Tangent-on-Tangent vs. Tangent-on-Reverse for Second Differentiation of Constrained Functionals

Massimiliano Martinelli and Laurent Hascoët

INRIA, TROPICS team, 2004 route des Lucioles 06902 Sophia-Antipolis France
{Massimiliano.Martinelli,Laurent.Hascoet}@sophia.inria.fr

Summary. We compare the Tangent-on-Tangent and the Tangent-on-Reverse strategies to build programs that compute second derivatives (a Hessian matrix) using Automatic Differentiation. In the specific case of a constrained functional, we find that Tangent-on-Reverse outperforms Tangent-on-Tangent only above a relatively high number of input parameters. We describe the algorithms to help the end-user apply the two strategies to a given application source. We discuss the modification needed inside the AD tool to improve Tangent-on-Reverse AD.

Key words: Automatic Differentiation, Gradient, Hessian, Tangent-on-Tangent, Tangent-on-Reverse, Software Tools, TAPENADE

1 Introduction

As computational power increases, Computational Fluid Dynamics evolves towards more complex simulation codes and more powerful optimization capabilties. However these high fidelity models cannot be used only for deterministic design, assuming perfect knowledge of all environmental and operational parameters. Many reasons, including social expectations, demand accuracy and safety control and even high fidelity models remain subject to errors and uncertainty. Numerical error for instance, need be controlled and partly corrected with linearized models. Uncertainty arises everywhere, e.g. in the mathematical model, in manufacturing tolerances, and in operational conditions that depend on atmospheric conditions. Techniques for propagating these uncertainties are now well established [16; 13; 5]. They require extra computational effort, but really improve the robustness of the design [9], and can help the designer sort out the crucial sources of uncertainty from the negligible.

We consider uncertainty propagation for a *cost functional*

$$j\colon \gamma \mapsto j(\gamma) = J(\gamma, W) \in \mathbb{R} \tag{1}$$

with uncertainty affecting the *control variables* $\gamma \in \mathbb{R}^n$, and where the *state variables* $W = W(\gamma) \in \mathbb{R}^N$ satisfy a (nonlinear) *state equation*

$$\Psi(\gamma, W) = 0. \tag{2}$$

Equation (2) expresses the discretisation of the PDE governing the mathematical model of the physical system e.g. the stationary part of the Euler or Navier-Stokes equations. We view (2) as an *equality constraint* for the functional (1).

The two main type of *probabilistic* approaches for propagating uncertainties are the Monte Carlo methods [10; 4] and the perturbative methods based on the Taylor expansion (Method of Moments [13] and Inexpensive Monte-Carlo [5]). The straightforward full nonlinear Monte-Carlo technique can be considered the most robust, general and accurate method, but it proves prohibitively slow since it converges only with the square root of the number of nonlinear simulations. In contrast, the Method of Moments gives approximate values of the mean and variance at the cost of only one nonlinear simulation plus a computation of the gradient and Hessian of the constrained functional. This requires far less runtime than the full nonlinear Monte-Carlo, but at the (high) cost of developing the code that computes the gradient j' and Hessian j'' of the constrained functional.

Hessians also occur in the context of *robust design* [1], in which the optimization cost functionals involve extra robustness terms such as $j_R(\gamma) = j(\gamma) + \frac{1}{2}\sum j''_{ij} C_{ij}$, where the C_{ij} are elements of the covariance matrix of uncertain variables.

Writing the code for the gradient and Hessian by hand is tedious and error-prone. A promising alternative is to build this code by Automatic Differentiation (AD) of the program that computes the constrained functional. This program has a general structure sketched in

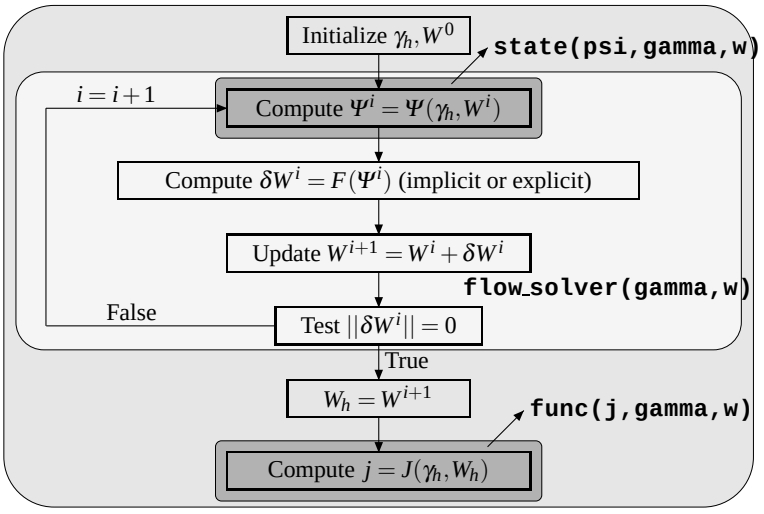

Fig. 1. Program structure for evaluating a constrained functional with a fixed-point algorithm

Fig. 1: a flow solver computes iteratively the state W_h satisfying (2) from a given γ_h, then computes j. This program can be seen as a *driver* that calls application-specific routines `state(psi,gamma,w)` for the state residual Ψ and `func(j,gamma,w)` for the functional J. Let us contrast two ways of building the program that computes j' and j'' using AD:

- *Brute-force differentiation:* Differentiate directly the function j as a function of γ. This means that the entire program of Fig. 1 is differentiated as a whole. This takes little advantage of the fixed-point structure of the algorithm. Performance is often poor as the differentiated part contains the iterative state equation solver `flow_solver(gamma,w)`. To be reliable, this strategy requires a careful control of the number of iterations, which is out of the scope of AD tools [6; 2]. A small change of the input may radically change the control flow of the program, e.g. the iteration number in `flow_solver`. The computed function is thus only piecewise-continuous, leaving the framework where AD is fully justified. Regarding performances, since j is scalar, *reverse mode* AD [7, Sec. 3-4] is recommended over *tangent mode* to compute j', and for the same reason Tangent-on-Reverse is recommended over Tangent-on-Tangent for j''.
- *Differentiation of explicit parts:* Do not differentiate the driver part, but only the routines for Ψ and J, then plug the resulting routines into a new, specialized driver to compute the gradient and Hessian. This sophisticated way proves more efficient and we will focus on it in the sequel. Notice that the derivatives for Ψ and J need to be computed only at the final state W_h, which results in a cheaper reverse mode because fewer intermediate values must be restored in reverse order during the computation of the derivatives.

For the gradient, several works advocate and illustrate this second way [3]. For the Hessian, the pioneering works of Taylor et al. [14; 15] define the mathematical basis and examine several approaches, of which two apply to our context of a constrained functional with a scalar output j. Following Ghate and Giles [5], we also call these approaches Tangent-on-Reverse (ToR) and Tangent-on-Tangent (ToT). The general complexity analysis provided in [14] finds linear costs with respect to the size n of γ. This leads to the conclusion that ToT is unconditionally better than ToR. This is slightly counter-intuitive compared to the general case of brute-force differentiation. However in our context where matrices can be too large to be stored, every linear system must be solved using a matrix-free method (e.g. GMRES) with an ILU(1) preconditioner. This paper revisits the Hessian evaluation problem in this context. The full mathematical development can be found in [11; 12].

Section 2 studies the ToR approach while Sec. 3 studies the ToT approach, both sections going from the mathematical equations to the algorithm and to a refined complexity analysis. Section 4 compares the two approaches and gives first experimental measurements. We claim that ToT is no longer linear with respect to n, and for large enough, yet realistic n, ToR does outperform ToT. Section 5 discusses the ToR approach from the point of view of the AD tool.

2 Tangent-on-Reverse Approach

Following [14], the projection of the Hessian along a direction $\delta \in \mathbb{R}^n$ is given by

$$\left(\frac{d^2 j}{d\gamma^2}\right)\delta = \frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^T\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^T\theta - \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi\right]\delta - \frac{\partial}{\partial W}\left[\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi\right]\theta q - \left(\frac{\partial\Psi}{\partial\gamma}\right)^T\lambda$$

where vectors Π, θ, and λ are the solutions of

$$\begin{cases} \left(\dfrac{\partial\Psi}{\partial W}\right)^T\Pi = \left(\dfrac{\partial J}{\partial W}\right)^T \\ \left(\dfrac{\partial\Psi}{\partial W}\right)\theta = -\left(\dfrac{\partial\Psi}{\partial\gamma}\right)\delta \\ \left(\dfrac{\partial\Psi}{\partial W}\right)^T\lambda = \dfrac{\partial}{\partial\gamma}\left(\dfrac{\partial J}{\partial W}\right)^T\delta + \dfrac{\partial}{\partial W}\left(\dfrac{\partial J}{\partial W}\right)^T\theta - \dfrac{\partial}{\partial\gamma}\left[\left(\dfrac{\partial\Psi}{\partial W}\right)^T\Pi\right]\delta - \dfrac{\partial}{\partial W}\left[\left(\dfrac{\partial\Psi}{\partial W}\right)^T\Pi\right]\theta \end{cases}$$

From these equations, we derive the algorithm sketched by Fig. 2. It evaluates the Hessian column by column, repeatedly computing $\frac{d^2 j}{d\gamma^2} e_i$ for each component e_i of the canonical basis of $\mathbb{R}^n$. Notice that the computation of Π is independent from the particular direction δ and is therefore done only once. In contrast, new vectors θ, λ are computed for each e_i. The

Solve for Π **in** $\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi = \left(\frac{\partial J}{\partial W}\right)^T$

For each $i \in 1..n$

- **Solve for** θ **in** $\left(\frac{\partial \Psi}{\partial W}\right)\theta = -\left(\frac{\partial \Psi}{\partial \gamma}\right) e_i$
- **Compute** $\overleftarrow{\gamma}_J = \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T e_i + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^T \theta$
- **Compute** $\overleftarrow{W}_J = \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^T e_i + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T \theta$
- **Compute** $\overleftarrow{\gamma}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi\right] e_i + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi\right] \theta$
- **Compute** $\overleftarrow{W}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi\right] e_i + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi\right] \theta$
- **Solve for** λ **in** $\left(\frac{\partial \Psi}{\partial W}\right)^T \lambda = \overleftarrow{W}_J - \overleftarrow{W}_\Psi$
- **Compute** $\left(\frac{d^2 j}{d\gamma^2}\right) e_i = \overleftarrow{\gamma}_J - \overleftarrow{\gamma}_\Psi - \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \lambda$

End For

Fig. 2. Algorithm to compute the Hessian with the ToR approach

vectors Π, θ, and λ are solutions of linear systems, and can be computed using an iterative linear solver. In our experiments, we use GMRES with an ILU(1) preconditioner built from an available approximate Jacobian. During this process, the left-hand side of the equations for Π, θ, and λ is evaluated repeatedly for different vectors. The routine that performs this evaluation is obtained by differentiation of the routine `state` that computes Ψ, in tangent mode for θ, in reverse mode for Π and λ. The rest of the algorithm needs $\frac{\partial J}{\partial W}^T$, $\frac{\partial \Psi}{\partial \gamma} e_i$, $\frac{\partial \Psi}{\partial \gamma}^T \lambda$, which are obtained through a single tangent or reverse differentiation. It also needs the complex expressions that we name $\overleftarrow{W}_J$, $\overleftarrow{\gamma}_J$, $\overleftarrow{W}_\Psi$, and $\overleftarrow{\gamma}_\Psi$, which are obtained through ToR differentiation of the routines `state` (evaluating $\Psi(\gamma, W)$) and `func` (evaluating $J(\gamma, W)$). For instance the ToR differentiation of `state` with respect to input variables `gamma` and `w` has the following inputs and outputs:

$$\texttt{state\_bd}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psib}}, \underset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammad}}, \underset{\downarrow}{\texttt{gammab}}, \underset{\downarrow}{\texttt{gammabd}}, \underset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wd}}, \underset{\downarrow}{\texttt{wb}}, \underset{\downarrow}{\texttt{wbd}})$$
$$\begin{matrix} \Pi & \gamma & e_i & & & W & \theta & & \\ \downarrow & \downarrow & \downarrow & & & \downarrow & \downarrow & & \\ \Psi & & & \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi & \overleftarrow{\gamma}_\Psi & & & \left(\frac{\partial \Psi}{\partial W}\right)^T \Pi & \overleftarrow{W}_\Psi \end{matrix}$$

Similar differentiation of `func` gives us $\overleftarrow{W}_J$ and $\overleftarrow{\gamma}_J$.

After implementing this algorithm, we observe that all iterative solutions take roughly the same number of steps n_{iter}. Moreover, the runtime to compute Ψ largely dominates the runtime to compute J, and the same holds for their derivatives. A differentiated code is generally slower than its original code by a factor we call α, which varies with the original code. We call α_T (resp. α_R) the slowdown factor of the tangent (resp. reverse) code of Ψ. We call α_{TR} the slowdown factor of the second differentiation step that computes the ToR derivative of Ψ. Normalizing with respect to the runtime to compute Ψ, we find the cost for the full Hessian:

$$n_{\text{iter}}\alpha_R + n\left(n_{\text{iter}}\alpha_T + \alpha_{TR}\alpha_R + n_{\text{iter}}\alpha_R\right)$$

3 Tangent-on-Tangent Approach

In contrast, the ToT approach computes each element of the Hessian separately. Following [14] and introducing the differential operator $D^2_{i,k}$ for functions $F(\gamma, W)$ as:

$$D^2_{i,k}F = \frac{\partial}{\partial\gamma}\left(\frac{\partial F}{\partial\gamma}e_i\right)e_k + \frac{\partial}{\partial W}\left(\frac{\partial F}{\partial\gamma}e_i\right)\frac{dW}{d\gamma_k} + \frac{\partial}{\partial W}\left(\frac{\partial F}{\partial\gamma}e_k\right)\frac{dW}{d\gamma_i} + \frac{\partial}{\partial W}\left(\frac{\partial F}{\partial W}\frac{dW}{d\gamma_i}\right)\frac{dW}{d\gamma_k}\ .$$

the elements of the Hessian are

$$\frac{d^2 j}{d\gamma_i d\gamma_k} = D^2_{i,k}J + \frac{\partial J}{\partial W}\frac{d^2W}{d\gamma_i d\gamma_k} = D^2_{i,k}J - \Pi^T\left(D^2_{i,k}\Psi\right)$$

where Π is the adjoint state, i.e. the solution of the linear system $\left(\frac{\partial\Psi}{\partial W}\right)^T\Pi = \left(\frac{\partial J}{\partial W}\right)^T$. These equations give us the algorithm sketched by Fig. 3 to evaluate the Hessian element by element. Efficiency comes from the key observation that the total derivatives $\frac{dW}{d\gamma_i}$ occur in many places and should be precomputed and stored. They are actually the θ of the ToR approach, for each vector e_i of the canonical basis of $\mathbb{R}^n$. Terms $D^2_{i,k}\Psi$ (resp. $D^2_{i,k}J$) are obtained through ToT

Solve for Π **in** $\left(\dfrac{\partial\Psi}{\partial W}\right)^T\Pi = \left(\dfrac{\partial J}{\partial W}\right)^T$
For each $i \in 1..n$
  Solve for θ_i **in** $\left(\dfrac{\partial\Psi}{\partial W}\right)\theta_i = -\left(\dfrac{\partial\Psi}{\partial\gamma}\right)e_i$ **and store it**
End For
For each $i \in 1..n$
  For each $k \in 1..i$
    Compute $\dfrac{d^2 j}{d\gamma_i d\gamma_k} = D^2_{i,k}J - \Pi^T\left(D^2_{i,k}\Psi\right)$
  End For
End For

Fig. 3. Algorithm to compute the Hessian with the ToT approach

differentiation of routine `state` (resp. `func`). For instance the ToT differentiation of `state` with respect to input variables `gamma` and `w` has the following inputs and outputs:

$$\underset{\downarrow\ \Psi}{\texttt{psi}},\ \underset{\downarrow\ \dot{\Psi}}{\texttt{psid}},\ \underset{\downarrow\ D^2_{i,k}\Psi}{\texttt{psidd}},\ \underset{\downarrow}{\texttt{gamma}}^{\gamma},\ \underset{\downarrow}{\texttt{gammad0}}^{e_k},\ \underset{\downarrow}{\texttt{gammad}}^{e_i},\ \underset{\downarrow}{\texttt{w}}^{W},\ \underset{\downarrow}{\texttt{wd0}}^{\theta_k},\ \underset{\downarrow}{\texttt{wd}}^{\theta_i}\Big). \tag{3}$$

Similar differentiation of `func` gives us $D^2_{i,k}J$.

After implementing this algorithm, we observe that the expensive parts are solving for the θ_i and computing the $D^2_{i,k}\Psi$. With the same conventions as in Sec. 2, and introducing α_{TT} as the slowdown factor for the second tangent differentiation step, we obtain the cost for the full Hessian:

$$n_{\text{iter}}\alpha_R + n\, n_{\text{iter}}\alpha_T + \frac{n(n+1)}{2}\alpha_{TT}\alpha_T$$

Observe that this cost has a quadratic term in n.

4 Comparing ToR and ToT Approaches

The slowdown factors α resulting from AD depend on the differentiation mode (tangent or reverse), on the technology of the AD tool, and on the original program. For instance, on the 11 big codes that we use as validation tests for the AD tool TAPENADE[8], we observe that α_T ranges from 1.02 to 2.5 (rough average 1.8), and α_R ranges from 2.02 to 9.4 (rough average 5.0). Notice that TAPENADE offers the possibility to perform the second level of differentiation (e.g "T" in "ToR") in multi-directional (also called "vector") mode. This can slightly reduce the slowdown α_{TT} by sharing the original function evaluation between many differentiation directions. For technical reasons, we didn't use this possibility yet. However this doesn't affect the quadratic nature of the complexity of the ToT approach.

On the 3D Euler CFD code that we are using for this first experiment, we measured $\alpha_T = 2.4$, $\alpha_R = 5.9$, $\alpha_{TT} = 4.9$ and $\alpha_{TR} = 3.0$, with very small variations between different runs. The higher α_{TT} may come from cache miss problems, as first and second derivative arrays try to reside in cache together with the original arrays.

We also observe that the number of iterations n_{iter} of the GMRES solver remain remarkably constant between 199 and 201 for the tangent linear systems, and between 206 and 212 for the adjoint linear systems.

With these figures and our cost analysis, we find that ToR will outperform ToT for the Hessian when the dimension n is over a break-even value of about 210.

It turns out that this CFD code does not easily lend itself to increasing the value of n above a dozen. So we have devised a second test case, with a simplified (*artificial*) nonlinear state function Ψ in which the dimension of the state W was 100000 and the number of control variables moves from 1 to 280. To be more precise we used the functional $J(\gamma,W) = \sum_{i=1}^{N}\sqrt{|W_i|}$ and the state residual

$$\Psi_i(\gamma,W) = \frac{1}{W_i^2} - \frac{1}{\left[1-f(\gamma)\right]^2\alpha_i^2} + \frac{1}{W_i^2\prod_{j=1}^{n}\gamma_j} - \frac{1}{\alpha_i^2}$$

with $\alpha_i > 0$, $f(\gamma) = \sum_{i=1}^{n-1}\left[(1-\gamma_i)^2 + 10^{-6}(\gamma_{i+1}-\gamma_i^2)^2\right]$. With the above definitions and using $\gamma = (1,1,\ldots,1)$ the state equation $\Psi = 0$ is satisfied with $W_i = \pm\alpha_i$. For all runs, we observe that the solutions of the linear systems requires 64 GMRES iterations (without preconditioning) for the tangent and 66 iterations for the adjoint version, with very little variability with respect to the rhs. Figure 4 shows the CPU times for the ToT and ToR approaches, when the number n of control variables varies from 1 to 280. We observe that ToR is cheaper than ToT

when $n \gtrsim 65$. We also observe the linear behaviour of the ToR cost and the quadratic cost for ToT, as expected.

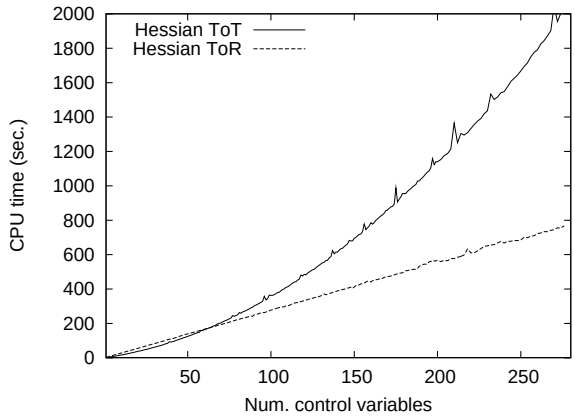

Fig. 4. CPU times cost to compute the full Hessian via the ToT and ToR approaches, with respect to the number of control variables. We assume the adjoint state Π is available and its cost (equal for ToT and ToR and independent from n) is not shown in the figure.

It can be observed that all the linear systems always use the same two matrices $\frac{\partial \Psi}{\partial W}$ and $\frac{\partial \Psi}{\partial W}^T$. This can be used to devise solution strategies even more efficient than the GMRES+ILU that we have been using here. That could further decrease the relative cost of the solving steps, and therefore strengthen the advantage of the ToR approach.

We have been considering the costs for computing the full Hessian, which is not always necessary. Actual choice of the approach also depends on which part of the Hessian is effectively required. The algorithms and costs that we provide can be easily adapted to obtain single Hessian elements, Hessian diagonals, or Hessian$\times$vector products. Specifically for the Hessian$\times$vector case, the cost through the ToR approach becomes independent from the number n of control variables, namely

$$n_{\text{iter}}\alpha_R + n_{\text{iter}}\alpha_T + \alpha_{TR}\alpha_R + n_{\text{iter}}\alpha_R$$

whereas the ToT approach still requires computing all the θ_i for $i \in 1..n$, for a total cost of

$$n_{\text{iter}}\alpha_R + n\, n_{\text{iter}}\alpha_T + n\alpha_{TT}\alpha_T \ .$$

In this case, ToR outperforms ToT starting from much smaller values of n.

5 The Art of ToR

This work gives us the opportunity to study the difficulties that arise when building the ToR code with an AD tool. These questions are not limited to the context of constrained functionals.

Assuming that a source transformation tool is able to handle the complete source language, its repeated application should not be problematic. AD tools such as TAPENADE belong to this category. Indeed, ToT approach works fine, although some additional efficiency could be gained by detecting common derivative sub-expressions. However, problems arise with ToR, due to the presence of external calls for the stack management in the reverse code. These external calls result from the architectural choice of the reverse mode of TAPENADE, which uses a storage stack. The situation might be different with reverse codes that rely on recomputation instead of storage.

With the stack approach, stack primitives are external routines because many of our users still use Fortran77, which has no standard memory allocation mechanism. Though reverse AD of programs with external calls is possible, it must be done with care. It relies on the user to provide the type and data flow information of the external routines, together with their hand-written differentiated versions. This user-given data is crucial and mistakes may cause subtle errors. Maybe the safest rule of thumb is to write an alternative Fortran implementation of the external primitives as a temporary replacement, then differentiate the code, and then replace back with the external primitives. For instance in the present case, we can easily write a replacement `PUSH` and `POP` using a large enough storage array. After reverse AD, we observe two things:

- First, as shown in Fig. 5, the storage array remains passive until some active variable is `PUSH`ed. It then remains active forever, even when reaching the `POP` of some variable that was passive when it was `PUSH`ed. As a consequence, the matching `POP` of a passive `PUSH` may be active.
- Second, the storage array has given birth to a separate, differentiated storage array devoted to derivatives.

This guides us on the data flow information to be provided about `PUSH` and `POP`, and most importantly on the correct implementation of their tangent derivatives. Specifically `PUSH_D(x, xd)` must push `x` onto the original stack and `xd` onto the differentiated stack, and `POP_D(x, xd)` must set `xd` to `0.0` when the differentiated stack happens to be empty. Incidentally, a different implementation using a single stack would produce a run-time error.

Although correct, the ToR code shown in Fig. 5 is not fully satisfactory. The last call to `POP_D` should rather be a plain `POP`. Also, once the stack becomes active, all `PUSH`'es become `PUSH_D`, even when the variable is passive, in which case a `0.0` is put on the differentiated stack. We would prefer a code in which matching `PUSH`/`POP` have their own activity status, and do not get differentiated when the `PUSH`ed variable is passive. In general, matching `PUSH`/`POP` cannot be found by static analysis of the reverse code. It requires an annotated source.

Although matching `PUSH`/`POP`'s cannot be found from the reverse code, this information was available when the reverse code was built. Thus, TAPENADE now provides support to the ToR differentiation by placing annotations in the reverse code, and by using those during tangent differentiation to find all calls to stack operations that need not be differentiated.

6 Conclusion

We have studied two approaches to efficiently compute the second derivatives of constrained functionals. These approaches appear particularly adapted in the case where the constraint contains a complex mathematical model such as PDE's, which is generally solved iteratively.

Original: $F : a, b, c \mapsto r$	**Reverse:** $\overline{F} : a, b, c, \overline{r} \mapsto \overline{a}, \overline{b}, \overline{c}$	**Tangent-on-Reverse:** $\dot{\overline{F}} : a, \dot{a}, b, \dot{b}, c, \dot{c}, \overline{r}, \dot{\overline{r}} \mapsto \overline{a}, \dot{\overline{a}}, \overline{b}, \dot{\overline{b}}, \overline{c}, \dot{\overline{c}}$
x = 2.0 r = x*a x += c r += x*b x = 3.0 r += x*c	x = 2.0 r = x*a PUSH(x) x += c r += x*b PUSH(x) x = 3.0 r += x*c $\overline{x}$= c*$\overline{r}$ $\overline{c}$+= x*$\overline{r}$ POP(x) $\overline{x}$= 0.0 $\overline{x}$= b*$\overline{r}$ $\overline{b}$+= x*$\overline{r}$ POP(x) $\overline{c}$+= $\overline{x}$ $\overline{x}$+= a*$\overline{r}$ $\overline{a}$+= x*$\overline{r}$ $\overline{x}$= 0.0	$\dot{x}$= 0.0 x = 2.0 PUSH(x) $\dot{x}$= $\dot{c}$ x += c PUSH_D(x,$\dot{x}$) $\dot{x}$= 0.0 x = 3.0 $\dot{\overline{c}}$+= x*$\dot{\overline{r}}$ $\overline{c}$+= x*$\overline{r}$ POP_D(x,$\dot{x}$) $\dot{\overline{x}}$= $\dot{b}$*$\overline{r}$+b*$\dot{\overline{r}}$ $\overline{x}$= b*$\overline{r}$ $\dot{\overline{b}}$+= $\dot{x}$*$\overline{r}$+x*$\dot{\overline{r}}$ $\overline{b}$+= x*$\overline{r}$ POP_D(x,$\dot{x}$) $\dot{\overline{c}}$+= $\dot{\overline{x}}$ $\overline{c}$+= $\overline{x}$ $\dot{\overline{a}}$+= $\dot{x}$*$\overline{r}$+x*$\dot{\overline{r}}$ $\overline{a}$+= x*$\overline{r}$

Fig. 5. ToR differentiation on a small code. *Left*: Original code, *middle*: Reverse code, *right*: ToR code. Reverse-differentiated variables ($\overline{x}$) are shown with a bar above, tangent-differentiated variables ($\dot{x}$, $\dot{\overline{x}}$) with a dot above. Code in light gray is actually dead and stripped away by TAPENADE, with no influence on the present study.

Both approaches rely on building differentiated versions of selected subroutines of the original program by means of Automatic Differentiation.

Our main result is that comparing complexity of the Tangent-on-Reverse approach versus Tangent-on-Tangent is not so clear-cut, and it depends on the size *n* of the problem and on the derivatives effectively needed. Also, we propose an automated implementation of both approaches, based on shell scripts and using the AD tool TAPENADE, which had to be modified for better results in the Tangent-on-Reverse mode.

In addition to applying these approaches to even larger CFD codes, one shorter term further research is to study the Reverse-on-Tangent alternative to Tangent-on-Reverse. This option might prove easier for the AD tool, but further experiments are required to compare performances.

References

[1] Beyer, H.G., Sendhoff, B.: Robust optimization – A comprehensive survey. Comput. Methods Appl. Mech. Engrg. **196**, 3190–3218 (2007)

[2] Christianson, B.: Reverse accumulation and attractive fixed points. Optimization Methods and Software **3**, 311–326 (1994)
[3] Courty, F., Dervieux, A., Koobus, B., Hascoët, L.: Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. Optimization Methods and Software **18**(5), 615–627 (2003)
[4] Garzon, V.E.: Probabilistic aerothermal design of compressor airfoils. Ph.D. thesis, MIT (2003)
[5] Ghate, D., Giles, M.B.: Inexpensive Monte Carlo uncertainty analysis, pp. 203–210. Recent Trends in Aerospace Design and Optimization. Tata McGraw-Hill, New Delhi (2006)
[6] Gilbert, J.: Automatic differentiation and iterative processes. Optimization Methods and Software **1**, 13–21 (1992)
[7] Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM (2000)
[8] Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Tech. Rep. 0300, INRIA (2004)
[9] Huyse, L.: Free-form airfoil shape optimization under uncertainty using maximum expected value and second-order second-moment strategies. Tech. Rep. 2001-211020, NASA (2001). ICASE Report No. 2001-18
[10] Liu, J.S.: Monte Carlo Strategies in Scientific Computing. Springer-Verlag (2001)
[11] Martinelli, M.: Sensitivity Evaluation in Aerodynamic Optimal Design. Ph.D. thesis, Scuola Normale Superiore (Pisa) - Université de Nice-Sophia Antipolis (2007)
[12] Martinelli, M., Dervieux, A., Hascoët, L.: Strategies for computing second-order derivatives in CFD design problems. In: Proceedings of WEHSFF2007 (2007)
[13] Putko, M.M., Newman, P.A., Taylor III, A.C., Green, L.L.: Approach for uncertainty propagation and robust design in CFD using sensitivity derivatives. Tech. Rep. 2528, AIAA (2001)
[14] Sherman, L.L., Taylor III, A.C., Green, L.L., Newman, P.A.: First and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. Journal of Computational Physics **129**, 307–331 (1996)
[15] Taylor III, A.C., Green, L.L., Newman, P.A., Putko, M.M.: Some advanced concepts in discrete aerodynamic sensitivity analysis. AIAA Journal **41**(7), 1224–1229 (2003)
[16] Walters, R.W., Huyse, L.: Uncertainty analysis for fluid mechanics with applications. Tech. Rep. 2002-211449, NASA (2002). ICASE Report No. 2002-1