADE's. OpenAD/F is made of Adifor and Adjfor components integrated into the OpenAD framework.
- TAMC [11], and its commercial offspring TAF differentiate Fortran files. TAF also differentiates Fortran95 files, under certain restrictions.

TAF is commercialized by the FastOpt company in Hamburg, Germany. TAF differentiates in tangent and reverse mode, with the recompute-all approach to restore intermediate values in reverse AD. Checkpointing and an algorithm to avoid useless recomputations (ERA) are used to avoid explosion of run-time. TAF also provides a mode that efficiently computes the sparsity pattern of Jacobian matrices, using bit-sets.

- TAC++ [11] is the C version of TAF. It is also developed by FastOpt. Presently, TAC++ only handles a large subset of C, and it is still in its development phase, although making significant progress. Like TAF, it will provide tangent and reverse modes, with the same strategies, e.g. the recompute-all approach with checkpointing for the reverse mode.


There are also AD tools that directly interface to an existing compiler. In fact, these are extensions to the compiler so that differentiated code is added at compile time. For instance the NAGWare Fortran95 compiler has AD facilities inside, that are triggered by user directives in the Fortran source. It so far provides tangent-mode differentiation only.

There are AD tools that target higher-level languages, such as MATLAB. We know of ADiMat, MAD, and INTLAB. Even when they rely on operator overloading, they may embed a fair bit of program analysis to produce efficient differentiated code.

5.1 TAPENADE

Here are more details on the Automatic Differentiation tool TAPENADE [18], which is developed by our research team. TAPENADE progressively implements the results of our research about models and static analyses for AD. Development of TAPENADE started in 1999. It is the successor of ODYSSEE [10], a former AD tool developed by INRIA and universit  de Nice from 1992 to 1998. TAPENADE and ODYSSEE share some fundamental concepts. TAPENADE is distributed by INRIA. At present, we are aware of regular industrial use of TAPENADE at Dassault Aviation, CEA Cadarache, Rolls-Royce (UK), BAe (UK), Alenia (Italy), Cargill (USA). Academic colleagues use it on a regular basis at INRA (French agronomy research center), Oxford (UK), Argonne National Lab (USA). TAPENADE is still under permanent development, and figure 18 summarizes its present state.

TAPENADE has the following two principal objectives:



Automatic Differentiation Tool

Name: TAPENADE version 2.2
Date of birth: January 2002
Ancestors: Odyssee 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, Dead Adjoint Code, TBR

Figure 18: TAPENADE's ID card

- To serve as an implementation platform to experiment with new modes of AD or to validate new AD algorithms and analyses. Ideally, this experimentation could also take place outside of our research team.
- To provide external end-users, academic or industrial, with state-of-the-art AD of real programs, with no restriction on the style or on the size of the application to differentiate.

The architecture of TAPENADE was designed according to the following guidelines:

- Tools that work on programs must have an internal representation which is convenient for analysis more than for mere edition. Therefore, instead of syntax trees, we prefer Call Graphs or Flow Graphs.
- The internal representation must concentrate on the semantics of the program and eliminate or normalize what is less essential. In particular, the internal representation should be independent of the particular programming language used by the program.
- The internal representation must facilitate data flow analysis. Internal representation of variables is essential, and must be chosen very carefully. In particular, it must once and for all solve the tedious array region management due to constructs such as the Fortran `COMMON` and `EQUIVALENCE`.
- Many program analyses are indeed independent of Automatic Differentiation, and are identical to those needed by parallelizers. These analyses must form a fundamental layer, clearly separated from the AD specific part which is built above it.
- It is not necessary that a tool be restricted to mini-languages. Experience shows that real programs use all possibilities of their implementation language. If we want a usable tool, we must take care of every construct or possibility offered by the language. Indeed, this is far less boring than one could fear!

Figure 19 summarizes the architecture of TAPENADE, built after these guidelines. To enforce a clear separation from the language of the programs, we devised an Intermediate Language (IL), which should eventually contain all

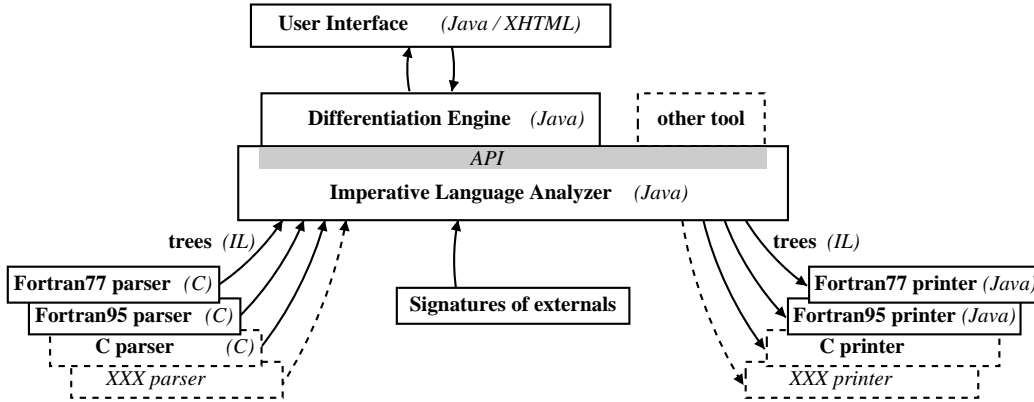


Figure 19: *Overall Architecture of TAPENADE*

the syntactic constructs of imperative languages. IL has no concrete syntax, i.e. no textual form. It is an abstract syntax, that strives to represent by the same construct equivalent concrete constructs from different languages. IL currently covers all Fortran77, Fortran95, and C. Object-oriented constructs are progressively being added. The front-end for a given language, and the corresponding back-end, are put outside of TAPENADE, and communicate with it by a simple tree transfer protocol.

Real programs often use external libraries, or more generally “black-box” routines. The architecture must cope for that, because static analyses greatly benefit from compact, summarized information on these black-box routines. The end-user can provide TAPENADE with these signatures of black-box routines in a separate file.

The Imperative Language Analyzer performs general purpose analyses, independent from AD. Figure 20 shows these analyses with their relative dependencies. All of them run on Flow Graphs, according to their formal description as data flow equations. The results of these general analyses are of course used later by the AD level (*cf* figure 21), which performs Differentiable Dependency Analysis followed by Activity Analysis (section 4.1) and, specifically for the reverse mode, TBR Analysis (section 4.2) and Dead Adjoint Code detection (section 4.4).

TAPENADE comes as a set of JAVA classes, from a source 86000 lines long.

TAPENADE can be downloaded and installed on a local computer from our ftp server:

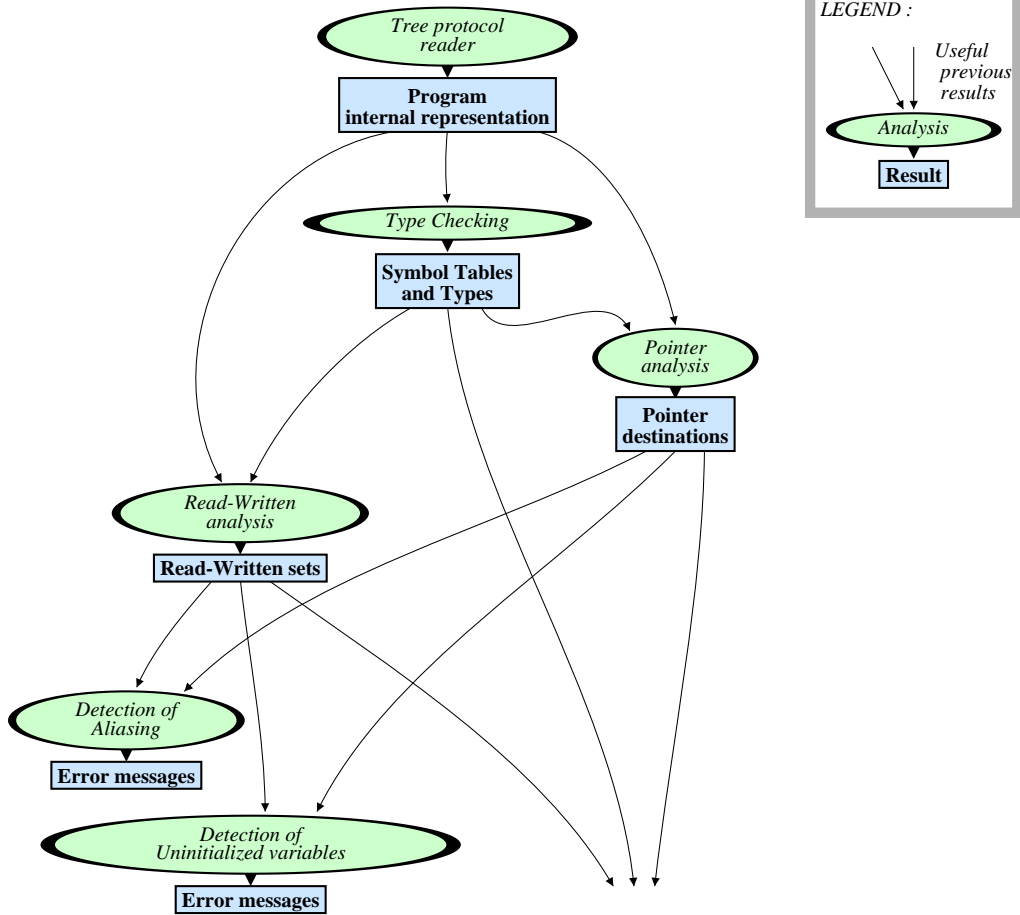


Figure 20: Ordering of generic static data flow analyses in TAPENADE

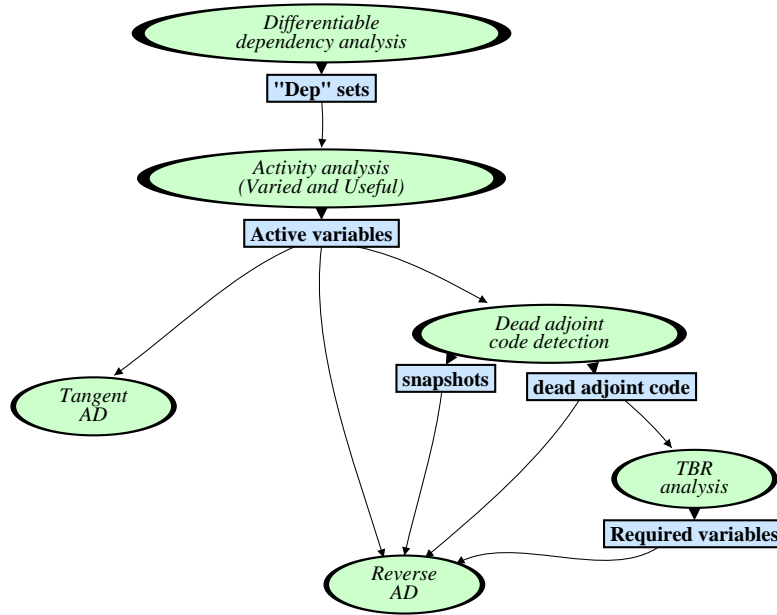


Figure 21: Ordering of AD static data flow analyses in TAPENADE

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

and then it can be called directly as a command with arguments, e.g. from a Makefile, or through a graphical user interface. The TAPENADE executable exists for both Linux and Windows systems.

Alternatively, TAPENADE can be used directly as a web server by a distant user, without any need for a local installation. The server address is:

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

which takes you to the TAPENADE input page shown on figure 22. 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.

A moment after the user clicked on one AD mode button which triggered differentiation, the TAPENADE server sends the differentiation output in a new web page, shown on figure 23. This graphical user interface helps examine TAPENADE output, highlighting 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`.

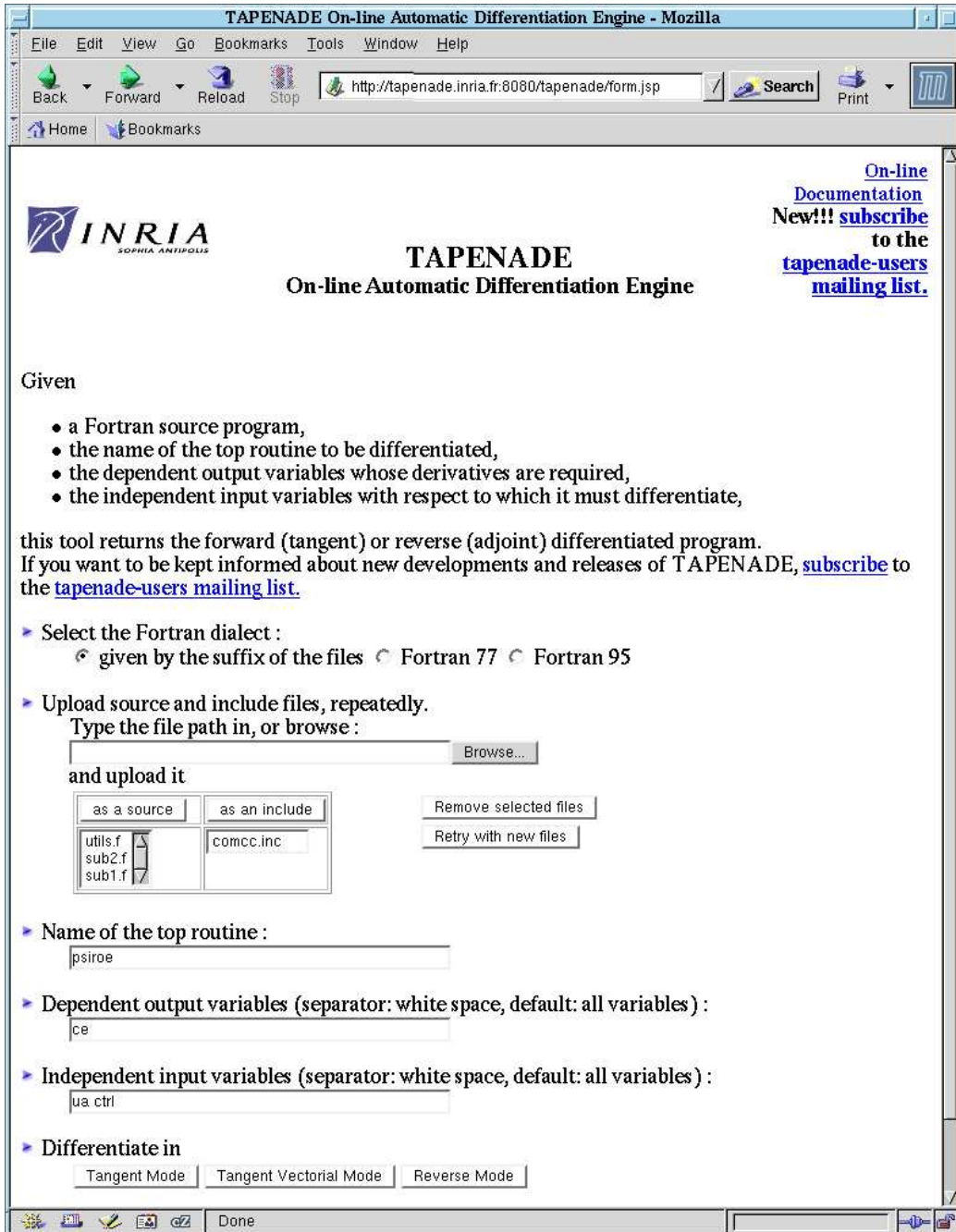


Figure 22: *HTML interface for TAPENADE input*

The screenshot displays the TAPENADE differentiation result in a Mozilla browser window. The interface is divided into two main panels: 'Original call graph' and 'Differentiated call graph'. The 'Original call graph' shows a hierarchical tree structure starting with 'psiroe', which branches into 'conddirflux', 'transpiration', 'normcoq', 'vcurvm', 'fluroe', and 'gradnod'. The 'Differentiated call graph' shows a similar structure with differentiated sub-nodes: 'gradnod_b', 'gradfb_b', 'transp_b', 'fluroe_b', 'vcurvm_b', 'transpiration_b', and 'normcoq_b'. Below these graphs, the original Fortran code is shown on the left, and the differentiated code is on the right. The original code includes a loop for calculating nodal gradients. The differentiated code shows the same logic with additional calls to differentiation functions like 'CALL POPREAL8' and 'CALL TRANSB'. At the bottom, there are several error messages related to type mismatches and undeclared functions.

Figure 23: *HTML interface for TAPENADE output*

To replace TAPENADE in the context of AD tools, let's describe its actual differentiation model. TAPENADE performs AD by program transformation instead of overloading. As could be expected, the primary goals of TAPENADE are first derivatives, through the tangent and reverse modes of AD. We are planning extensions for second-order derivatives, but this is still research at this time.

To produce a better code, TAPENADE implements all the data-flow analyses that we described in sections 4.1, 4.2, and 4.4 or pictured on figures 20 and 21. Whereas there is not much to add about the tangent mode, the reverse mode basically relies on a Store-All strategy (*cf* figure 4), implemented through primitives that PUSH and POP values on/from a stack. As we mentioned in section 2, this strategy cannot give good results on large programs without the storage/recomputation trade-off known as *checkpointing*. The default strategy of TAPENADE is therefore to apply checkpointing to each procedure call. Figure 24 illustrates this strategy on a simple call tree. Execution of a procedure A in its original form is shown as A. The *forward*

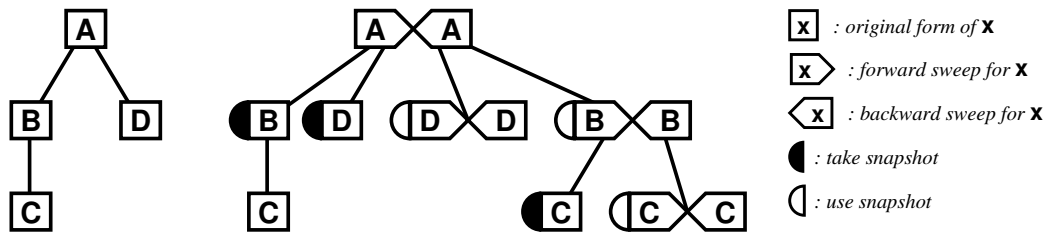


Figure 24: *Checkpointing on calls in TAPENADE reverse AD*

sweep, i.e. execution of A augmented with storage of variables on the stack just before they are overwritten, is shown as \overrightarrow{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{A} . For each procedure call, e.g. B, the procedure is run *without* storage during the enclosing forward sweep \overrightarrow{A} . When the backward sweep \overleftarrow{A} reaches B, it runs \overrightarrow{B} , i.e. B again but this time with storage and then immediately it runs the backward sweep \overleftarrow{B} and finally the rest of \overleftarrow{A} . Duplicate execution of B requires that some variables used by B (a *snapshot*) be stored.

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.

Since call trees are rarely well-balanced in real programs, TAPENADE provides a way to control checkpointing, through user-given directives that designate the procedure calls that must not be checkpointed.

6 Validation of Differentiated Programs

Recalling the notations of section 2, we differentiate a program P that computes a function F , with input $X \in \mathbb{R}^m$ and output $Y = F(X) \in \mathbb{R}^n$.

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 $x \in \mathbb{R}$:

$$f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \quad (15)$$

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 (15) (on the left hand side) and with the chain rule (on the right hand side) tell us that

$$\lim_{\varepsilon \rightarrow 0} \frac{F(X + \varepsilon \times \dot{X}) - F(X)}{\varepsilon} = g'(0) = F'(X) \times \dot{X} = \dot{Y} \quad (16)$$

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} :

$$(\overline{X} \cdot \dot{X}) = (F'^*(X) \times \dot{Y} \cdot \dot{X}) = \dot{Y}^* \times F'(X) \times \dot{X} = \dot{Y}^* \times \dot{Y} = (\dot{Y} \cdot \dot{Y}) \quad (17)$$

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

7 Conclusion

We have presented Automatic Differentiation, and more precisely the fundamental notions that are behind the AD tools that use source program transformation. We shall use figure 25 as a visual support to compare AD with other ways to obtain derivatives. Our strongest claim is that if you need

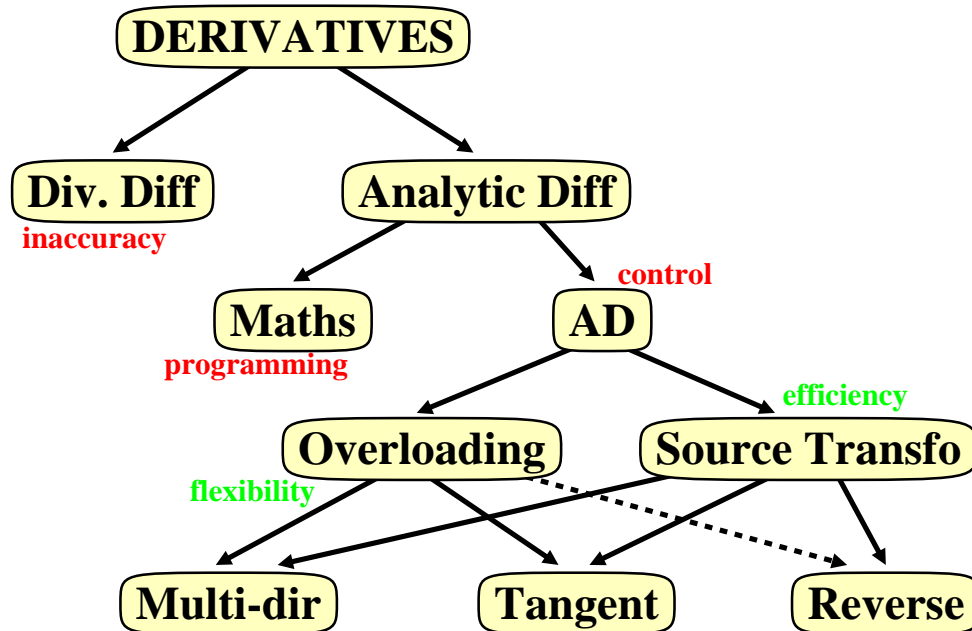


Figure 25: *AD and other ways to compute derivatives*

derivatives of functions that are already implemented as programs, then you should seriously consider AD. At first thought, it is simpler to apply the Divided Differences method (sometimes known also as “Finite Differences”), but its inaccuracy is its major drawback.

Notice that Divided Differences sometimes behave better when the implemented function is not differentiable, because its very inaccuracy has the effect of smoothing discontinuities of the computed function. Therefore Di-

vided Differences can be an option when one only has piecewise differentiability, like shown in section 2.1. Also, it is true that Divided Differences may actually cost a little less than the tangent mode, which is their AD equivalent.

Nevertheless when it is possible, it is probably safer to look for exact analytical derivatives. Divided Differences are definitely not good if you need gradients or if you need higher order derivatives.

Then again two options arise: one can consider the problem of finding the derivatives as a new mathematical problem, with mathematical equations that must be discretized and solved numerically. This is a satisfying mathematical approach, but one must consider its development cost.

AD is a very promising alternative when the function to be differentiated has already been implemented. In a sense, AD reuses the resolution strategy that has been implemented for the original function into the resolution of its derivatives. When the original model or code changes, AD can be applied again at very little cost.

There are still open questions of non-differentiability introduced by the program's control, or the fact that the iterative resolution of the derivatives is not guaranteed to converge at the same speed as the original function resolution. But in practice, AD returns derivatives that are just as good as those returned by the "mathematical" approach above.

Inside the AD methods we distinguish Overloading-based approaches, which are more flexible and can be adapted to all sorts of derivatives and similar concepts. On the other hand, we advocate Source-Transformation-based tools for the well-identified goals of tangent and reverse first-order derivatives. Source transformation gives it full power when it performs global analyses and transformations on the code being differentiated.

Source Transformation AD is really the best approach to the reverse mode of AD, which computes gradients at a remarkably low cost. Reverse AD is a discrete equivalent of the adjoint methods from control theory. Reverse AD may appear puzzling and even complex at first sight. But AD tools apply it very reliably so that a basic understanding of it generally suffices. Reverse AD is really the choice method to get the gradients required by inverse problems (e.g. data assimilation) and optimization problems.

AD tools can build highly optimized derivative programs in a matter of minutes. AD tools are making progress steadily, but the best AD will always require end-user intervention. Moreover, one must keep in mind the limitations of AD in order to make a sensible usage of it. Fundamentally:

- real programs are always only piecewise differentiable, and only the user can tell if these algorithmic discontinuities will be harmful or not.
- iterative resolution of the derivatives may not converge as well as the original program, and the knowledge of the Numerical Scientist is invaluable to study this problem.
- reverse AD of large codes will always require the knowledge of a Computer Scientist to implement the best storage/recomputation trade-off.

There are also a number of technical limitations to AD tools, which may be partly lifted in the future, but which are the current frontier of AD tool development:

- pointers are still hard to analyze, and memory allocation in the original program is a challenge for the memory restoration mechanism of the reverse mode.
- object-oriented languages pose several very practical problems, because they far more intensively use the mechanisms of overloading and dynamic allocation. Data-Flow analysis of object-oriented programs may become harder and return less useful results.
- parallel communications [8] or other system-related operations may introduce a degree of randomness in the control flow, which is then hard to reproduce, duplicate, or reverse.

References

- [1] Adol-C. <http://www.math.tu-dresden.de/~adol-c>.
- [2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [3] C. Bischof, A. Carle, P. Khademi, and A. Maurer. The adifor 2.0 system for automatic differentiation of fortran77 programs. *IEEE Comp. Sci. & Eng.*, 3(3):18–32, 1996.

- [4] 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.
- [5] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.
- [6] 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.
- [7] B. Creusillet and F. Irigoien. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.
- [8] P. Dutto, C. Faure, and Fidanova S. Automatic differentiation and parallelism. In *Proceedings of Enumath 99, Finland*, 1999.
- [9] C. Faure and U. Naumann. Minimizing the tape size. In *in [6]*, 2001.
- [10] Ch. Faure and Y. Papegay. Odyssee 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. Technical report, 1997. <http://www.autodiff.com/tamc>.
- [12] J.C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.
- [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*, 1:35–54, 1992.
- [15] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Frontiers in Applied Mathematics, 2000.

- [16] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications and Tools*, pages 95–106. SIAM, 1996. rapport de Recherche INRIA 2794.
- [17] L. Hascoët, U. Naumann, and V. Pascual. Tbr analysis in reverse mode automatic differentiation. *Future Generation Computer Systems – Special Issue on Automatic Differentiation*, 2004. *to appear*.
- [18] L. Hascoët and V. Pascual. Tapenade 2.1 user’s guide. Technical report 300, INRIA, 2004.
- [19] L. Hascoët, M. Vázquez, and A. Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V. Kumar et al., editor, *Proceedings of the International Conference on Computational Science and its Applications, ICCSA ’03, Montreal, Canada*, pages 85–94. LNCS 2668, Springer, 2003.
- [20] W. Kyle-Anderson, J. Newman, D. Whitfield, and E. Nielsen. Sensitivity analysis for navier-stokes equations on unstructured meshes using complex variables. *AIAA Journal*, 39(1):56–63, 2001.
- [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] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.
- [24] 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.
- [25] OpenAD. <http://www.mcs.anl.gov/OpenAD>.