# Reversal Strategies for Adjoint Algorithms

Laurent Hascoët

June 4, 2007

**Abstract**

Adjoint Algorithms are a powerful way to obtain the gradients that are needed in Scientific Computing. Automatic Differentiation can build Adjoint Algorithms automatically by source transformation of the direct algorithm. The specific structure of Adjoint Algorithms strongly relies on reversal of the sequence of computations made by the direct algorithm. This reversal problem is at the same time difficult and interesting. This paper makes a survey of the reversal strategies employed in recent tools and describes some of the more abstract formalizations used to justify these strategies.

## 1 Why build Adjoint Algorithms?

Gradients are a powerful tool for mathematical optimization. The Newton method for example uses the gradient to find a zero of a function, iteratively, with an excellent accuracy that grows quadratically with the number of iterations. In the context of optimization, the optimum is a zero of the gradient itself, and therefore the Newton method needs second derivatives in addition to the gradient. In Scientific Computing the most popular optimization methods, such as BFGS [16], all give best performances when provided gradients too.

In real-life engineering, the systems that must be simulated are complex: even when they are modeled by classical mathematical equations, analytic resolution is totally out of reach. Thus, the equations must be discretized on the simulation domain, and then solved e.g. iteratively by a computer algorithm.

Optimization comes into play when, after simulating the system for a given set of input parameters, one wants to modify these parameters in order to minimize some cost function defined on the simulation's result. The mathematical local optimization approach requires a gradient of the cost with respect to the input parameters. Notice furthermore that the gradient of the simulated function has several other applications. To quote just one, the gradient characterizes the sensitivity of the system simulation to small variations or inaccuracies of the input parameters.

How can we get this gradient? One can write a system of mathematical equations whose solution is the gradient, and here again analytic resolution is out of reach. Therefore, one must discretize and solve these equations, i.e. do what was done for the original equations. There is however an alternative approach that takes the algorithm that was built to solve the original system, and transforms it into a new *adjoint algorithm* that computes the gradient. This can be done at a relatively low development cost by *Algorithmic Differentiation*, also known as *Automatic Differentiation (AD)* of algorithms [8, 3, 1].

The fundamental observation of AD is that the original program `P`, whatever its size and run time, computes a function $F, X \in \mathbb{R}^m \mapsto Y \in \mathbb{R}^n$ which is the composition of the elementary functions computed by each run-time instruction. In other words if `P` executes a sequence of elementary statements $I_k, k \in [1..p]$, then `P` actually evaluates

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

where each $f_k$ is the function implemented by $I_k$. Therefore one can apply the chain rule of calculus to get the *Jacobian* matrix $F'$, i.e. the partial derivatives of each component of $Y$ with respect to each component of $X$. Calling $X_0 = X$ and $X_k = f_k(X_{k-1})$ the successive values of each intermediate variable, i.e. the successive *states* of the memory, throughout execution of `P`, we get

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

Recalling now that we are looking for a gradient, which implies strictly speaking that $X_p = Y$ is scalar, we see that Equation (1) is more efficiently computed from left to right because vector×matrix products are so much cheaper

than matrix×matrix. We end up with an iterative *adjoint* algorithm which, for each statement $I_k$ for $k = p$ down to 1, i.e. in *reverse order*, executes an adjoint code $\overleftarrow{I_k}$ that computes $\overline{X}_{k-1} = \overline{X}_k \times f'_k(X_{k-1})$. In other words, $\overline{X}_k$ is indeed the gradient of the final scalar cost with respect to the variables $X_k$ just before $I_k$, and finally $\overline{X}_0$ is the required gradient $\overline{X}$. For every variable `x` in every $X_k$, we thus define $\overline{\texttt{x}}$ which is the gradient of the final scalar cost with respect to this `x`, and we will call it the "*gradient on* `x`" for short.

Before looking further into the problems posed by the AD adjoint algorithm, let's underline its decisive advantage. Observing that the cost of $\overleftarrow{I_k}$ is only a small multiple of the cost of the original statement $I_k$, with a factor generally between 2 and 4, we see that we get the gradient at a cost which is a small multiple of the cost of `P`. This cost is independent from the dimension $m$ of the input parameter space. If on the other hand one computes the gradient by evaluating Equation (1) from right to left, one repeatedly multiplies a matrix $f'_k(X_{k-1})$ by another matrix with $m$ columns. This is called the AD *tangent* algorithm, and its cost is proportional to $m$. In real applications the number $m$ of optimization parameters can range from several hundred up to several million, and the tangent algorithm is no longer an option.

However, the adjoint algorithm needs and uses the $f'_k(X_{k-1})$ in the reverse order, from $k = p$ down to 1. We call this the program *reversal* problem. For instance the adjoint algorithm first needs $X_{p-1}$, which in turn requires execution of nearly all the original program `P`. Then the adjoint algorithm needs $X_{p-2}$, but going from $X_{p-1}$ back to $X_{p-2}$ is by no means easy. One needs reversal strategies based either on a new run of (a slice of) `P`, or on undoing statement $I_{p-1}$ possibly using some clever preliminary storage of values before $I_{p-1}$.

This paper makes a survey of the reversal strategies employed in the most recent AD tools, with some emphasis on the strategies that we implemented and sometimes designed for our AD tool Tapenade. In the sequel, we will organize the reversal problems into reversal of individual statements in Section 2, reversal of the Data-Flow in Section 3, and reversal of the Control-Flow in Section 4.

# 2 Reversal of Individual Statements

Consider a statement $I_k$ from the original simulation program `P`. We can focus without loss of generality on the case where $I_k$ is an assignment. Control statements will be dealt with in Section 4, procedure calls are out of the scope of this paper and can be considered inlined as a sub-sequence of statements, and other statements such as I-O statements are of limited effect on gradient computations.

The derivative adjoint code $\overleftarrow{I_k}$ that computes $\overline{X}_{k-1} = \overline{X}_k \times f'_k(X_{k-1})$ is somewhat different from the usual derivatives from textbooks, because the gradients are propagated backwards. Specifically, naming `y` the variable overwritten by $I_k$, and considering each variable `x` used in its the right-hand side

$$I_k \ : \quad \texttt{y}{=}f_k(\ldots\texttt{x}\ldots)$$

considering, for the sake of clarity only, that `x` and `y` are distinct program variables, the adjoint $\overleftarrow{I_k}$ performs for each `x`

$$\overline{\texttt{x}} = \overline{\texttt{x}} + \frac{\partial f_k}{\partial \texttt{x}} * \overline{\texttt{y}}$$

and terminates resetting $\overline{\texttt{y}}$=`0.0`. For example if $I_k$ is:

```
y = x*(a(j)+1.0)
```

its adjoint code is:

```
x̄ = x̄ + (a(j)+1.0)*ȳ
ā(j) = ā(j) + x*ȳ
ȳ = 0.0
```

Let us now focus on the problem of common sub-expressions in the derivative code. Consider an example assignment:

```
res = (tau-w(i,j))*g(i,j)*(z(j)-2.0)/v(j)
```

Its adjoint code is

```
z̄(j) = z̄(j) + g(i, j)*(tau-w(j))*r̄e̅s̄/v(j)
v̄(j) = v̄(j) - g(i, j)*(tau-w(j))*(z(j)+2.0)*r̄e̅s̄/v(j)**2
ḡ(i, j) = ḡ(i, j) + (z(j)+2.0)*(tau-w(j))*r̄e̅s̄/v(j)
t̄a̅u̅ = t̄a̅u̅ + (z(j)+2.0)*g(i, j)*r̄e̅s̄/v(j)
w̄(j) = w̄(j) - (z(j)+2.0)*g(i, j)*r̄e̅s̄/v(j)
r̄e̅s̄ = 0.0
```

We see that differentiation has introduced many common sub-expressions that slow down the derivative code. This is because the $\frac{\partial f_k}{\partial \ldots}$ often share sub-expressions from $f_k$. This problem differs from the optimal accumulation of partial derivatives addressed by Naumann in [13], which he recently proved to be NP-complete [14]. Instead of looking for the optimal combination of partial derivatives across several successive statements, we are here looking for an optimal common sub-expressions elimination among the expressions that return these partial derivatives for one statement. Instead of leaving this problem to some post-processor, we are going to use the known structure of adjoint codes to eliminate common sub-expressions right from differentiation time. This elimination is governed by cost/benefit considerations and therefore we analyze the cost of the adjoint code of $I$, looking at the abstract syntax tree of $I$.

Gradients are computed backward: for any arbitrary sub-tree $S$ of the right-hand side, we can define $\overline{S}$ the gradient on the result of $S$, which is the product of the gradient on the right-hand side `r̄e̅s̄` with the partial derivative of `res` with respect to $S$. Thus, $\overline{S}$ is computed independently of the inside of $S$, and it is used to compute the gradient on each variable that occurs in $S$. For example if we take $S$ to be `(z(j)-2.0)/v(j)`, $\overline{S}$ is `g(i,j)*(tau-w(i,j))*r̄e̅s̄`, and it occurs twice in the adjoint code, to compute `z̄(j)` and `v̄(j)`. Let us evaluate, for each sub-tree $S$ (*resp.* for its corresponding $\overline{S}$), its evaluation cost $c$ (*resp.* $\overline{c}$) and the number of times $t$ (*resp.* $\overline{t}$) it occurs in the adjoint code:

- $c$ is obviously a simple synthesized (bottom-up) attribute.
- $\overline{t}$ is in fact the number of variables inside $S$, because the gradient on each of these variables uses $\overline{S}$, and no one else does. It is therefore a very simple synthesized attribute.
- $\overline{c}$ is an inherited (top-down) attribute: if $S$ is a child of an expression $P = op(\ldots, S, \ldots)$, then the cost of $\overline{S}$ is the cost of $\overline{P}$, plus one product,

plus the cost of $\frac{\partial op}{\partial S}$ which in general depends on the costs of each children of $P$. According to the operator $op$, the partial derivative may in fact depend only on some children, and this has a strong impact on the total cost.

- $t$ is also inherited: if $S$ is a child of an expression $P = op(\dots, S, \dots)$, then $S$ occurs once inside every occurrence of $P$, plus each time $S$ is used in the partial derivative $\frac{\partial op}{\partial \dots}$ of $P$ with respect to any of its children. Here also, this depends on the actual operator $op$.

The total cost attached to a sub-tree $S$ is $tc$. It is worth replacing each occurrence of $S$ by a precomputed temporary when $tc$ is larger than the cost of assigning $S$ to the temporary and using it $t$ times, i.e. $c+1+t$, assuming that each access to the temporary costs 1. Similarly, it is worth introducing a temporary for $\overline{S}$ when $\overline{t}\overline{c} > \overline{c}+1+\overline{t}$. Therefore we propose the following adjoint sub-expression elimination algorithm:

compute $c$ and $\overline{t}$ bottom-up on the syntax tree
compute $\overline{c}$ and $t$ top-down on the syntax tree
while (some sub-tree $S$ has a positive $tc-t-c-1$ or $\overline{t}\overline{c}-\overline{t}-\overline{c}-1$)
  find the $S$ that maximizes max($tc-t-c-1$,$\overline{t}\overline{c}-\overline{t}-\overline{c}-1$)
  create a temporary variable for $S$ or $\overline{S}$, whichever is better
  update $c$ and $\overline{t}$ bottom-up above $S$
  update $\overline{c}$ and $t$ top-down on the syntax tree

This greedy algorithm is not guaranteed to create a minimal number of temporary variables. On the other hand it is efficient on large expressions and gives good enough results on real codes. Going back to the example assignment, this algorithm produces the following adjoint code

```
tmp1 = (z(j)+2.0)/v(j)
tmp1_b = g(i, j)*(tau-w(j))*res_b/v(j)
tmp2_b = tmp1*g(i, j)*res_b
z_b(j) = z_b(j) + tmp1_b
v_b(j) = v_b(j) - tmp1*tmp1_b
g_b(i, j) = g_b(i, j) + tmp1*(tau-w(j))*res_b
tau_b = tau_b + tmp2_b
w_b(j) = w_b(j) - tmp2_b
res_b = 0.0
```

Notice that common expression `tau-w(j)` was not eliminated because of the cost/benefit tradeoff. In real engineering codes, long expressions spanning on several lines are commonplace. On these codes, we observed speedups up to 20% coming from adjoint sub-expression elimination.

# 3 Reversal of the Data-Flow

Scientific programs frequently overwrite variables, and there's really no way around that. Programs cannot be turned into single-assignment form because they use iterative solvers and the number of iterations is dynamic. This is the heart of the problem for the adjoint algorithm, since it uses intermediate variables in the reverse order. If the adjoint $\overleftarrow{I_k}$ really uses variable `x` $\in X_{k-1}$, and if some statement $I_{k+l}$ downstream overwrites `x`, then the previous value of `x` must be recovered. We call this the Data-Flow reversal problem.

To our knowledge, Data-Flow reversal strategies always apply one of the two approaches (or a combination), sketched on figure 1:

- *Forward recomputation* of the required subset of state $X_{k-1}$, starting from a stored previous state between $X_0$ and $X_{k-1}$. This is done before running $\overleftarrow{I_k}$, and must be repeated similarly before $\overleftarrow{I_{k-1}}$, $\overleftarrow{I_{k-2}}$, and so on.

- *Backward restoration* of the required subset of state $X_k$, progressively, interleaved with $\overleftarrow{I_p}$ back to $\overleftarrow{I_k}$. In a few cases, this restoration can be done at no cost (think of `x = 2.0*x`), but in general it requires storage of intermediate values on a stack, known as the *tape*, during a preliminary forward execution of $I_1$ to $I_{p-1}$ .

In figure 1, we represent the actual computation of the derivatives as arrows pointing to the left, and the computation of the original statements $I_k$ that build the required intermediate variables, as arrows pointing to the right. Vertically, we represent time, as what is represented on one line can be done only when all lines above are done. Dots indicate whenever values must be stored (black dots) in order to be retrieved later (white dots).

Figure 1 shows the respective merits of the two approaches: recomputation uses little memory but a lot of CPU, opposite for restoration. However, it

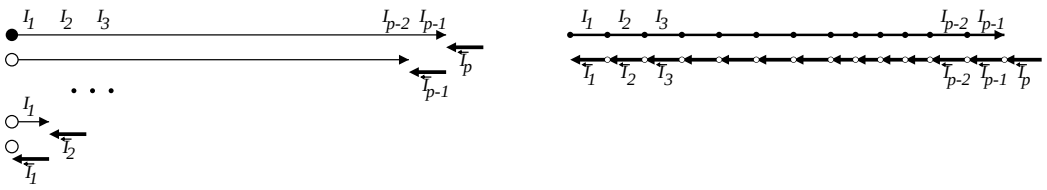


Figure 1: *Structure of the adjoint algorithm using Forward recomputation (left) vs. Backward restoration (right)*

is clear that neither method can run efficiently on a large program. AD tools, whether they are based on forward recomputation (like TAMC [6], TAF) or on backward restoration (like Adifor [2], Tapenade [12], OpenAD [18]), all use a time/memory trade-off known as checkpointing [8, Chapter 12]. Figure 2 illustrates checkpointing of the first half `S` of a program `P`. In Forward recom-

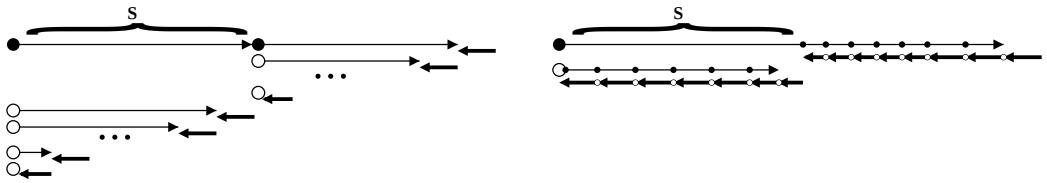


Figure 2: *Checkpointing in the context of Forward recomputation (left) vs. Backward restoration (right)*

putation, a snapshot of the memory is taken so that forward recomputations can restart from it, saving CPU time. In Backward restoration, the first half of `P` is run without storing the intermediate values. It will be run again, with storage, when these values are really needed. The maximum size of the stack is roughly divided by two. Checkpoints can be recursively nested. In this case the multiplicative cost factor of the adjoint program compared to the cost of `P`, both time-wise and memory-wise, grows only like the logarithm of the size of `P`, which is very good. In this case also, the overall shapes of the adjoint program become quite similar, whether one starts from the Forward recomputation extreme or from the Backward restoration extreme. These shapes differ only inside the lower-level checkpoints i.e. program fragments that contain no checkpoint.

Checkpointing is certainly a crucial issue for adjoint algorithms, but it is not strictly speaking a reversal problem. In the sequel we will concentrate on the program fragments that contain no checkpoint, i.e. for which the reversal scheme is one from figure 1.

The first class of improvements to the reversal strategies of figure 1 are basically applications of slicing. In the forward recomputation context, it was called the "ERA" improvement [7] for TAMC. Basically, the goal of a recomputation from $I_1$ to $I_{k-1}$ is to restore state $X_{k-1}$ used by $\overleftarrow{I_{k-1}}$. But in reality $I_k$ uses only some of the available variables, and its adjoint code $\overleftarrow{I_k}$ uses only a subset of these. Therefore recomputation needs to run only a slice, i.e. the statements between $I_1$ and $I_{k-1}$ that are really involved to compute the desired values.

In the backward restoration context, the similar trick is called "adjoint-liveness" [10] and was devised for Tapenade. Basically, we observe that some of the original $I_k$ in the unique forward sweep, although involved in computing the result of `P`, are not involved in computing the gradients. Therefore we can take them out of the adjoint algorithm. In addition, this saves extra storage, because some values are not overwritten any more. The recursive equation that computes the set $\overline{\mathbf{Live}}(I_k; \ldots; I_p)$ of adjoint-live variables just before $I_k$ is straightforward:

$$\overline{\mathbf{Live}}(I_k; D) = \mathbf{Use}(\overleftarrow{I_k}) \cup (\overline{\mathbf{Live}}(D) \otimes \mathbf{Dep}(I_k))$$

which states that a variable is adjoint-live either if it is used in the adjoint code $\overleftarrow{I_k}$ of $I_k$, or if $I_k$ depends on it to compute some result that is adjoint-live for the tail $D$ of the program. This recursive rule stops when the code tail is empty, for which $\overline{\mathbf{Live}}([]) = \emptyset$. When the $\overline{\mathbf{Live}}$ sets are computed and we find a statement whose results are out of the following $\overline{\mathbf{Live}}$ set, the statement can be taken away.

In the same backward restoration context, there is another slicing-based improvement: it was called the "TBR" improvement [4, 11] for Tapenade and OpenAD. As we said in the "ERA" description, the adjoint code for $I_k$ may not use all the variables that $I_k$ uses. Therefore, it often happens that some value of a variable `x` is never needed for the derivatives computation. If this happens, then even if `x` is overwritten we can neglect to restore its

value, saving memory space. The "TBR" analysis finds the set of variables from the original program that are indeed used in the partial derivatives. The recursive equation that computes the set $\mathbf{TBR}(I_1;\ldots;I_k)$ of variables whose value just after $I_k$ is necessary for the adjoint code of $I_1$ till $I_k$ is:

$$\mathbf{TBR}(U;I_k) = (\mathbf{TBR}(U) \setminus \mathbf{Kill}(I_k)) \cup \mathbf{Use}(\overleftarrow{I_k})$$

This equation states that a variable value is necessary immediately after $I_k$ either if it is used in the adjoint code $\overleftarrow{I_k}$ of $I_k$, or if it was already necessary before $I_k$ and it was not overwritten by $I_k$, in which case it is still the same value which is needed. This recursive rule stops when reaching the beginning of `P`, for which $\mathbf{TBR}([]) = \emptyset$. When the **TBR** sets are computed and we reach a statement that overwrites some variables, only the intersection of these variables with the **TBR** set just before the statement need to be stored.

Let us now take a closer look at the backward restoration strategies. Storing and retrieving is not the only option: sometimes, inversion is possible. The simplest example is statement `x = 2.0*x`. If restoration of the upstream `x` is needed, it can very well be done by `x = x/2.0`, provided that the downstream `x` has been restored too. Such statements are not very frequent, however this can be generalized: statement `x = a-b` can also be inverted to restore say, `b`, provided `a` and the downstream `x` have been restored too. In fact the main problem is blending this tactique with memory storage, as there is a combinatorial explosion when looking for the best strategy. This is now becoming a hot topic in the community of AD tool builders.

To gain more insight in the Data-Flow reversal requires that we consider the data-dependence graph. Figure 3 is an magnified view of the right of figure 1, showing a possible data-dependence graph. For clarity, only the necessary anti (read-to-overwrite) dependencies are shown, and output dependencies (write-to-overwrite) are omitted. The true dependencies are shown, and the computation dependencies inside each instruction are shown with thinner arrows.

The first thing to notice is that the dependencies between the gradient values are symmetric of the dependencies between the original variables in the original code. In [9], we proposed a demonstration of this particularly useful symmetry: this isomorphism between the backward sweep and the forward

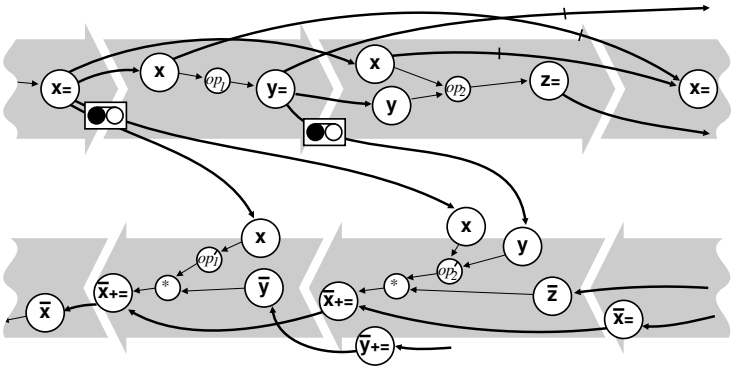


Figure 3: *The data-dependence graph of the adjoint algorithm*

sweep of the adjoint algorithm tells us that many data-dependence-based properties are preserved, such as parallelizability. The three dependencies labeled with a small tape-recorder symbol are very special: these are indeed true dependencies that *cannot* be satisfied by the algorithm as it is, because the variable gets overwritten before the read operation is made. Therefore, to satisfy these dependencies requires storage of the value during the forward sweep and retrieval during the backward sweep, using the tape. In figure 3, two values are stored, `x` and `y`. However, this might not be optimal in terms of memory consumption. Consider the bipartite graph going from the *read* nodes of the forward sweep to the operations nodes of the backward sweep that combine gradient values with partial derivatives depending on the forward sweep (here the two multiplication nodes labeled with `*`). To minimize memory consumption, one must look for a minimal cut of this bipartite graph. In the example, one could store $\mathtt{x}\, op_2' \,\mathtt{y}$. In other words, the question is what to store, the intermediate values or the partial derivatives that use them? The answer depends on the actual code. On the example, there is actually a tie since there are two intermediate values `x` and `y`, as well as two partial derivatives. The OpenAD tool implements heuristics that sometimes decide to store the partial derivatives. On the other hand, Tapenade and Adifor always store the intermediate values.

If the choice is to store the intermediate values, another question is *when* to push them on the stack. To preserve the convenient stack structure of the tape, the popping order must conform with the order of first uses during

the backward sweep. Reversal comes into play here, as the answer is to push `x` on the stack during the forward sweep, between the *last* use of `x` and the next overwrite of `x`. The simplest way to do that is the *save on kill* strategy, which stores `x` just before it is overwritten.

# 4 Reversal of the Control-Flow

The topmost level at which reversal is needed is the Control-Flow. A good introduction to the question can be found e.g. in [15, 17]. Let's start with the representative example of a conditional statement shown by figure 4. If at some time during the original run the Control-Flow goes to one branch of the conditional, then the adjoint algorithm must go to the adjoint of the same branch. Therefore the direction chosen by the conditional must somehow be retrieved later. The danger that appears on figure 4 is that this must be

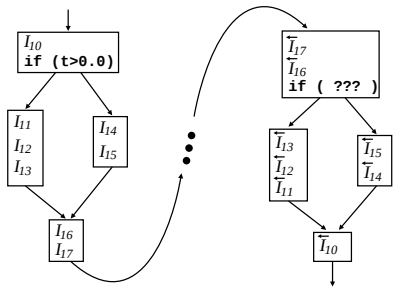


Figure 4: *Control-Flow Reversal of a conditional*

done by the adjoint algorithm just after $\overleftarrow{I_{16}}$, and not just before $\overleftarrow{I_{10}}$, which would be more consistent with the reversal order.

There are basically two ways to retrieve the chosen direction in the adjoint algorithm:

- Either we duplicate the test in the adjoint conditional. This is analogous to the *Forward recomputation* strategy of section 3. But because the adjoint test occurs too early, we can employ this strategy only if the variables used in the test (e.g. `t`) are not modified in the conditional itself.

- Or one stores the direction taken when the conditional actually terminates, so that this direction is available at the beginning of the adjoint conditional. This is analogous to the *Backward restoration* of section 3. This amounts to simply storing a boolean on the stack just before the flow merges reaching $I_{16}$.

Structured loops can be treated in a similar fashion. When the variables that occur in the loop bounds are not modified by the loop body, then the adjoint loop control can be built easily. For example the adjoint loop bounds of a "DO" loop control `DO i=2,n,1` is simply `DO i=n,2,-1`. There are minor technical details e.g. when the loop length is not a multiple of the loop stride. Again, problems occur when the variables in the loop bounds are modified by the loop. The loop bounds must be stored in temporaries at loop beginning, but these temporaries are stored on the tape only at loop exit, so that they are available at the beginning of the adjoint loop.

Generally speaking, the adjoint of a well-structured code is another well-structured code whose control statements are directly derived from the original control. When the variables in a control statement are modified by the controlled structured statement, a temporary may be necessary. This is usually cheap. The amount of tape memory required to reverse the Control-Flow is in general very small compared to the tape required to reverse the data flow. When on the other hand the original program is not well structured, then we must resort to a slightly more brutal strategy, that we call *save control on join*. Specifically, each time Control-Flow arrows join at the entrance of a basic block, we save on the tape the arrow effectively taken. In the adjoint code, the tape is read to find the adjoint arrow that must be taken. This is indeed the Data-Flow *save on kill* strategy applied to Control-Flow. This may increase the total size of the tape significantly, especially on systems where a boolean is actually encoded as an integer.

There are two other classes of reversal questions that we view as extensions of the Control-Flow reversal problem. AD tools are beginning to consider these questions only now, and we don't know of a tool that treats them fully. The first class is the use of pointers, that effectively control the addresses that are referred to by an expression. Figure 5 shows just one representative example, using the C syntax. Pointer `p` is used to select which

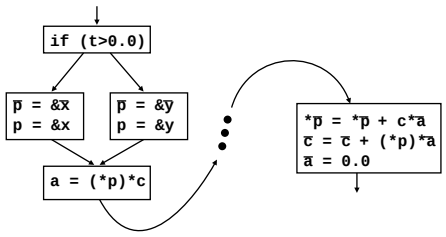


Figure 5: *Control-Flow reversal with pointer usage*

variable is read by statement `a=2*(*p)`. To select which gradient variable will be incremented by its adjoint code, we may define a *gradient pointer* $\overline{\texttt{p}}$. This comes is somewhat surprising as derivatives are usually defined on real numbers only, but this is actually an address memorization strategy. The moment when $\overline{\texttt{p}}$ can be set is exactly when `p` itself is set, and if ever `p` is overwritten during the sequel of the original program, then `p` and $\overline{\texttt{p}}$ must be stored on the tape.

Storing a pointer on the tape works fine as long as the memory it points to is not freed. This is granted for static memory, but we are lacking a convenient solution for dynamic memory. Suppose the forward sweep of the adjoint program, following the original program, allocates some memory, makes pointer `p` point into this memory, and then frees this memory. During the backward sweep, some memory will be allocated again when needed, but we cannot guarantee that it is the same memory chunk. The tape mechanism ensures that `p` recovers its value, but it is an address in the previous memory chunk, not in the new one ! There may be a convenient solution if we can request for a memory allocation at a specified memory location: we are confident we can prove that the memory freed on the forward sweep is still free when the backward sweep reaches the same location. It would then suffice to allocate exactly the same memory chunk on the way back.

The second class of control reversal questions is related to message-passing parallelism, which is routinely used in the scientific programs we are targeting at. As long as communications are one-to-one, Control-Flow reversal is easy: when process `P1` sends a variable `x` to process `P2`, the adjoint algorithm must send the gradient variable $\overline{\texttt{x}}$ from `P2` back to `P1`. Communications one-to-all

and all-to-one require in addition that the sender of a message be identified in the message itself, so that the adjoint algorithm can send the gradient back. Synchronization may also require care, but so far we saw no application for which synchronization has an effect on differentiable values.

# 5 Automatic Differentiation at INRIA

Automatic Differentiation has been around for many years: a russian paper from L.M. Beda mentions the name and concept in 1959, and libraries that execute operations together with propagation of analytical derivatives date from the 70's (WCOMP, UCOMP). Adjoint Algorithms primarily existed as hand-written programs that solve the so-called Adjoint State Equations from the theory of control. Only in the 80's did some Automatic Differentiation tools appear that could generate good enough Adjoint Algorithms (JAKE, JAKEF), and true awareness of these tools in industrial Scientific Computing dates from the 90's.

At INRIA, research on AD was initiated by Jacques Morgenstern and André Galligo in the research team SAFIR. One of the outcomes of this work was the AD tool O$\partial$yssée [5], which very soon featured a successful adjoint mode. At that time already, this work was warmly supported by Gilles Kahn, who saw the interest of a scientific collaboration between the algorithmic developments of SAFIR and the interactive programming environment Centaur developed by the CROAP team.

In 1999, Valérie Pascual and myself proposed the creation of a new research team TROPICS, to further develop AD for Adjoint Algorithms. Gilles was again our strongest support. He liked to say facetiously that the difficulties we sometimes experienced in front of skilled hand-programmers of Adjoint Algorithms, sounded like the first compiler programs trying to compete with hand-writing of assembly code.

# 6 Current status and Perspectives

Adjoint Algorithms produced by AD have made impressive progress in the last few years. Large industrial codes that were simply out of reach are now

routinely "adjointed" by AD tools. This is partly due to the development of the reversal strategies described in this paper, as well as to the progress made a few years before in nested checkpointing strategies. AD now produces Adjoint algorithms whose quality competes with that of hand-written adjoints, which took months or years of development and debugging. Since the adjoint code must evolve together with its original code, the ability to rebuild it in a matter of minutes is much appreciated.

Problems remain, though, and we discussed some of them. These problems essentially connect to the weaknesses of our static analyses. Undecidability of static analysis of course sets a limit to the power of AD tools. But very far before that, AD tools are already limited by their simplifying assumptions: no AD tool uses the very clever (and expensive) algorithms from parallellization, such as array region analysis or reaching definition analysis.

Maybe the path to overcome these limitations is to develop interaction with the end-user, e.g. through directives. An end-user always knows more about the code than any static analysis will find out. Most AD tools are now going in this direction, and TAF is probably ahead in this respect. For the general application domain of Adjoint Algorithms, i.e. Scientific Computing, knowledge from the user is even more invaluable than elsewhere. Large parts of the programs correspond to well-known mathematical operations whose properties have been studied for centuries. For example, no static analysis will be able to detect that a code fragment actually solves a symmetric linear system using the latest champion iterative algorithm. Yet if the user provides a clever AD tool with this knowledge, the tool should be able to find that the adjoint is indeed the original solver itself.

Reversal strategies are also explored in other domains. Some debuggers try to keep track of execution before a program crashed, allowing the user to go back in time to discover the true origin of the problem. This is a challenge, even more so when debugging a distributed parallel execution. We hope this paper could sparkle cross discussions, resulting in improved reversal strategies for adjoint algorithms. I'm sure Gilles would like that.

# References


[1] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*. LNCSE. Springer, 2006. Selected papers from AD2004, Chicago, July 2004.

[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*. LNCSE. Springer, 2002. Selected papers from AD2000, Nice, June 2000.

[4] C. Faure and U. Naumann. Minimizing the tape size. In [3], chapter VIII, pages 293–298. 2002.

[5] C. Faure and Y. Papegay. Odyssée User's Guide. Version 1.7. Technical report 224, INRIA, 1998.

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

[7] R. Giering and T. Kaminski. Generating recomputations in reverse mode AD. In [3], chapter VIII, pages 283–291. 2002.

[8] A. Griewank. *Evaluating derivatives: principles and techniques of Algorithmic Differentiation*. SIAM, Frontiers in Applied Mathematics, 2000.

[9] L. Hascoët. The data-dependence graph of adjoint programs. Research Report 4167, INRIA, 2001.

[10] L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In [1], pages 135–146. 2006.

[11] L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode Automatic Differentiation. *Future Generation Computer System*, 21(8):1401–1417, 2005.

[12] L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.

[13] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming, Ser. A*, 99(3):399–421, 2004.

[14] U. Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Prog.*, 2006. In press. Appeared Online First.

[15] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64. IEEE Computer Society, 2004.

[16] J. Nocedal and S.-J. Wright. *Numerical Optimization*. Springer, Series in Operations Research, 1999.

[17] J. Utke, A. Lyons, and U. Naumann. Efficient reversal of the intraprocedural flow of control in adjoint computations. *J. Syst. Softw.*, 79(9):1280–1294, 2006.

[18] J. Utke, U. Naumann, M. Fagan, N. Tallent, Strout M., P. Heimbach, C. Hill, and C. Wunsch. OpenAD/F: A modular, open-source tool for Automatic Differentiation of Fortran codes. Technical report ANL/MCS-P1230-0205, Argonne National Laboratory, 2006. Submitted to ACM TOMS.