West-East High Speed Flow Field Conference
19-22, November 2007
Moscow, Russia

STRATEGIES FOR COMPUTING SECOND-ORDER DERIVATIVES IN CFD DESIGN PROBLEMS

Massimiliano Martinelli, Alain Dervieux, Laurent Hascoët

INRIA, 2004 Route des Lucioles, 06902 Sophia-Antipolis, France.
{Massimiliano.Martinelli, Alain.Dervieux, Laurent.Hascoet}@sophia.inria.fr

Key words: Automatic Differentiation, second-order derivatives, uncertainty, design, CFD.

Abstract. Building second derivatives of design functionals involving state equations is a useful mean for addressing uncertainties and robust design. We analyse strategies for deriving second-order derivatives of programs by applying Automatic Differentiation (AD). We consider the application of the two differnetiation modes, Tangent and Reverse, of the AD tool TAPENADE.

1 Introduction

Due to the high complexity reached by computational fluid dynamics codes combined with the rapid advance of computational power, research in the field of CFD design has experienced a large development in the last years, allowing to deal with more and more complex optimization problem. However, high fidelity models are only used for deterministic design, which assumes a perfect knowledge of the environmental and operational parameters. In reality, uncertainty can arise in many aspects of the entire design-production-operational process: from the assumptions done in the mathematical model describing the underlying physical process, to the manufacturing tolerances, and to the operational parameters and conditions that could be affected by unpredictable factors (e.g. atmospheric conditions). Exact and approximate techniques for propagating these uncertainties require additional computational effort but are progressively well established ([Walters and Huyse, 2002], [Putko et al., 2001], [Ghate and Giles, 2006]) and could be applied to many optimization problems in order to improve the robustness of the design (see for instance [Huyse, 2001] for a shape optimization application). In addition, a systematic uncertainty analysis can lead, from one hand to the identification of the key sources of uncertainty which merit further research, and from the other hand to the sources of uncertainty that are not important with respect to a given response.

In optimization problems, uncertainty propagation analysis may concern the study of the *cost functional*

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

where all varying parameters are represented by the *uncertain (i.e. not-deterministic) control variables* $\gamma \in \mathbb{R}^n$, and where the state variables $W = W(\gamma) \in \mathbb{R}^N$ are solution of the (nonlinear) *state equation*

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

It is important to note that the state equation (2) contains the governing PDE of the mathematical model of the physical system of interest (for example the stationary part of the Euler or Navier-Stokes equations) and it can be viewed as an *equality constraint* for the functional (1).

The two main type of *probabilistic* approaches for analyzing the propagation of uncertainties are the Monte Carlo methods [Liu, 2001, Garzon, 2003] and the perturbative methods based on the Taylor expansion (Method of Moments [Putko et al., 2001] and Inexpensive Monte-Carlo [Ghate and Giles, 2006]). Both approaches rely on the fact that we cannot assign a single value to a quantity affected by a random varable, but we should instead keep in account the different values that the quantities under exam could have.

The most straightforward, general and accurate method to study the uncertainty propagation is full nonlinear Monte-Carlo technique, but for CFD computations it is prohibitively expensive in terms of CPU time due to the fact that to have accurate results (i.e. to have a statistic for the uncertainty) we need to solve many times (> 1000) the nonlinear equation (2).

At the opposite, the Method of Moments gives us informations about uncertainty through approximate values of its mean and variance, and requires only one nonlinear simulation (2) and the additional knowledge of the gradient and Hessian of the constrained functional (1), resulting in a great saving in terms of runtime cost respect to the full nonlinear Monte-Carlo. Moreover, the tedious (and error-prone) task of hand-code differentiation can be avoided and replaced in an automatic way using Automatic Differentiation tools (e.g. TAPENADE [Hascoët et al., 2005]).

Therefore, in the following sections we would to describe some approaches and algorithms to compute first and second order derivatives of a constrained functional using Automatic Differentiation.

2 Automatic Differentiation of constrained functionals

We are interested in obtaining the first and second derivatives of a functional j depending on the control $\gamma \in \mathbb{R}^n$, and expressed in terms of a state $W(\gamma) \in \mathbb{R}^N$ as follows:

$$j\colon \gamma \mapsto j(\gamma) := J(\gamma, W) \quad \text{with } W\colon \gamma \mapsto W(\gamma) \text{ such that } \Psi(\gamma, W) = 0 \tag{3}$$

and where $J\colon \mathbb{R}^n \times \mathbb{R}^N \to \mathbb{R}$.

We observe that $W\colon \gamma \to W(\gamma)$ is a function *implicitly* defined through the state equation $\Psi(\gamma, W(\gamma)) = 0$, and the functional $j(\gamma) := J(\gamma, W(\gamma))$ is evaluated at the solution $W(\gamma)$ for the state equation. In general, the solution for the state equation can be found only numerically, i.e. we have a program that takes a value γ_h for the control and gives us the corresponding state W_h that satisfies the constraint $\Psi(\gamma_h, W_h) = 0$. Therefore our problem is how to compute the gradient j' and the Hessian j'' at the point γ_h.

We can consider two different points of view:

- *Implicit (brute-force) differentiation:* it consists in differentiating directly the *implicit* function j as a function of the control variable γ. This means that the entire process, involving the solution algorithm for state equation and the evaluation of the functional, is considered to be implemented by the single program `func_implicit(j,gamma)` and it is differentiated as a whole (Fig. 1).
- *Differentiation of explicit parts:* the second point of view is to consider the solution algorithm for state equation and the functional evaluation as separated processes, and applying differentiation only to the routines which compute *explicit* functions (that are functions implementing the state residual Ψ and the functional J). A typical structure for a program that performs the evaluation of a functional where the state equation $\Psi(\gamma, W) = 0$ is solved with a fixed-point algorithm is given in the Fig. 2, in which the subroutine implementing the state residual is called `state_residuals(psi,gamma,w)` and the routine implementing the evaluation of the functional is called `functional(j,gamma,w)`.

The underlying idea is that an explicit function is implemented by a sequence of arithmetic computations, whereas implicit functions are implemented using solvers and other iterative algorithms. Explicit functions basically have a fixed computational graph, therefore the underlying function can be considered continuous and differentiable and the AD theory is well-founded in this case. The computational graph of implicit functions is dynamic (this is the case where we

have branches or controls depending on active variables), where a small change of the input may change completely this computational graph: the function is then only piecewise-continuous and in correspondence to these discontinuity points we are getting out of the framework for which the AD is fully justified, resulting in the possibility of wrong results. AD should be used with extreme care for programs having dynamic computational graph.

In the first approach, differentiating the entire program implementing j can be performed with either Tangent or Reverse mode (see [Griewank, 2000]). It directly produces a differentiated program, in a black box manner. The risk is that this program is sometimes not reliable and it often exhibits very poor performance.

To analyze this last issue, we observe that since the program implementing j contains the iterative solver method for the state equation, the differentiated program will contain this solver in differentiated form. Let's assume that we need n_{iter} iterations to obtain the nonlinear solution, and that each iteration costs $(1 + c)$, where we assume an unit runtime cost for the evaluation of the residual $\Psi(\gamma, W)$ and a cost $c > 0$ for the remaining part of the iteration (that contains the algorithm for updating the solution to the next step): the total cost is then $n_{\text{iter}}(1 + c)$.

Tangent mode produces a program that we need to apply n times for computing the entire gradient. The cost is $n(n_{\text{iter}}\alpha_T)(1+c)$ where α_T is the overhead associated with the differentiated code with respect to the original one. One has usually $1 < \alpha_T < 4$, see for example [Griewank, 2000]. Further, the memory requirements will be about twice the memory needed by the original code.

With Reverse mode, we are able to obtain the entire gradient with a single evaluation of the differentiated routine. But, the Reverse mode with the Store-All (SA) strategy produces a code which involves two successive parts [Hascoët and Pascual, 2004]:

- a *forward sweep* close to original code,
- a *backward sweep* performed in the reverse order of the original code.

The problem is that the *backward sweep* needs data computed in the *forward sweep*, but in the reverse order. In the SA strategy, these data are stored in a stack during the *forward sweep* (using a `PUSH` function) and taken from the stack during the *backward sweep* (using a `POP` function).

The total cost (in terms of CPU time and memory) strongly depends on the strategy applied by the AD tool to solve the problem of making the intermediate values available in reverse order (see [Griewank, 2000, Hascoët and Pascual, 2004]). For the case of SA strategy, the CPU cost to evaluate the gradient will be $(n_{\text{iter}}\alpha_R)(1+c)$ with $1 < \alpha_R < 5$, i.e. α_R times the original code, but there is an additional cost in memory to store values on the stack. This stack size is proportional to n_{iter} and it can quickly exceed the available memory. For a Recompute-All (RA) strategy the memory will be of the same order as the original routine, but the CPU cost will be $(n_{\text{iter}}^2\alpha_R)(1+c)$, i.e. $(n_{\text{iter}}\alpha_R)$ times the nonlinear solution.

For real large programs, neither SA or RA strategy can usually work (SA requires too much memory and RA requires too much runtime), so we need a special storage/recomputation trade-off in order to be efficient using *checkpoints* (see [Hascoët and Pascual, 2004]). The idea is to store enough variables (*snapshots*) to be able to restart execution of the backward sweep from a given point, in order to reduce the stack size for the SA strategy or the lenght of recomputation sequence for the RA strategy.

Obviously, the runtime cost of SA strategy with checkpointing will be greater than the pure SA strategy (with the benefit of a smaller stack). However in many cases we can keep this cost reasonably low. For example in the case of iterative processes of fixed lenght n_{iter}, it has been shown [Griewank, 1992] that the runtime cost of the differentiated code is of the order of $\sqrt[s]{n_{\text{iter}}}(n_{\text{iter}}\alpha_R)(1 + c)$ and the stack size grows as $\sqrt[s]{n_{\text{iter}}}$ (where s is the number of snapshots available). To the opposite, RA with checkpointing results in a lower runtime (and higher memory requirements) with respect the pure RA.

In many cases, these checkpointing strategies are the only way to go. This is the case for unsteady nonlinear systems, where we have not a steady solution but a time-dependent solution which depends on the initial conditions, and for which we do not know a strategy working without the intermediate-time state variable values. Checkpointing can be applied quasi-automatically by

the Automatic Differentiator or applied by hand-coding (see [Tber et al., 2007] for an application of checkpointing to an Oceanographic code).

In contrast to the brute force approach, we consider the case where the iterative algorithm is a fixed point one, e.g. when we have stationary problems (see the Fig. 2). In this case we can avoid the differentiation of the iterative algorithms (implemented in the routine `flow_solver(gamma,w)` in Fig. 2) that could come from the pseudo-time advancing scheme, but we differentiate only the routines implementing the state residuals computation and the functional evaluation (that we assume not containing any iterative algorithm), and we will use only the values relative to the final state W_h. This strategy results in a differentiated code that is faster and that does not suffer from reliability problems. Moreover, in the context of the fixed-point algorithms the solution does not depend on the initial guess W^0: therefore in the Reverse mode only the final state variable W_h is necessary for the *backward sweep*, resulting in a smaller stack size and therefore in a lower memory requirement.

For the above reasons, in case of fixed-point algorithms, we recommend the differentiation of explicit parts instead of the implicit (brute-force) approach. Moreover, as we will see in the following sections, with an appropriate design of the interfaces of the routine implementing the state residual Ψ and the functional J, we can provide a framework that frees the user from the complex task of organizing the algorithms needed for the gradient and Hessian evaluation.

We present now in more details this last strategy: we go back to the mathematical equations of the constrained functional, then manipulate them to obtain equations for the required derivatives, and from there we deduce our architecture of the differentiated code that avoids the problems described above.

3 First-order derivative

As we have seen in the previous section, to compute the gradient of a function using AD, we can choose between two modes: Tangent and Reverse mode differentiation. Now we want compute the gradient j' of the constrained functional (3) using the differentiation of explicit parts implementing Ψ and J with the two differentiation modes.

3.1 Tangent mode differentiation

It consists in computing the Gâteaux-derivatives of j with respect to each component direction e_i, $i = 1, \dots, n$ ($e_i = (0, \dots 0, 1, 0, \dots, 0)^T$, where 1 is at the i-th component):

$$\frac{dj}{d\gamma_i} = \frac{dj}{d\gamma} e_i = \frac{\partial J}{\partial \gamma} e_i + \frac{\partial J}{\partial W} \frac{dW}{d\gamma} e_i \tag{4}$$

where $\frac{dW}{d\gamma} e_i$ is the solution of the linear system:

$$\frac{\partial \Psi}{\partial W} \frac{dW}{d\gamma} e_i = -\frac{\partial \Psi}{\partial \gamma} e_i \ . \tag{5}$$

In order to get the gradient, (5) must be solved and (4) has to be evaluated at the point (γ_h, W_h) for each vector e_i of the canonical basis, i.e. n times and the main cost is due to the solution of n linearised N-dimensional systems.

If we choose to solve the single system (5) with an iterative method (that could be perfomed using AD without store the matrix $\frac{\partial \Psi}{\partial W}$ into memory, see [Martinelli, 2007]), and the solution is obtained after $n_{\text{iter},T}$ step, the total runtime cost will be of the order of $\alpha_T n_{\text{iter},T}$, i.e. $n_{\text{iter},T}$ evaluation of the matrix-by-vector operation $\frac{\partial \Psi}{\partial W} x$, where we assume that each evaluation costs α_T times the evaluation of the state residual $\Psi(\gamma, W)$ and the cost of the state residual is taken as reference equal to 1. Therefore, the runtime cost of the full gradient will be $n \alpha_T n_{\text{iter},T}$.

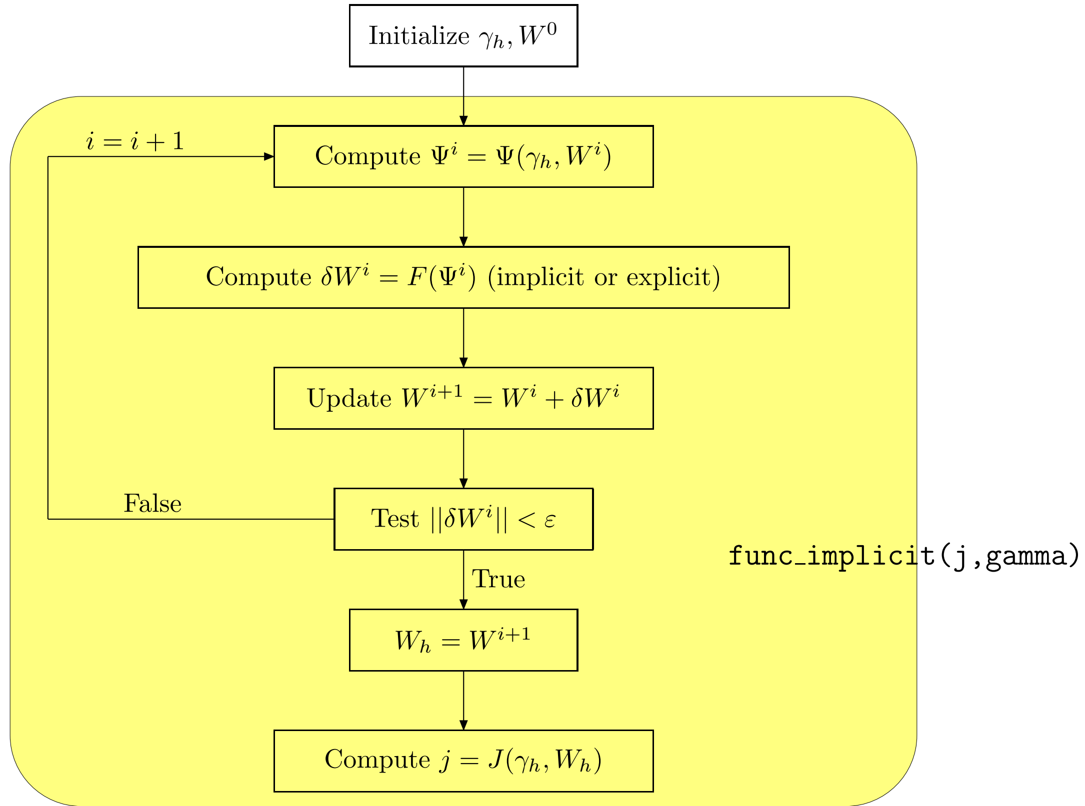

Figure 1: Typical structure for a program that performs the evaluation of a functional with a fixed-point algorithm. In this case the entire process (solution algorithm for the state equation and the evaluation of the functional) is considered to be implemented by the single program `func_implicit(j,gamma)`: the functional is considered as an implicit function only of the control variable γ, i.e. a function containing iterative algorithms. The application of Automatic Differentiation on this kind of programs could give incorrect results, due to the presence of the iterative algorithms for which the AD framework is not fully justified.

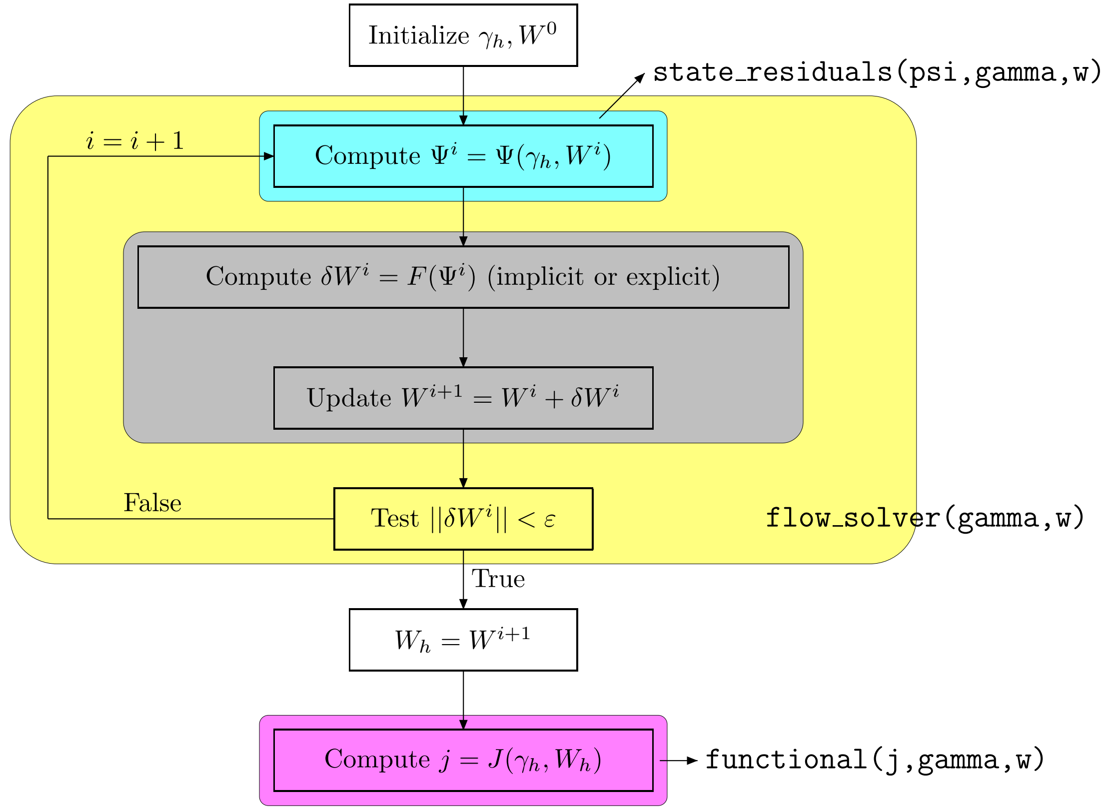

Figure 2: Typical structure for a program that performs the evaluation of a functional with a fixed-point algorithm. The iterative flow-solver can be viewed as the composition of two phases: the evaluation of the residual (performed by the subroutine `state_residuals(psi,gamma,w)`) that does not contain any iterative algorithm, and the evolution (with implicit or explicit methods) to the next step. In our strategy we want to avoid the differentiation of iterative algorithms (for which the application of AD is not fully justified), therefore we differentiate only the state residual and the routine `functional(j,gamma,w)` that performs the evaluation of the functional.

3.2 Reverse mode differentiation

The complete gradient is given by the equation

$$\left(\frac{dj}{d\gamma}\right)^T = \left(\frac{\partial J}{\partial \gamma}\right)^T - \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi \tag{6}$$

where $\Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W) \in \mathbb{R}^N$ (the adjoint state) is the solution of the linear system

$$\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi = \left(\frac{\partial J}{\partial W}\right)^T . \tag{7}$$

It is important to note that the above formulation permits us to obtain all the derivatives needed by (6)-(7) using only Reverse mode differentiation of the programs implementing $J(\gamma, W)$ and $\Psi(\gamma, W)$.

To compute the gradient j' with this approach we need to solve only one linearised N-dimensional system, the adjoint system (7). If we choose to solve the adjoint system with an iterative method, we can apply the same estimate as in the case of the Tangent mode differentiation, but this time the overhead associated with the evaluation of the matrix-by-vector operation $\left(\frac{\partial \Psi}{\partial W}\right)^T x$ with respect to the state residual evaluation will be α_R (and usually $\alpha_R > \alpha_T$) and the number of iterations $n_{\text{iter},R}$ for the convergence of the solution could be different from $n_{\text{iter},T}$ of the previous case (in our experience, the number of iterations needed by GMRES to solve the system $Ax = b$ and $A^T x = c$ are of the same order, see [Martinelli, 2007]): therefore the total runtime cost for the gradient will be $\alpha_R n_{\text{iter},R}$.

From the previous arguments it clearly appears that, if we need to compute the gradient j' only, the Reverse mode is cheaper in terms of CPU time respect to the Tangent mode if $n \gg 1$. Nevertheless, the Tangent mode algorithm in Section 3.1 will be used in the following because it is the basis for the Hessian computation with the Tangent-on-Tangent approach.

4 Second-order derivative

In the same manner as the computation of the gradient, to compute second derivatives we have different possibilities, which are theoretically equivalent, but they differ in the computational cost (and the choice of the best strategy depends, in the end, on the number n of the control variables for the given functional). Moreover, the best strategy depends on which use of the second-order derivative we need, i.e. the best strategy to obtain the full Hessian matrix could be different from the best strategy to obtain its diagonal part or the multiplication of the Hessian matrix by a vector.

The first method to obtain the second-order differentiation of a constrained functional performs two successive Tangent mode differentiations for both the functional and the state residuals and use the adjoint state to compute each single element in the Hessian matrix [Ghate and Giles, 2007]: we call this approach Tangent-on-Tangent (ToT). The second approach (Tangent-on-Reverse, ToR) performs a Tangent mode differentiation of the gradient (6) obtained with Reverse differentiation (Section 3.2).

4.1 Tangent-on-Tangent option

This method was initially investigated by [Sherman et al., 1996] along with various other algorithms, but the publication does not go into the implementation details for a generic fluid dynamic code. Here we present the mathematical background behind the idea and the efficient AD implementation of [Ghate and Giles, 2007] but with a different analysis of the computational cost.

Starting from the i-th element of the gradient (4), we perform another differentiation with respect to the variable γ_k obtaining the i-k element of the Hessian matrix

$$\left(\frac{d^2 j}{d\gamma^2}\right)_{i,k} = \frac{d^2 j}{d\gamma_i d\gamma_k} = D^2_{i,k} J + \frac{\partial J}{\partial W} \frac{d^2 W}{d\gamma_i d\gamma_k} \tag{8}$$

where

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

Differentiating the equation (5) we get

$$D^2_{i,k}\Psi + \frac{\partial\Psi}{\partial W}\frac{d^2W}{d\gamma_i d\gamma_k} = 0 \tag{9}$$

where

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

and e_i (e_k) is the usual vector of the canonical basis with 1 at the i-th (k-th) component and zero otherwise. Substituting the second derivatives of the state with respect to the control variables $\frac{d^2W}{d\gamma_i d\gamma_k}$ in equation (8) from equation (9) we get

$$\begin{aligned}\frac{d^2j}{d\gamma_i d\gamma_k} &= D^2_{i,k}J - \frac{\partial J}{\partial W}\left(\frac{\partial\Psi}{\partial W}\right)^{-1}D^2_{i,k}\Psi \\ &= D^2_{i,k}J - \Pi^T D^2_{i,k}\Psi\end{aligned} \tag{10}$$

where Π is the solution of the adjoint system (7). The i-k element of the Hessian matrix $\left(\frac{d^2j}{d\gamma^2}\big|_{\gamma_h}\right)$ is then obtained evaluating the (10) at the point (γ_h, W_h) solution of the state equation $\Psi = 0$, namely

$$\left(\frac{d^2j}{d\gamma^2}\bigg|_{\gamma_h}\right)_{i,k} = \left(D^2_{i,k}J\right)\big|_{(\gamma_h,W_h)} - \Pi_h^T\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h,W_h)} \tag{11}$$

where $\Pi_h \in \mathbb{R}^N$ is the solution of the adjoint linear system

$$\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^T\Pi_h = \left(\frac{\partial J}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)^T .$$

The n derivatives $\dfrac{dW}{d\gamma_i}$ in the formulas for $D^2_{i,k}J$ and $D^2_{i,k}\Psi$, should be computed (and stored) solving the linear systems

$$\left(\frac{\partial\Psi}{\partial W}\bigg|_{(\gamma_h,W_h)}\right)\frac{dW}{d\gamma_i} = -\left(\frac{\partial\Psi}{\partial\gamma}\bigg|_{(\gamma_h,W_h)}\right)e_i \qquad i = 1,\ldots,n$$

and this task can be performed using an iterative matrix-free method [Martinelli, 2007]. We assume the number of iterations needed for the iterative linear solver to the convergence of the solution to be $n_{\mathrm{iter},T}$, and each iteration calls a tangent-differentiated routine implementing the matrix-by-vector multiplication $\frac{\partial\Psi}{\partial W}x$ whose cost is α_T times the cost of the original routine implementing the evaluation of the state residuals $\Psi(\gamma, W)$.

Implementation. Now the question is: how can we obtain the quantities $\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h,W_h)}$ and $\left(D^2_{i,k}J\right)\big|_{(\gamma_h,W_h)}$ in (11) using Automatic Differentiation? As we will see soon, if we perform two successive Tangent-mode differentiation of the routine implementing Ψ (J) we will able to compute $\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h,W_h)}$ (or the same quantity relative to J) with a single invocation of the resulting double-differentiated routine. Let us suppose that the subroutine computing the state residual equation $\Psi(\gamma, W)$ is `state_residuals(psi,gamma,w)`, where the input variables are `gamma` and `w`, and the output variable is `psi`.

$$\mathtt{state_residuals}(\underset{\downarrow}{\mathtt{psi}}, \overset{\downarrow}{\mathtt{gamma}}, \overset{\downarrow}{\mathtt{w}})$$

Automatic Differentiation in Tangent mode with respect to the input variables `gamma` and `w` builds subroutine:

$$\texttt{state_residuals_d}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psid}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}})$$

that has the additional output $\texttt{psid} = \dot{\Psi} = \left(\frac{\partial \Psi}{\partial \gamma}\right)\dot{\gamma} + \left(\frac{\partial \Psi}{\partial W}\right)\dot{W}$, calling $\texttt{gammad} = \dot{\gamma}$ and $\texttt{wd} = \dot{W}$ the additional input variables.

Now we differentiate the routine `state_residuals_d` in tangent mode considering `psid` as the output variable and with respect to `gamma` and `w`, obtaining

$$\texttt{state_residuals_d_d}(\underset{\downarrow}{\texttt{psi}}, \underset{\downarrow}{\texttt{psid}}, \underset{\downarrow}{\texttt{psidd}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad0}}, \overset{\downarrow}{\texttt{gammad}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd0}}, \overset{\downarrow}{\texttt{wd}}) \tag{12}$$

the additional output of which is

$$\texttt{psidd} = \dot{\dot{\Psi}} = \frac{\partial}{\partial \gamma}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}\right)\dot{\gamma}_0 + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}\right)\dot{W}_0 + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}_0\right)\dot{W} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial W}\dot{W}\right)\dot{W}_0$$

and where $\texttt{gammad0} = \dot{\gamma}_0$ and $\texttt{wd0} = \dot{W}_0$ are additional input variables.

In order to evaluate the term $D^2_{i,k}\Psi$ at the point (γ_h, W_h) we must call the routine (12) with the appropriate arguments, that is:

$$\texttt{state_residuals_d_d}(\underset{\Psi}{\texttt{psi}}, \underset{\dot{\Psi}}{\texttt{psid}}, \underset{\dot{\dot{\Psi}}}{\texttt{psidd}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_k}{\texttt{gammad0}}, \overset{e_i}{\texttt{gammad}}, \overset{W_h}{\texttt{w}}, \overset{\frac{dW}{d\gamma_k}}{\texttt{wd0}}, \overset{\frac{dW}{d\gamma_i}}{\texttt{wd}}) \tag{13}$$

where the derivative of the state variables with respect to the control $\frac{dW}{d\gamma_i}$ is obtained as solution of the linear system (5) and the resulting output variable is

$$\texttt{psidd} = \dot{\dot{\Psi}} = D^2_{i,k}\Psi\big|_{(\gamma_h, W_h)} .$$

Therefore, the Tangent-on-Tangent approach is the application of two successive tangent-mode differentiations: the first differentiation acts on the original routine, the second one acts on the result of the first differentiation.

The same argument applies to the evaluation of the term $D^2_{i,k}J$. In this case, we perform a Tangent-on-Tangent derivative of the routine

$$\texttt{functional}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

and we get

$$\texttt{functional_d_d}(\underset{J}{\texttt{j}}, \underset{\dot{J}}{\texttt{jd}}, \underset{\dot{\dot{J}}}{\texttt{jdd}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_k}{\texttt{gammad0}}, \overset{e_i}{\texttt{gammad}}, \overset{W_h}{\texttt{w}}, \overset{\frac{dW}{d\gamma_k}}{\texttt{wd0}}, \overset{\frac{dW}{d\gamma_i}}{\texttt{wd}}) \tag{14}$$

where the resulting $\texttt{jdd} = \dot{\dot{J}}$ is the value of $\left(D^2_{i,k}J\right)\big|_{(\gamma_h, W_h)}$.

It is useful to note that the n derivatives of the state with respect to the control $\frac{dW}{d\gamma_i}$ must be evaluated and stored *before* any evaluation of $D^2_{i,k}J$ or $D^2_{i,k}\Psi$. If the number of state variables N and/or the number of control variables n are high, the previous strategy could be not applicable. One possible solution for this problem could be to store the vectors $\frac{dW}{d\gamma_i}$ on the hard-disk instead of keeping them into the RAM, but this strategy could have negative impact on the performance of the computation due to the I/O overhead.

Description of the algorithm for the Hessian matrix with the ToT approach. The algorithm to compute the Hessian matrix with the ToT approach can be summarized as follow:

1. compute the state W_h such that $\Psi(\gamma_h, W_h) = 0$;

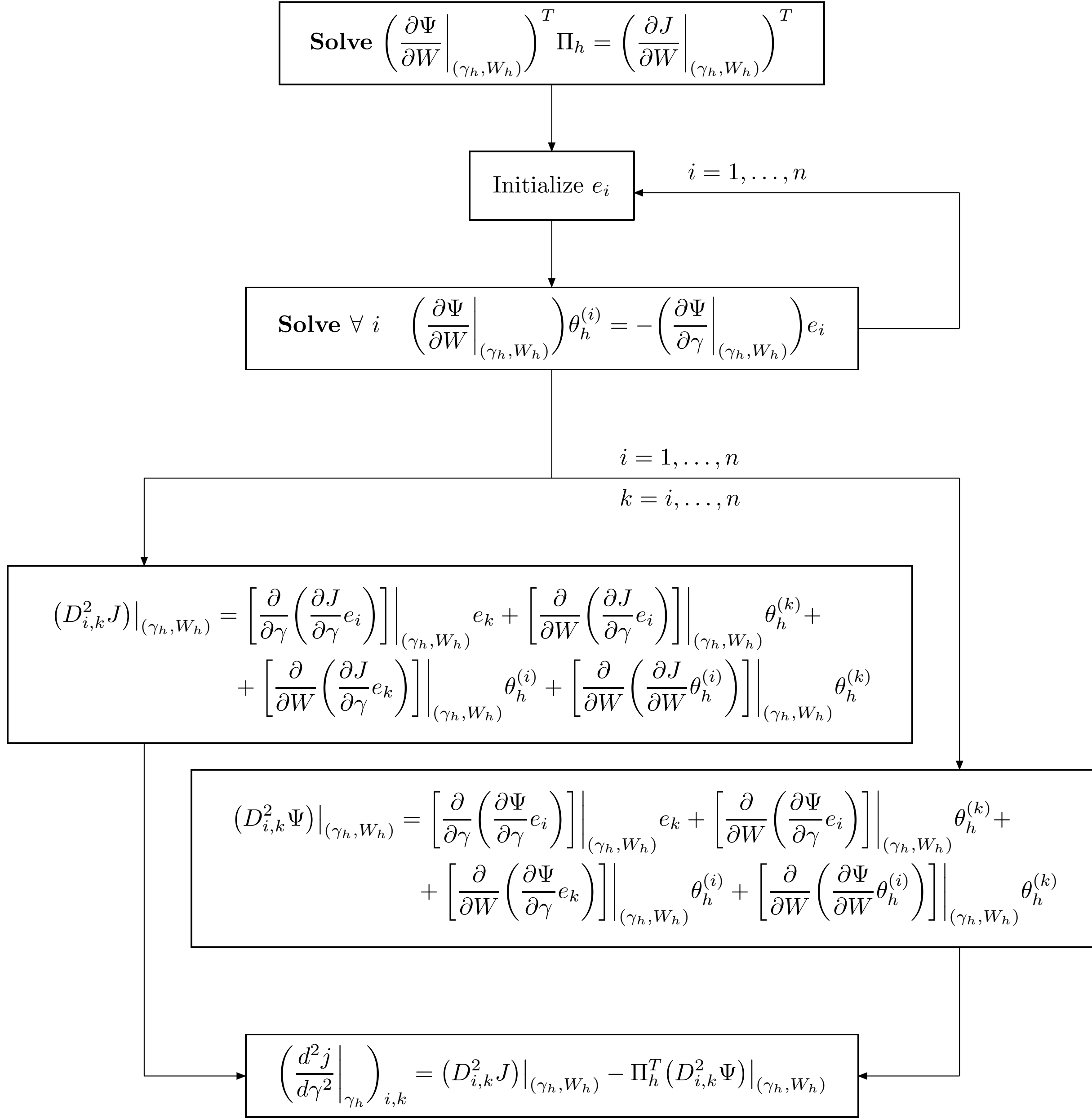

Figure 3: Tangent-on-Tangent algorithm for the full Hessian matrix.

2. compute $\bar{W}_J = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h, W_h)} \right)^T$

3. compute the adjoint state Π_h solving the linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)} \right)^T \Pi_h = \bar{W}_J$;

4. for each element of the canonical basis e_i, $i = 1, \ldots, n$

   (a) compute $\dot{\Psi}_\gamma^{(i)} = \left(\frac{\partial \Psi}{\partial \gamma}\Big|_{(\gamma_h, W_h)} \right) e_i$

   (b) compute (and store) the vector $\theta_h^{(i)}$ solution of the linear system

$$\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h, W_h)} \right) \theta_h^{(i)} = -\dot{\Psi}_\gamma^{(i)} \tag{15}$$

5. for $i = 1, \ldots, n$ and $k = i, \ldots, n$

   (a) compute $\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h, W_h)}$ using the subroutine 13;

   (b) compute $\left(D^2_{i,k}J\right)\big|_{(\gamma_h, W_h)}$ using the subroutine 14;

   (c) compute $\left(\frac{d^2 j}{d\gamma^2}\Big|_{\gamma_h} \right)_{i,k} = \left(D^2_{i,k}J\right)\big|_{(\gamma_h, W_h)} - \Pi_h^T \left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h, W_h)}$.

From equation (11) we see that the ToT approach gives us a single element i-k of the Hessian matrix at time, then using the symmetry property of the Hessian we can compute the full $n \times n$ matrix applying $\frac{n(n+1)}{2}$ time the steps 5a–5c in the algorithm above.

For each element i-k we need to know the vectors $\theta_h^{(i)} = \frac{dW}{d\gamma_i}$ and $\theta_h^{(k)} = \frac{dW}{d\gamma_k}$ obtained solving iteratively (e.g using GMRES [Saad, 1996]) the linear system (15) whose cost is $\alpha_T n_{\mathrm{iter,T}}$ (for simplicity we assume that the number of iterations needed to solve the linear system is independent from the right hand side).

Moreover, the quantity $\left(D^2_{i,k}\Psi\right)\big|_{(\gamma_h, W_h)}$ (step 5a) can be obtained with a single invocation of the differentiated-twice subroutine (13) and its cost is α_T^2 times the cost of the evaluation of the residual $\Psi(\gamma, W)$ (that it is assumed to be unitary). The analogous quantity relative to the functional $\left(D^2_{i,k}J\right)\big|_{(\gamma_h, W_h)}$ (step 5b) can be obtained with a single invocation of the differentiated-twice subroutine (14) and its cost is negligible respect to (13), being negligible the cost to evaluate the subroutine `functional(j,gamma,w)` respect to `state_residuals(psi,gamma,w)`.

Therefore, assuming the adjoint state Π_h to be available, the evaluation of the full Hessian with the ToT approach costs

$$n\alpha_T \left[n_{\mathrm{iter},T} + \frac{(n+1)}{2} \alpha_T \right]$$

and we note that the cost is quadratic respect to the dimension of the control variables but, if the we have $n_{\mathrm{iter},T} \gg n$ the main contribution could be from the cost to solve the n linear systems (15). Therefore, if the dimension of the control variables n is small, the cost is dominated by the solution of the linear systems, otherwise (and assuming the number of iterations n_{iter} to be independent from n) the main cost is due to the differentiated-twice subroutines.

Another important thing to note is the fact that with ToT we can compute the diagonal of the Hessian without computing the extra-diagonal values, due to the fact that the Hessian is built element-by-element: this fact results in a cost for the entire diagonal of

$$n\alpha_T \left[n_{\mathrm{iter},T} + \alpha_T \right]$$

(i.e. one linear system (15) and one evaluation of the differentiated-twice routines in the steps 5a-5c for each element of the diagonal).

Remark 4.1. If we want to evaluate the multiplication of the Hessian by a vector $\delta \in \mathbb{R}^n$, we can evaluate the resulting vector element-by-element, using the Tangent differentiation of derivative (4) along the direction δ instead of e_k. This results in the algorithm in Fig. 4 where we have the single loop over $i = 1, \dots, n$ and where the derivative of the state $\frac{dW}{d\gamma_k} = \frac{dW}{d\gamma} e_k$ is substituted with $\frac{dW}{d\gamma}\delta$. This last quantity can be obtained from the computed $\frac{dW}{d\gamma_k}$ using the fact that the vector δ can be considered as linear combination of vector e_k of the canonical basis. Thus, the resulting cost for the Hessian-by-vector evaluation is

$$n\alpha_T \left[n_{\mathrm{iter},T} + \alpha_T \right] .$$

4.2 Tangent-on-Reverse option

This consists in the direct derivation in any direction e_i, $i = 1, \dots, n$ of the (non-scalar) function:

$$\left(\frac{dj}{d\gamma}\right)^T = \left(\frac{\partial J}{\partial \gamma}\right)^T - \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi$$

where $W\colon \gamma \mapsto W(\gamma)$ such that $\Psi(\gamma, W) = 0$ and $\Pi\colon (\gamma, W) \mapsto \Pi(\gamma, W)$ is the adjoint state defined as

$$\Pi = \left(\frac{\partial \Psi}{\partial W}\right)^{-T} \left(\frac{\partial J}{\partial W}\right)^T .$$

To build the algorithm to compute the Hessian in the present context we need the following

Lemma 4.1 (Hessian-by-vector). *Let $\gamma_h \in \mathbb{R}^n$ and $W_h \in \mathbb{R}^N$ such that $\Psi(\gamma_h, W_h) = 0$ and let*

$$\begin{array}{rcl} j\colon \mathbb{R}^n & \longrightarrow & \mathbb{R} \\ \gamma & \longmapsto & j(\gamma) := J(\gamma, W) \end{array}$$

then the projection of the Hessian $\left(\left.\frac{d^2 j}{d\gamma^2}\right|_{\gamma_h}\right) \in \mathbb{R}^{n\times n}$ along a direction $\delta \in \mathbb{R}^n$ is given by

$$\boxed{\begin{aligned} \left(\left.\frac{d^2 j}{d\gamma^2}\right|_{\gamma_h}\right)\delta = & \left.\left[\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T\right]\right|_{(\gamma_h, W_h)} \delta + \left.\left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^T\right]\right|_{(\gamma_h, W_h)} \theta_h + \\ & - \left.\left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi_h\right)\right]\right|_{(\gamma_h, W_h)} \delta - \left.\left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi_h\right)\right]\right|_{(\gamma_h, W_h)} \theta_h + \\ & - \left(\left.\frac{\partial \Psi}{\partial \gamma}\right|_{(\gamma_h, W_h)}\right)^T \lambda_h \end{aligned}}$$

where $\Pi_h, \theta_h, \lambda_h \in \mathbb{R}^N$ satisfy

$$\left\{ \begin{aligned} & \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)^T \Pi_h = \left(\left.\frac{\partial J}{\partial W}\right|_{(\gamma_h, W_h)}\right)^T \\ & \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right) \theta_h = -\left(\left.\frac{\partial \Psi}{\partial \gamma}\right|_{(\gamma_h, W_h)}\right) \delta \\ & \left(\left.\frac{\partial \Psi}{\partial W}\right|_{(\gamma_h, W_h)}\right)^T \lambda_h = \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^T \delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T \theta_h + \\ & \qquad - \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h\right]\delta - \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h\right]\theta_h . \end{aligned} \right.$$

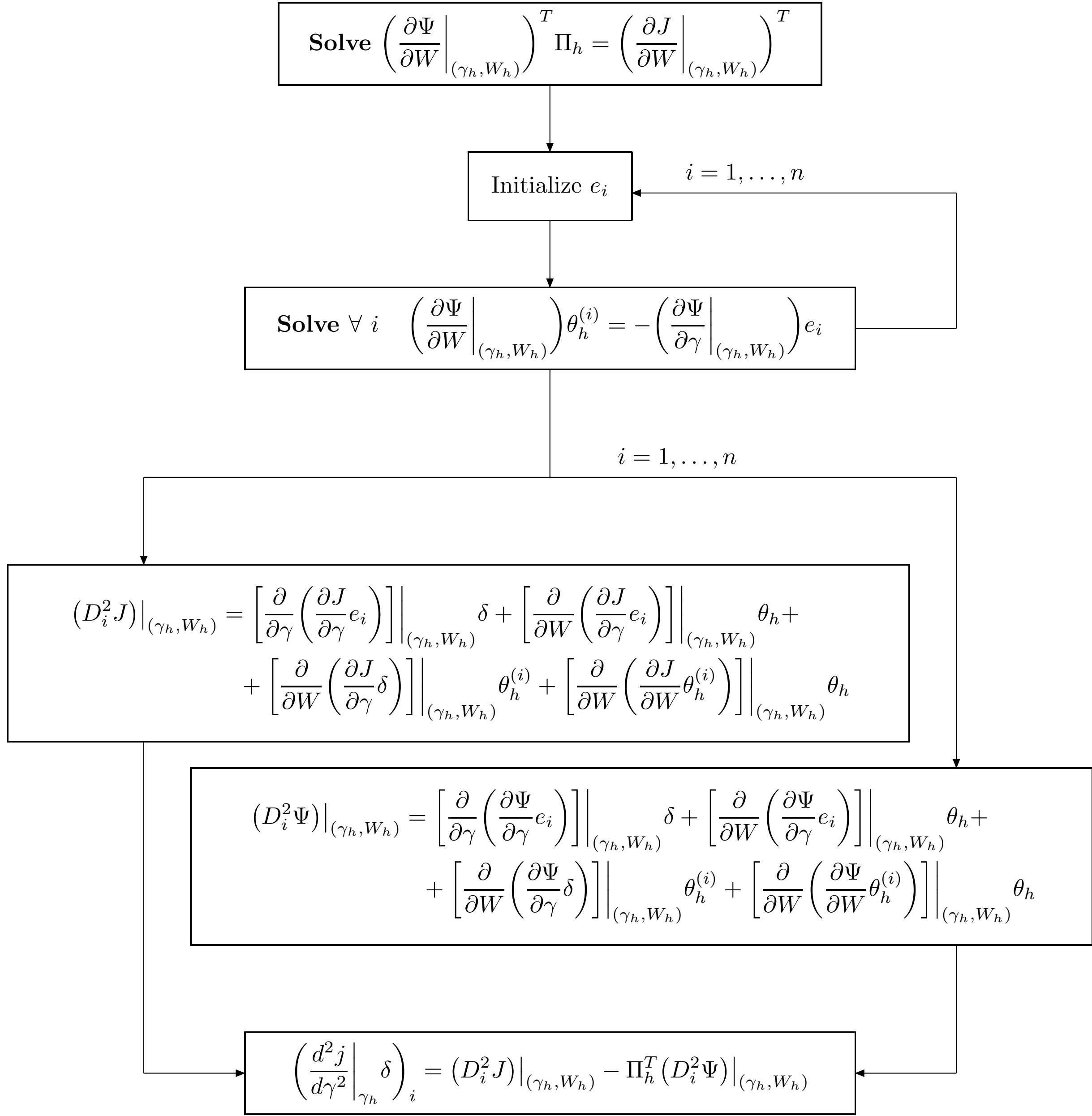

Figure 4: Tangent-on-Tangent algorithm for the Hessian-by-vector multiplication. The vector θ_h is the solution of the linear system $\left(\frac{\partial \Psi}{\partial W}\right)\theta_h = -\left(\frac{\partial \Psi}{\partial \gamma}\right)\delta$ and it could be obtained as linear combination of the vector $\theta_h^{(i)}$, namely $\theta_h = \sum_i^n \delta_i \theta_h^{(i)}$ where δ_i is the i-th component of the vector δ.

Proof. First of all, we note that the tangent derivative along the direction δ of a (n-dimensional) function $j\colon \gamma \mapsto j(\gamma) := J(\gamma, W)$ subject to $\Psi(\gamma, W) = 0$, is given by

$$\begin{aligned} \frac{dj}{d\gamma}\delta &= \frac{\partial J}{\partial \gamma}\delta + \frac{\partial J}{\partial W}\frac{dW}{d\gamma}\delta \\ &= \frac{\partial J}{\partial \gamma}\delta - \frac{\partial J}{\partial W}\left(\frac{\partial \Psi}{\partial W}\right)^{-1}\frac{\partial \Psi}{\partial \gamma}\delta \\ &= \frac{\partial J}{\partial \gamma}\delta + \frac{\partial J}{\partial W}\theta \end{aligned}$$

where $\theta\colon (\gamma, W) \mapsto \theta(\gamma, W)$ is the solution of the linear system

$$\frac{\partial \Psi}{\partial W}\theta = -\frac{\partial \Psi}{\partial \gamma}\delta\ .$$

Now we can perform the tangent derivative (along the direction δ) of $\left(\frac{dj}{d\gamma}\right)^T$

$$\left(\frac{d^2 j}{d\gamma^2}\right)\delta = \frac{d}{d\gamma}\left(\frac{dj}{d\gamma}\right)^T\delta = \frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T\delta - \frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T\Pi\right]\delta\ . \tag{16}$$

The first term is

$$\frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T\delta = \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^T\theta \tag{17}$$

while the second one is

$$\frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T\Pi\right]\delta = \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial \gamma}\right)^T\Pi\right]\delta + \left(\frac{\partial \Psi}{\partial \gamma}\right)^T\frac{d\Pi}{d\gamma}\delta\ . \tag{18}$$

Performing a tangent derivative of the adjoint equation $\left(\frac{\partial \Psi}{\partial W}\right)^T\Pi - \left(\frac{\partial J}{\partial W}\right)^T = 0$ along the direction δ we obtain

$$\begin{aligned} 0 &= \frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T\Pi - \left(\frac{\partial J}{\partial W}\right)^T\right]\delta = \\ &= \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial W}\right)^T\Pi\right]\delta + \left(\frac{\partial \Psi}{\partial W}\right)^T\frac{d\Pi}{d\gamma}\delta - \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^T\delta - \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\theta \end{aligned}$$

and therefore

$$\frac{d\Pi}{d\gamma}\delta = \left(\frac{\partial \Psi}{\partial W}\right)^{-T}\left\{\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^T\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\theta - \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial W}\right)^T\Pi\right]\delta\right\}\ . \tag{19}$$

Putting the (17)–(19) into the (16) we obtain the projection of the Hessian $\frac{d^2 j}{d\gamma^2}$ along a generic direction δ

$$\begin{aligned} \left(\frac{d^2 j}{d\gamma^2}\right)\delta &= \frac{d}{d\gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T\delta - \frac{d}{d\gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T\Pi\right]\delta = \\ &= \frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial \gamma}\right)^T\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial \gamma}\right)^T\theta - \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial \gamma}\right)^T\Pi\right]\delta + \\ &\quad - \left(\frac{\partial \Psi}{\partial \gamma}\right)^T\left(\frac{\partial \Psi}{\partial W}\right)^{-T}\left\{\frac{\partial}{\partial \gamma}\left(\frac{\partial J}{\partial W}\right)^T\delta + \frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\theta - \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial W}\right)^T\Pi\right]\delta\right\}\ . \end{aligned}$$

Evaluating this last expression at the point (γ_h, W_h) and remembering that

$$\left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial x}\right)^T\Pi\right]\Bigg|_{(\gamma_h,W_h)} = \left[\frac{d}{d\gamma}\left(\frac{\partial \Psi}{\partial x}\right)^T\Pi_h\right]\Bigg|_{(\gamma_h,W_h)} = \left[\frac{d}{d\gamma}\left(\left(\frac{\partial \Psi}{\partial x}\right)^T\Pi_h\right)\right]\Bigg|_{(\gamma_h,W_h)} \qquad x = \{\gamma, W\}$$

we obtain

$$\left(\frac{d^2 j}{d\gamma^2}\Big|_{\gamma_h}\right)\delta = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Big|_{(\gamma_h,W_h)}\delta + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Big|_{(\gamma_h,W_h)}\theta_h + \\ - \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}\delta - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h + \\ - \left(\frac{\partial\Psi}{\partial\gamma}\Big|_{(\gamma_h,W_h)}\right)^T\lambda_h$$

where θ_h is the solution of the linear system

$$\left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h))}\right)\theta_h = -\left(\frac{\partial\Psi}{\partial\gamma}\Big|_{(\gamma_h,W_h))}\right)\delta$$

and λ_h is the solution of the linear system

$$\left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^T\lambda_h = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^T\right]\Big|_{(\gamma_h,W_h)}\delta + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\right]\Big|_{(\gamma_h,W_h)}\theta_h + \\ - \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial W}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}\delta - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial W}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h \; .$$

□

If we apply the Lemma 4.1 using $\delta = e_i$ (where $e_i = (0,\dots,1,\dots,0)^T$ is the vector having the only not-zero value at the i-th position, i.e. the i-th element of the canonical basis of $\mathbb{R}^n$), it means that we are computing the i-th column (and, by symmetry, the i-th row) of the Hessian, obtaining

$$\boxed{\begin{aligned}\left(\frac{d^2 j}{d\gamma^2}\Big|_{\gamma_h}\right)e_i = {} & \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Big|_{(\gamma_h,W_h)}e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Big|_{(\gamma_h,W_h)}\theta_h^{(i)} + \\ & - \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}e_i - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial\gamma}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h^{(i)} + \\ & - \left(\frac{\partial\Psi}{\partial\gamma}\Big|_{(\gamma_h,W_h)}\right)^T\lambda_h^{(i)} \; .\end{aligned}} \tag{20}$$

Then, to compute the full Hessian, we need to apply the Hessian-by-vector multiplication (20) to each component of the canonical basis of $\mathbb{R}^n$.

For each $i = 1,\dots,n$, the equation (20) needs the adjoint state Π_h, solution of the adjoint system

$$\left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^T\Pi_h = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^T \tag{21}$$

and the arrays $\theta_h^{(i)}$, $\lambda_h^{(i)}$ solutions of the linear systems:

$$\begin{cases}\left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)\theta_h^{(i)} = -\left(\frac{\partial\Psi}{\partial\gamma}\Big|_{(\gamma_h,W_h)}\right)e_i \\ \left(\frac{\partial\Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^T\lambda_h^{(i)} = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^T\right]\Big|_{(\gamma_h,W_h)}e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\right]\Big|_{(\gamma_h,W_h)}\theta_h^{(i)} + \\ \qquad - \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial\Psi}{\partial W}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}e_i - \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial\Psi}{\partial W}\right)^T\Pi_h\right)\right]\Big|_{(\gamma_h,W_h)}\theta_h^{(i)}\end{cases} \tag{22}$$

where all the derivatives in the equations (20)–(22) are evalued at the final state W_h. Moreover, the second linear system in (22) is of the same type of the adjoint system (21) but with a different right hand side, so we can use the same matrix-free algorithm and the same preconditioner (but with different right hand side) for both equations.

Implementation of the Tangent-on-Reverse derivatives As we have done in Section 4.1 for ToT approach, let us suppose that the subroutine computing the state residual equation $\Psi(\gamma, W)$ is `state_residuals(psi,gamma,w)`, where the input variables are `gamma` and `w`, and the output variable is `psi`.

$$\texttt{state_residuals}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

If we perform a differentiation in reverse mode with respect to the input variables `gamma` and `w` we have

$$\texttt{state_residuals_b}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{psib}}, \overset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammab}}, \overset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wb}})$$

where $\texttt{gammab} = \bar{\gamma}_\Psi = \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}$ and $\texttt{wb} = \bar{W}_\Psi = \left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}$ are the new output variables and calling $\texttt{psib} = \bar{\Psi}$ the additional input variable.

Now we differentiate the routine `state_residuals_b` in tangent mode (with respect to the same input variables `gamma` and `w`) considering `gammab` and `wb` as output variables, obtaining

$$\texttt{state_residuals_b_d}(\underset{\downarrow}{\texttt{psi}}, \overset{\downarrow}{\texttt{psib}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad}}, \underset{\downarrow}{\texttt{gammab}}, \underset{\downarrow}{\texttt{gammabd}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}}, \underset{\downarrow}{\texttt{wb}}, \underset{\downarrow}{\texttt{wbd}}) \tag{23}$$

where $\texttt{gammad} = \dot{\gamma}$, $\texttt{wd} = \dot{W}$ and the second order derivatives are in the variables

$$\texttt{gammabd} = \dot{\bar{\gamma}}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}\right]\dot{W}$$
$$\texttt{wbd} = \dot{\bar{W}}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}\right]\dot{W}\ .$$

In order to solve the equations equations (20)–(22) we should call the routine (23) with the right arguments:

$$\texttt{state_residuals_b_d}(\underset{\Psi}{\texttt{psi}}, \overset{\Pi_h}{\texttt{psib}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_i}{\texttt{gammad}}, \underset{\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi_h}{\texttt{gammab}}, \underset{\dot{\bar{\gamma}}_\Psi}{\texttt{gammabd}}, \overset{W_h}{\texttt{w}}, \overset{\theta_h^{(i)}}{\texttt{wd}}, \underset{\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h}{\texttt{wb}}, \underset{\dot{\bar{W}}_\Psi}{\texttt{wbd}}) \tag{24}$$

where

$$\dot{\bar{\gamma}}_\Psi = \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi_h\right)\right]\Bigg|_{(\gamma_h, W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \Pi_h\right)\right]\Bigg|_{(\gamma_h, W_h)} \theta_h^{(i)}$$
$$\dot{\bar{W}}_\Psi = \left[\frac{\partial}{\partial \gamma}\left(\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h\right)\right]\Bigg|_{(\gamma_h, W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h\right)\right]\Bigg|_{(\gamma_h, W_h)} \theta_h^{(i)}\ .$$

If the subroutine computing the functional $J(\gamma, W)$ is

$$\texttt{functional}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{w}})$$

performing a reverse mode differentiation with respect to `gamma` and `w`

$$\texttt{functional_b}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{jb}}, \overset{\downarrow}{\texttt{gamma}}, \underset{\downarrow}{\texttt{gammab}}, \overset{\downarrow}{\texttt{w}}, \underset{\downarrow}{\texttt{wb}})$$

and then a tangent mode differentiation of the output variables `gammab` and `wb` with respect to `gamma` and `w`

$$\texttt{functional_b_d}(\underset{\downarrow}{\texttt{j}}, \overset{\downarrow}{\texttt{jb}}, \overset{\downarrow}{\texttt{gamma}}, \overset{\downarrow}{\texttt{gammad}}, \underset{\downarrow}{\texttt{gammab}}, \underset{\downarrow}{\texttt{gammabd}}, \overset{\downarrow}{\texttt{w}}, \overset{\downarrow}{\texttt{wd}}, \underset{\downarrow}{\texttt{wb}}, \underset{\downarrow}{\texttt{wbd}}) \tag{25}$$

where $\texttt{jb} = \bar{J}$, $\texttt{gammad} = \dot{\gamma}$, $\texttt{wd} = \dot{W}$ and

$$\texttt{gammabd} = \dot{\bar{\gamma}}_J = \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial J}{\partial\gamma}\right)^T \bar{J}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial J}{\partial\gamma}\right)^T \bar{J}\right]\dot{W}$$

$$\texttt{wbd} = \dot{\bar{W}}_J = \frac{\partial}{\partial\gamma}\left[\left(\frac{\partial J}{\partial W}\right)^T \bar{J}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial J}{\partial W}\right)^T \bar{J}\right]\dot{W}\,.$$

As usual, we call the routine above with the right arguments to obtain needed quantities in the equations (20) and (22), i.e.

$$\texttt{functional_b_d}(\underset{J}{\texttt{j}}, \overset{1.0}{\texttt{jb}}, \overset{\gamma_h}{\texttt{gamma}}, \overset{e_i}{\texttt{gammad}}, \underset{\left(\frac{\partial J}{\partial\gamma}\right)^T}{\texttt{gammab}}, \underset{\dot{\bar{\gamma}}_J}{\texttt{gammabd}}, \overset{W_h}{\texttt{w}}, \overset{\theta_h^{(i)}}{\texttt{wd}}, \underset{\left(\frac{\partial J}{\partial W}\right)^T}{\texttt{wb}}, \underset{\dot{\bar{W}}_J}{\texttt{wbd}}) \tag{26}$$

where

$$\dot{\bar{\gamma}}_J = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Bigg|_{(\gamma_h,W_h)} \theta_h^{(i)}$$

$$\dot{\bar{W}}_J = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^T\right]\Bigg|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\right]\Bigg|_{(\gamma_h,W_h)} \theta_h^{(i)}\,.$$

Description of the algorithm for the Hessian matrix with the ToR approach. The algorithm to compute the Hessian matrix wit the ToR approach can be summarized as follow:

1. compute the state W_h such that $\Psi(\gamma_h, W_h) = 0$;
2. compute $\bar{W}_J = \left(\frac{\partial J}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^T$
3. compute the adjoint state Π_h solving the linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)^T \Pi_h = \bar{W}_J$;
4. for each element of the canonical basis e_i, $i = 1, \ldots, n$:
   (a) compute $\dot{\Psi}_\gamma = \left(\frac{\partial \Psi}{\partial\gamma}\Big|_{(\gamma_h,W_h)}\right) e_i$;
   (b) compute the vector $\theta_h^{(i)}$ solving the linear system $\left(\frac{\partial \Psi}{\partial W}\Big|_{(\gamma_h,W_h)}\right)\theta_h^{(i)} = -\dot{\Psi}_\gamma$;
   (c) compute $\dot{\bar{W}}_J = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial W}\right)^T\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial W}\right)^T\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;
   (d) compute $\dot{\bar{\gamma}}_J = \left[\frac{\partial}{\partial\gamma}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\frac{\partial J}{\partial\gamma}\right)^T\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;
   (e) compute $\dot{\bar{W}}_\Psi = \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial W}\right)^T \Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;
   (f) compute $\dot{\bar{\gamma}}_\Psi = \left[\frac{\partial}{\partial\gamma}\left(\left(\frac{\partial \Psi}{\partial\gamma}\right)^T \Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} e_i + \left[\frac{\partial}{\partial W}\left(\left(\frac{\partial \Psi}{\partial\gamma}\right)^T \Pi_h\right)\right]\Big|_{(\gamma_h,W_h)} \theta_h^{(i)}$;

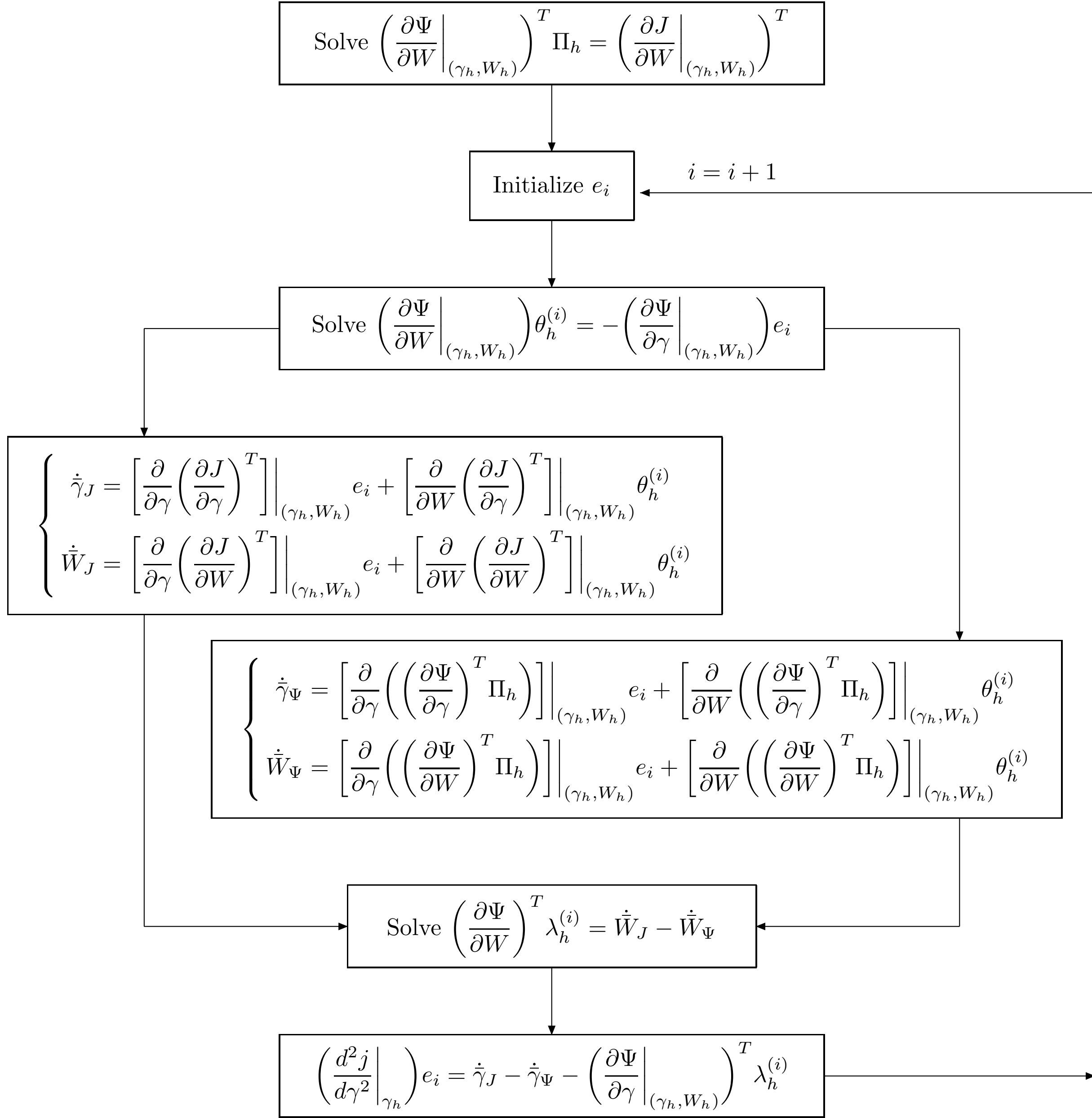

Figure 5: Tangent-on-Reverse algorithm

(g) compute the vector $\lambda_h^{(i)}$ solving the linear system

$$\left(\left. \frac{\partial \Psi}{\partial W} \right|_{(\gamma_h, W_h)} \right)^T \lambda_h^{(i)} = \dot{\bar{W}}_J - \dot{\bar{W}}_\Psi \ ;$$

(h) compute $\bar{\gamma}_\Psi = \left(\left. \frac{\partial \Psi}{\partial \gamma} \right|_{(\gamma_h, W_h)} \right) \lambda_h^{(i)}$;

(i) compute the i-th column (or row) of the Hessian matrix:

$$\left(\left. \frac{dj}{d\gamma} \right|_{\gamma_h} \right) e_i = \dot{\bar{\gamma}}_J - \dot{\bar{\gamma}}_\Psi - \bar{\gamma}_\Psi \ .$$

From the previous algorithm we see that for each column of the Hessian matrix we need to solve 2 linear systems: one is to compute the vector $\theta_h^{(i)}$ (step 4b) and the other is to compute the the vector $\lambda_h^{(i)}$ (step 4g). Moreover, the quantities $\dot{\bar{W}}_J$, $\dot{\bar{\gamma}}_J$ (steps 4c and 4d) can be obtained with a single invocation of the twice-differentiated subroutine (26); while the quantities $\dot{\bar{W}}_\Psi$, $\dot{\bar{\gamma}}_\Psi$ (steps 4e and 4f) can be obtained with a single invocation of the differentiated-twice subroutine (24).

The computational cost for a single Hessian-by-vector multiplication, evaluated with the Tangent-on-Reverse approach, can be estimated with the same assumptions made in Section 4.1, and is due to the following contributes:

- $\alpha_T n_{\mathrm{iter},T}$ for computing the derivatives of the state variables respect to the control $\theta_h^{(i)} = \frac{dW}{d\gamma_i}$ (step 4b), where $n_{\mathrm{iter},T}$ is the number of iterations needed by the matrix-free algorithm to solve the linear system;
- $\alpha_R \alpha_T$ for evaluating the quantities $\dot{\bar{W}}_\Psi$, $\dot{\bar{\gamma}}_\Psi$ (steps 4e and 4f) with the single invocation of the subroutine (24);
- $\alpha_R n_{\mathrm{iter},R}$ for computing the vector $\lambda_h^{(i)}$ (step 4g), where $n_{\mathrm{iter},R}$ is the number of iterations needed by the matrix-free algorithm to solve adjoint systems.

Therefore the full Hessian evaluation with ToR costs

$$n\alpha_T \left(n_{\mathrm{iter},T} + \alpha_R + \frac{\alpha_R}{\alpha_T} n_{\mathrm{iter},R} \right)$$

and we note that the major contribution is due to the solution of the linear systems, usually being the number of iterations to the convergence $\gg \alpha_R$.

Another important thing to note is the fact that with ToR we cannot compute the diagonal of the Hessian without computing the extra-diagonal values, due to the fact that the Hessian is built column-by-column (or, by symmetry, row-by-row) using the Lemma 4.1 on the elements of the canonical basis.

As minor remark, ToR approach for the full Hessian does not need to store the derivatives of the state variables respect to the control $\theta_h^{(i)} = \frac{dW}{d\gamma_i}$ for all $i = 1, \ldots, n$, but it can use the same memory locations for the various $\theta_h^{(i)}$, resulting in a memory saving and in a serialization of the algorithm, while using a different location for each vector results in an easily parallelizable algorithm (each hessian-by-vector multiplication is independent from the others, so each evaluation can be run in parallel).

4.3 Comparison between ToT and ToR

At this point, the natural question arising from the previous analysis is about the choice of the method that is less expensive for a given problem. The cost to evaluate the full Hessian, its diagonal part and the Hessian-by-vector multiplication for the two different strategies is given in

	Hessian (full)	Hessian (diagonal)	Hessian-by-vector
ToT	$n\alpha_T\left[n_{\text{iter},T} + \frac{(n+1)}{2}\alpha_T\right]$	$n\alpha_T\left[n_{\text{iter},T} + \alpha_T\right]$	$n\alpha_T\left[n_{\text{iter},T} + \alpha_T\right]$
ToR	$n\alpha_T\left(n_{\text{iter},T} + \alpha_R + \frac{\alpha_R}{\alpha_T}n_{\text{iter},R}\right)$	—	$\alpha_T\left(n_{\text{iter},T} + \alpha_R + \frac{\alpha_R}{\alpha_T}n_{\text{iter},R}\right)$

Table 1: ToT and ToR comparison. Computational cost for the evaluation of the full $n \times n$ Hessian matrix, only its diagonal part and the Hessian-by-vector multiplication. α_T, α_R are the overheads associated with the tangent- and reverse-mode differentiation for the subroutine implementing the evaluation of the state residual. $n_{\text{iter},T}$, $n_{\text{iter},R}$ are the number of iterations needed for the matrix-free algorithm to solve the tangent and adjoint linear system, respectively. The values in the table do not take into account the runtime cost to compute the adjoint state Π_h, that is assumed to be available. The cost to compute the adjoint state Π_h as solution of the adjoint linear system (7) can be estimated as $\alpha_R n_{\text{iter},R}$.

the Table 1, where we do not take into account the cost to solve the state equation $\Psi = 0$ and to solve the adjoint system (7).

From the algorithms in Sections 4.1 and 4.2 we note that the two approaches to evaluate the full Hessian share a common part, namely the computation of the derivatives of the state variables respect to the control $\frac{dW}{d\gamma_i}$ $(i = 1, \ldots, n)$ as solution of the linear system

$$\left(\frac{\partial \Psi}{\partial W}\bigg|_{(\gamma_h, W_h)}\right)\frac{dW}{d\gamma_i} = -\left(\frac{\partial \Psi}{\partial \gamma}\bigg|_{(\gamma_h, W_h)}\right)e_i \ .$$

This cost appears in Table 1 as the $n\alpha_T n_{\text{iter},T}$ term, therefore the characteristic cost grows as $\frac{n(n+1)}{2}\alpha_T^2$ for the ToT approach and $n\alpha_R\big(\alpha_T + n_{\text{iter},R}\big)$ for ToR. Thus we can say that, *using a single strategy* to compute the full Hessian, ToT has a lower computational cost with respect to ToR if

$$n < 2\frac{\alpha_R}{\alpha_T}\big(1 + \frac{n_{\text{iter},R}}{\alpha_T}\big) - 1.$$

Therefore ToT is cheaper than ToR if the dimension n of the control γ is small. This last result, can be used to build better strategy (i.e. less time-consuming) for the full Hessian *using ToT and ToR for evaluate different parts of the Hessian.* The key idea is to use ToT to build the upper triangular part of the Hessian until the $\bar{n}$-th column and then evaluate the remaining $n - \bar{n}$ columns with ToR (using the Hessian-by-vector multiplication). This mixed strategy costs

$$\begin{cases} n\alpha_T n_{\text{iter},T} + \dfrac{n(n+1)}{2}\alpha_T^2 & \text{for } n \leq \bar{n} \\[2ex] n\alpha_T n_{\text{iter},T} + \dfrac{\bar{n}(\bar{n}+1)}{2}\alpha_T^2 + (n - \bar{n})\alpha_R\big(\alpha_T + n_{\text{iter},R}\big) & \text{for } n > \bar{n} \end{cases}$$

where optimal values for $\bar{n}$ is found to be $\bar{n} = \dfrac{\alpha_R}{\alpha_T}\big(1 + \dfrac{n_{\text{iter},R}}{\alpha_T}\big)$. For a given problem we assume that the values α_T, α_R can be obtained with not too much effort in a preprocessing phase, using program profiling. Much more difficult could be the estimate of $n_{\text{iter},R}$, the number of iterations needed to solve the adjoint linear system, in fact this number depends on many factors: the dimension of the problem itself, the dimension of the Krylov space, the kind of preconditioner used, etc. A comparison for the cost of the full Hessian using different strategies is given in Figure 6 where we assumed $n_{\text{iter},R} = n_{\text{iter},T} = 300$ for the number of iterations needed to solve the linear sysyems (direct and adjoint) and $\alpha_T = 2$, $\alpha_R = 4$ for the overhead associated with the Tangent- and Reverse-differentiation (obtaining the correspondent treshold value for the mixed strategy $\bar{n} = 302$).

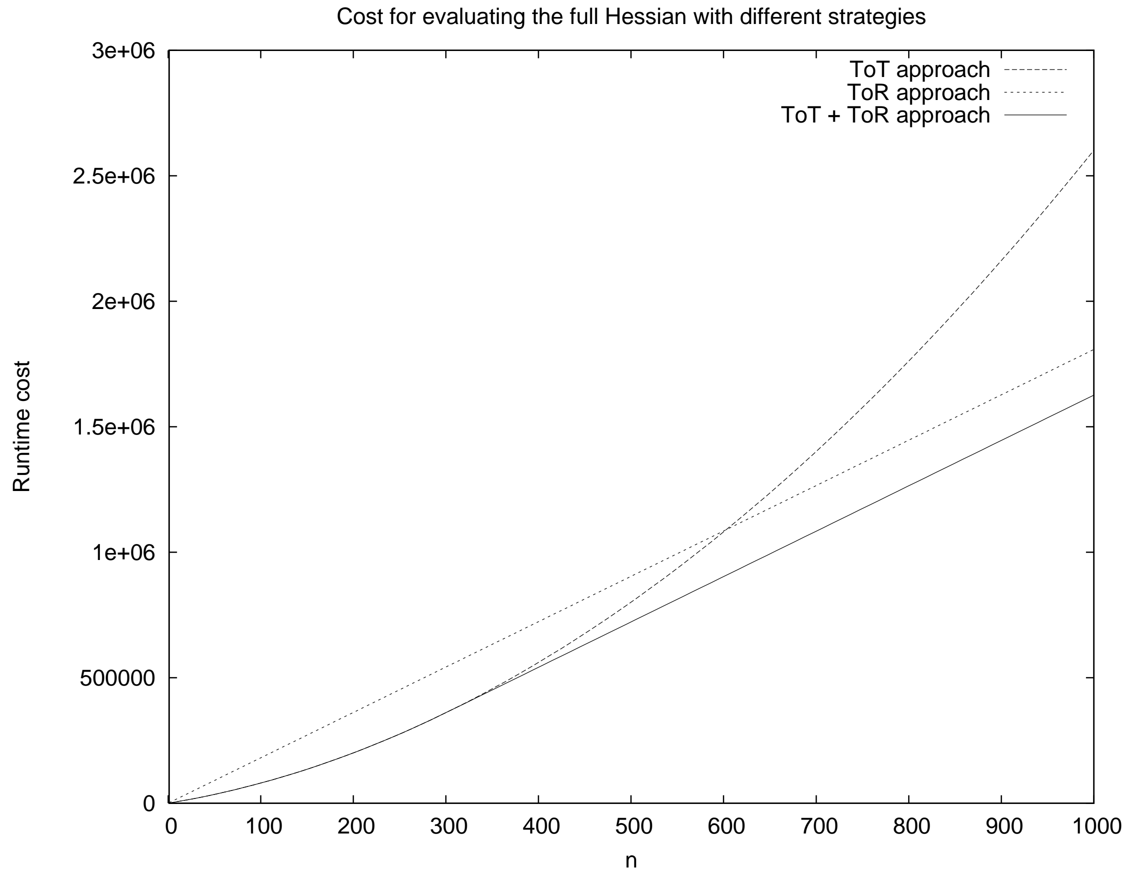

Figure 6: Comparison for the cost of the full Hessian using different strategies as a function of n, the dimension of the control variable γ. We assumed $n_{\text{iter},R} = n_{\text{iter},T} = 300$ for the number of iterations needed to solve the linear systems (tangent and adjoint) and $\alpha_T = 2$, $\alpha_R = 4$ for the overhead associated with the Tangent- and Reverse-differentiation (obtaining the corresponding threshold value for the mixed strategy $\bar{n} = 302$).

5 Stack management issue for ToR

The ToT strategy is very simple to implement and very well managed by TAPENADE: in fact a single Tangent-mode differentiation add only extra code that is executed in the same order of the original: this means that the differentiated code is considered as "normal" code for the second differentiation and therefore this strategy could be used without any difficulty to obtain higher-order derivatives.

Tangent-on-Reverse differentiation raises some Automatic Differentiation issues because of the specific structure of reverse differentiated programs. As we saw in Section 2, programs differentiated in reverse with the Store-All approach make heavy use of `PUSH` and `POP` primitives to store and retrieve intermediate values. In comparison, the Recompute-All approach does not *a priori* use `PUSH` and `POP`'s. However both approaches must use checkpointing for nontrivial applications, which means storing and retrieving snapshots, which boils down to `PUSH` and `POP`'s in the end.

Thus the structure of a reverse-differentiated program is unusual. Indeed we never found a similar structure in the numerical applications that we have differentiated so far. It is true that AD tools claim to be able to differentiate any program, and this claim is motivated by the assumption that a program can be considered as a sequence of istructions that can be identified as a composition of differentiable functions. There exist a few documented limitations to this theory, such as switches in the control-flow that may cause local non-differentiability, but nothing related to stack management. Still, we felt a little anxious when it came to differentiate reverse programs, and rightly so.

Calls to `PUSH` and `POP` communicate values through a stack, which is a *hidden global variable*. Moreover, the source code of `PUSH` and `POP` belongs to an external library and is not given to the AD tool. The classical way to handle these "black-box" routines in AD tools is to provide the tool with just enough information about how the so-called "activity" propagates across the black-boxes. Activity is a boolean information attached to each occurrence of a variable in the code, which is true when the variable has a nontrivial derivative and false otherwise. During differentiation, the AD tool generates calls to differentiated black-box routines, that the end-user must code by hand in the end. This black-box mechanism is very general, and has been used many times on real codes. For example, this is how calls to MPI communication routines are handled by AD tools. However this mechanism requires extreme care, as we will see. Making this mechanism easier and safer needs further research.

First, we characterize how activity propagates across `PUSH` and `POP`. There are two variables involved:

- the first argument of the `PUSH` (resp. `POP`) is the program variable stored (resp. retrieved). Let us call it `V`.
- the hidden stack that keeps all the stored values can be thought of as an array. Let us call it `S`. Initial stack `S` is empty and therefore not active a priori.

A `PUSH` does not change the value nor the activity of `V`. On the other hand if `V` is active, then `S` becomes or remains active. Unfortunately, activity of `S` is a single boolean that mixes or blurs the activity of all stored values, therefore the effect of a `POP` can not be described very accurately. We say that a `POP` never changes the activity of `S`, and it returns an active `V` if and only if `S` is active.

Figure 7 illustrates Tangent-on-Reverse differentiation on a small example code, displayed on the left. The reverse differentiated code has the usual two-sweeps structure, with two `PUSH`/`POP` pairs because the value of `x` needs to be restored. The tangent differentiation that follows begins with activity analysis. Examining the reverse code, one can check that `x` first receives a constant and thus is non active. The first `PUSH`, having non active arguments, need not be differentiated. When `x` is incremented by `c`, it becomes active and so the second `PUSH` makes the stack active. During the reverse sweep, the first `POP` has an active stack and thus returns an active `x` and a still active stack. Since the stack is active, the second `POP` also returns an `x` which is considered active. After activity analysis, tangent differentiation produces the code in the third column. The important point is that there is now an unexpected `PUSH`/`POP_D` pair. In a naive first attempt we implemented `PUSH_D` and `POP_D` using the same stack to store the value and its derivative, and the second `POP_D` causes a segmentation fault because the stack is empty. In more complex examples,

the POP_D might even pop the next top of stack which was expected to be read by a subsequent POP.

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

Figure 7: Tangent-on-Reverse differentiation on a small code. *Left*: Original code, *middle*: Reverse code, *right*: ToR code. Reverse-differentiated variables are shown with a bar above, as in $\overline{x}$. Tangent-differentiated variables are shown with a dot above, as in $\dot{x}$ and $\dot{\overline{x}}$. The reverse code produced with TAPENADE is actually shorter because of static data-flow analysis: the code in light grey is stripped away, but this has no influence on the demonstration.

What has gone wrong? Could it be an error in the tangent differentiation model? To check that, we considered an equivalent reversed program, with the source of PUSH and POP made explicit, using a simple array to store the values. Then the Tangent-on-Reverse program works fine. We notice that it has produced a differentiated array to store differentiated values. Thus, the problem does not lie with the differentiation model, but rather with the hand-written code for PUSH_D and POP_D that use the same stack for x and $\dot{x}$. With two different stacks, the second POP_D finds an empty stack and returns 0.0, which is consistant with the fact that this x is actually non active. Equivalently, we can simply decide that if at any time activity actually propagates across the hidden communication variable, then the communication variable is active right from the beginning. This would turn the first PUSH into a PUSH_D. This is what we did for this work, and the Tangent-on-Reverse strategy now produces correct results.

A much more elegant solution would be to keep the information on matching PUSH/POP pairs in the Reverse code. So we would know when a POP returns a non-active variable. Pairs PUSH_D/POP_D would be reserved for really active variables, therefore reducing the stack size for the Tangent-on-Reverse strategy. We are considering implementing this in TAPENADE.

6 ...putting ToT and ToR into practice

In Sections 4.1-4.2 we have presented two approaches to compute the Hessian and Hessian-related quantities (like its diagonal part or Hessian-by-vector multiplication) of a constrained functional, and at first sight it could appear very complicated to put in practice for codes of industrial complexity-level. This is not (entirely) true.

From the equations involved in the algorithms, we note that all the quantities used in both approaches are combination of derivatives of the state residual Ψ and the functional J, therefore we can implement the algorithms (ToT , ToR, mixed ToT/ToR, matrix-free methods for solving linear systems, routines for validation, etc.) referring to the subroutines for the *generic functions* (and *a priori* not defined yet) $\Psi(\gamma, W)$ and $J(\gamma, W)$ that we have called `state_residual(psi,gamma,w)` and `functional(j,gamma,w)` respectively. In this way we can build (compile) a library that contains only the algorithms above, and when a specific problem has to be solved, the user has to implement the correct residual and the functional and then compute the correspondent differentiated routines.

This approach results in a very flexible scheme and in a rapid application to a given problem: the final user must only

- provide its definitions for the functional and the state residual (in our experience this is the most difficult task, due to the fact that the state residual usually must be extracted from bad-written existing codes);
- run two scripts to drive the differentiation tool TAPENADE (one script is devoted to the first order differentiation, the other one to the second order differentiation);
- compile the resulting differentiated subroutines;
- link them with the library containing the algorithm for the derivatives;
- use the API within to compute the quantities of interests (i.e. gradient, Hessian, etc.);

where the three steps in the middle could be accomplished automatically using `Makefile`.

Following this framework, the user has only to respect one constraint: the subroutines implementing the residual and the functional evaluation must have the same interface used in the library containing the algorithms (in fact the library will refer to the differentiated routine by their interface).

Moreover, this constraint affects the name of the variables: in our examples, the user must use the name `gamma` for the control variables and `w` for the state variables, that are the names we used for the variables in the interface of the subroutines `state_residual` and `functional` referred by the library. The possible complication with this strategy is when the implementation for the state residual (or for the functional) use the dependent and independent variables in global space (i.e. `COMMON` blocks in FORTRAN programs): when this is the case, a possible solution is to copy the state and control variables (that are stored in the global space) to the corresponding `w` and `gamma` variables just *before* the invocation of the differentiated routines, and the inverse assignment (from the local `w` and `gamma` to the global state and control variables) as first instruction inside `state_residual(psi,gamma,W)` and `functional(J,gamma,W)`.

7 TAPENADE commands

In this section we give the TAPENADE commands to perform the different differentations needed by the algorithms (ToT and ToR) presesented in Sections 4.1-4.2.

- `tapenade -d -root state_residuals -outvars "psi" -vars "gamma w" -difffuncname _d -fixinterface`

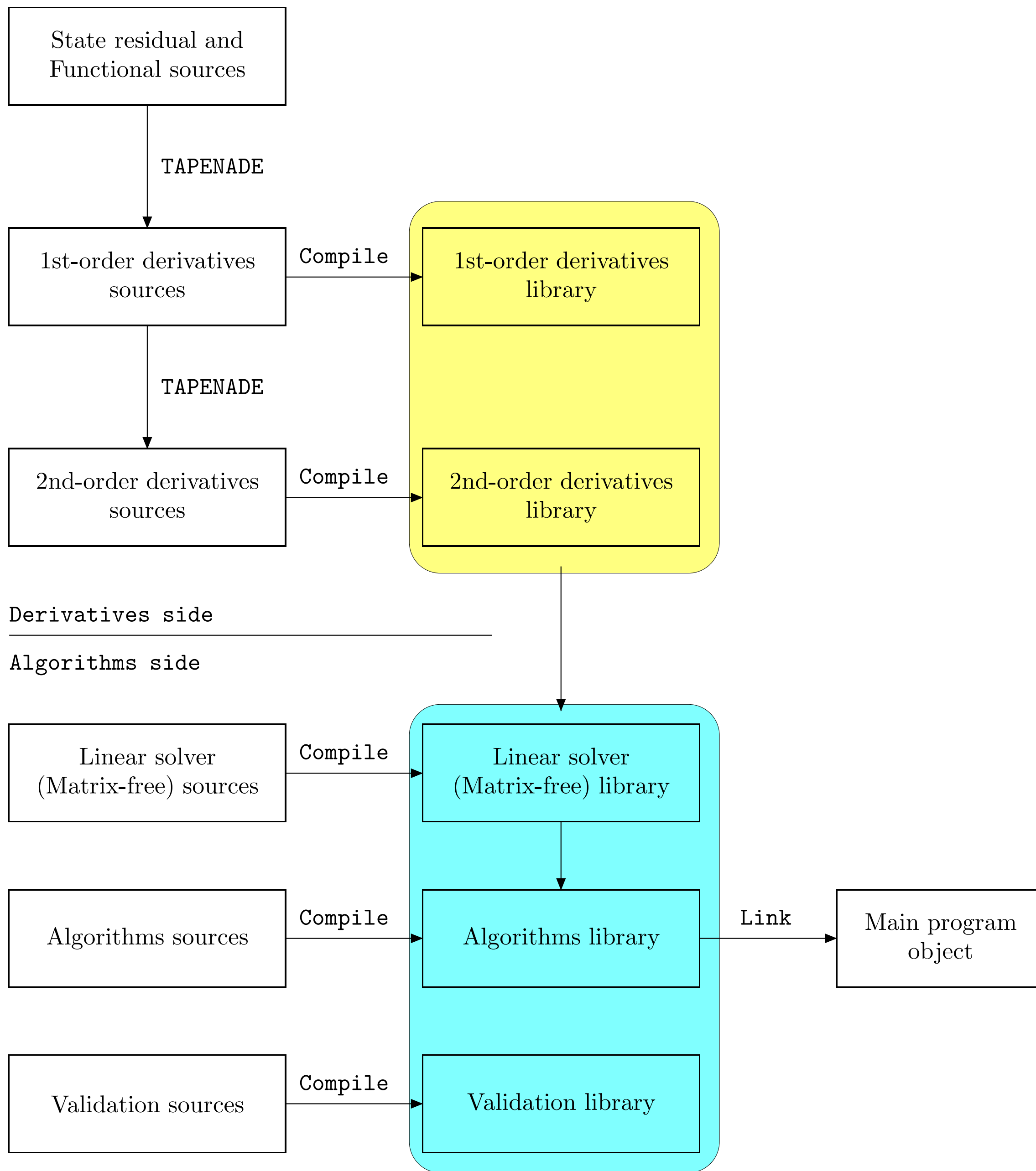

Figure 8: Framework for the implementation of the first and second-order derivatives. For each new definition for functional or the state residual, only the upper part (Derivatives side) is re-evaluated: the algorithms side is untouched. To do that, we are forced to use a fixed interface for the subroutines implementing the functional and the state residual.

Result	`state_residuals_d(psi,psid,gamma,gammad,w,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi} = \dfrac{\partial \Psi}{\partial \gamma}\dot{\gamma} + \dfrac{\partial \Psi}{\partial W}\dot{W} \end{cases}$

- `tapenade -d -root state_residuals -outvars "psi" -vars "gamma" -difffuncname _dgamma_d -fixinterface`

Result	`state_residuals_dgamma_d(psi,psid,gamma,gammad,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi}_\gamma = \dfrac{\partial \Psi}{\partial \gamma}\dot{\gamma} \end{cases}$

- `tapenade -d -root state_residuals -outvars "psi" -vars "w" -difffuncname _dw_d -fixinterface`

Result	`state_residuals_dw_d(psi,psid,gamma,w,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi}_W = \dfrac{\partial \Psi}{\partial W}\dot{W} \end{cases}$

- `tapenade -b -root state_residuals -outvars "psi" -vars "gamma w" -difffuncname _b -fixinterface`

Result	`state_residuals_b(psi,psib,gamma,gammab,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{psib} = \bar{\Psi} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{gammab} = \bar{\gamma}_\Psi = \left(\dfrac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi} \\ \texttt{wb} = \bar{W}_\Psi = \left(\dfrac{\partial \Psi}{\partial W}\right)^T \bar{\Psi} \end{cases}$

- `tapenade -b -root state_residuals -outvars "psi" -vars "gamma" -difffuncname _dgamma_b -fixinterface`

Result	`state_residuals_dgamma_b(psi,psib,gamma,gammab,w,wb)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{psib} = \bar{\Psi} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{gammab} = \bar{\gamma}_{\Psi} = \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi} \end{cases}$

- `tapenade -b -root state_residuals -outvars "psi" -vars "w" -difffuncname _dw_b -fixinterface`

Result	`state_residuals_dw_b(psi,psib,gamma,gammab,w)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{w} = W \\ \texttt{psib} = \bar{\Psi} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{wb} = \bar{W}_{\Psi} = \left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi} \end{cases}$

- `tapenade -d -root state_residual_d -outvars "psid" -vars "gamma w" -difffuncname _d -fixinterface`

Result	`state_residuals_d_d(psi,psid,psidd,gamma,gammad0,gammad,w,wd0,wd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{gammad0} = \dot{\gamma}_0 \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd0} = \dot{W}_0 \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{psid} = \dot{\Psi} = \frac{\partial \Psi}{\partial \gamma}\dot{\gamma} + \frac{\partial \Psi}{\partial W}\dot{W} \\ \texttt{psidd} = \dot{\dot{\Psi}} = \frac{\partial}{\partial \gamma}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}\right)\dot{\gamma_0} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma}\right)\dot{W}_0 + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial \gamma}\dot{\gamma_0}\right)\dot{W} + \frac{\partial}{\partial W}\left(\frac{\partial \Psi}{\partial W}\dot{W}\right)\dot{W}_0 \end{cases}$

- `tapenade -d -root state_residual_b -outvars "gammab wb" -vars "gamma w" -difffuncname _d -ext PUSHPOPGeneralLib -extAD PUSHPOPADLib-fixinterface`

Result	`state_residuals_b_d(psi,psib,gamma,gammad,gammab,gammabd,w,wd,wb,wbd)`
Input	$\begin{cases} \texttt{gamma} = \gamma \\ \texttt{psib} = \bar{\Psi} \\ \texttt{gammad} = \dot{\gamma} \\ \texttt{w} = W \\ \texttt{wd} = \dot{W} \end{cases}$
Output	$\begin{cases} \texttt{psi} = \Psi \\ \texttt{gammab} = \bar{\gamma}_\Psi = \left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi} \\ \texttt{wb} = \bar{W}_\Psi = \left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi} \\ \texttt{gammabd} = \dot{\bar{\gamma}}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial \gamma}\right)^T \bar{\Psi}\right]\dot{W} \\ \texttt{wbd} = \dot{\bar{W}}_\Psi = \frac{\partial}{\partial \gamma}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}\right]\dot{\gamma} + \frac{\partial}{\partial W}\left[\left(\frac{\partial \Psi}{\partial W}\right)^T \bar{\Psi}\right]\dot{W} \end{cases}$

Acknowledgments

This work was supported by the project NODESIM-CFD "Non-Deterministic Simulation for CFD-based Design Methodologies" funded by the European Community represented by the CEC, Research Directorate-General, in the 6th Framework Programme, under Contract No. AST5-CT-2006-030959.

References

V. E. Garzon. *Probabilistic Aerothermal Design of Compressor Airfoils*. PhD thesis, MIT, 2003.

D. Ghate and M. B. Giles. *Inexpensive Monte Carlo uncertainty analysis*, pages 203–210. Recent Trends in Aerospace Design and Optimization. Tata McGraw-Hill, New Delhi, 2006.

D. Ghate and M. B. Giles. Efficient Hessian Calculation Using Automatic Differentiation. Number 2007-4059. AIAA, June 2007. 25th Applied Aerodynamics Conference, Miami (Florida).

A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM Philadelphia, 2000.

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

L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Technical Report 0300, INRIA, Sep 2004.

L. Hascoët, R.-M. Greborio, and V. Pascual. *Computing Adjoints by Automatic Differentiation with TAPENADE*. Springer, 2005.

L. Huyse. Free-form airfoil shape optimization under uncertainty using maximum expected value and second-order second-moment strategies. Technical Report 2001-211020, NASA, Jun 2001. ICASE Report No. 2001-18.

J. S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer-Verlag, 2001. ISBN 0-387-95230-6.

M. Martinelli. *First and second-order derivatives of constrained functionals using Automatic Differentiation and applications in aerodynamic optimal design*. PhD thesis, Scuola Normale Superiore (Pisa) - Université de Nice-Sophia Antipolis, 2007. In preparation.

M. M. Putko, P. A. Newman, A. C. Taylor III, and L. L. Green. Approach for uncertainty propagation and robust design in CFD using sensitivity derivatives. Technical Report 2528, AIAA, 2001.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996. ISBN 0-534-94776-X.

L. L. Sherman, A. C. Taylor III, L. L. Green, and P. A. Newman. First and second-order aerodynamic sensitivity derivatives via automatic differentiation with incremental iterative methods. *Journal of Computational Physics*, 129:307–331, 1996.

M. H. Tber, L. Hascoët, A. Vidard, and B. Dauvergne. Building the Tangent and Adjoint codes of the Ocean General Circulation Model OPA with the Automatic Differentiation tool TAPENADE. Technical report, INRIA, Sep 2007.

R. W. Walters and L. Huyse. Uncertainty analysis for fluid mechanics with applications. Technical Report 2002-211449, NASA, Feb 2002. ICASE Report No. 2002-1.