Ecinads-D5.2: Adjoint differentiation of first version

H. Alcin,V. Pascual, L. Hascoet

Abstract This ECINADS deliverable is devoted in a first phase of the Reverse Automatic Differentiation of the paralle software AIRONUM. We recall the main features of reverse AD and propose a novel method delivering and adjoint state as the output of AD and not as an intermediate to intrusively find into the differentiated code.

1 INTRODUCTION

2 REVERSE-MODE AUTOMATIC DIFFERENTIATION

In this section we describe Automatic Differentiation (AD), focusing on the so-called reverse mode, which will be used intensively. Optimal Control problems make an intensive use of derivatives of the form transposed-Jacobian-times-vector, or, equivalently when the function has a scalar result, the whole Jacobian row vector. We will show why the reverse mode of AD is the most efficient way to get these derivatives.

Then we shall discuss the time-memory tradeoffs that must be made to use reverse-mode AD on large industrial-size applications. We shall show how specific data-flow analyses on the program to be differentiated can help create an efficient differentiated program.

H. Alcin
INRIA, B.P. 93, 06902 Sophia-Antipolis, France, e-mail: Hubert.Alcin@inria.fr

V. Pascual
INRIA, B.P. 93, 06902 Sophia-Antipolis, France,

L. Hascoet
INRIA, B.P. 93, 06902 Sophia-Antipolis, France.

In general, it requires an AD tool to perform AD. Several AD tools propose the reverse mode. For the applications presented in this paper, we used the tool TAPENADE [2], which is developed and distributed by our research team. The data-flow analyses that we describe are implemented and tested inside TAPENADE.

2.1 Principles of reverse AD

Given a source program `P` that evaluates a function F that goes from input x to result $y = F(x)$, AD is able to create a new source program that computes *analytical* derivatives of F. In particular the reverse mode of AD creates a source program $\overline{\mathtt{P}}$ that computes $F'^{*}(x) \cdot \bar{y}$ for any given vector $\bar{y}$. In the special case where y is scalar, if we just take $\bar{y}$ to be one, $\overline{\mathtt{P}}$ returns the row vector $F'(x)$, i.e. the gradient.

To explain the principle of the reverse mode, let's suppose for the sake of simplicity that `P` is a simple list of elementary statements $I_k, k \in [1..p]$. Calling f_k the function implemented by I_k, the F computed by `P` is

$$F = f_p \circ f_{p-1} \circ \ldots \circ f_1 \quad .$$

Using the chain rule, the Jacobian F' of F is:

$$\begin{aligned} F'(x) = \; & (f'_p \circ f_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & .\, (f'_{p-1} \circ f_{p-2} \circ \ldots \circ f_1(x)) \\ & .\, \ldots \\ & .\, (f'_1(x)) \quad . \end{aligned} \tag{1}$$

Let us call for short $x_0 = x$ and $x_k = f_k(x_{k-1})$. The transposed-Jacobian-times-vector product that we need writes:

$$F'^{*}(x).\bar{y} = f_1'^{*}(x_0).f_2'^{*}(x_1).\ldots.f_p'^{*}(x_{p-1}).\bar{y} \tag{2}$$

The reverse differentiated program $\overline{\mathtt{P}}$ will evaluate eq.(2), for any given x and $\bar{y}$, from *right to left*, because matrix$\times$vector products are much cheaper than matrix$\times$matrix products. In theory, the computation cost of $\overline{\mathtt{P}}$ is only a very small constant multiple of the cost of `P`.

In contrast, evaluating the expression in eq.(2) from left to right would result in computing $F'^{*}(x)$ explicitly, and this has a cost which is proportional to the dimension of x. In our examples, this can be a large number. This explains why reverse AD is definitely the most efficient way to compute the derivatives needed for our Optimal Control problems.

Evaluating eq.(2) from right to left results in the following structure of the reverse differentiated program:

```
$\overline{y}_{p-1} := f_p'^*(x_{p-1}).\overline{y}$
...
$\overline{y}_{k-1} := f_k'^*(x_{k-1}).\overline{y}_k$
...
$\overline{y}_0 := f_1'^*(x_0).\overline{y}_1$
return $\overline{y}_0$
```

On this structure the main drawback of reverse AD becomes apparent: the intermediate values x_k are used in the *reverse of their computation order* in P. A typical way to handle this is to run P first, this time storing the intermediate values x_k. This defines the *forward sweep* $\overrightarrow{\mathtt{P}}$, which must be run first. Then comes the *backward sweep* $\overleftarrow{\mathtt{P}}$, which consists of the differentiated instructions above, with additional instructions inserted to progressively restore the intermediate values x_k.

The forward and backward sweeps interact using a stack, and we shall call *restoration* of a variable the couple of statements that push this variable to the stack in $\overrightarrow{\mathtt{P}}$ and pop it from the stack in $\overleftarrow{\mathtt{P}}$. Before each instruction I_k in $\overrightarrow{\mathtt{P}}$, we consider the few variables that I_k may overwrite. For each such variable, if its present value is required in the derivatives of $I_1;\ldots;I_k$, then it must be restored. Detection of the variables required by the derivatives of instructions from I_1 to any I_k is called the *TBR (To Be Restored)* analysis, and is a standard data-flow analysis for AD.

This way, the stack grows reasonably slowly with the size, i.e. execution time, of P. Yet the growth is linear, and for a very large P, radical time-memory tradeoffs must come into play, which we discuss in the next section.

2.2 Data-flow analyses for time-memory tradeoffs

The general time-memory tradeoff is called *checkpointing*, illustrated on Fig. 1. The

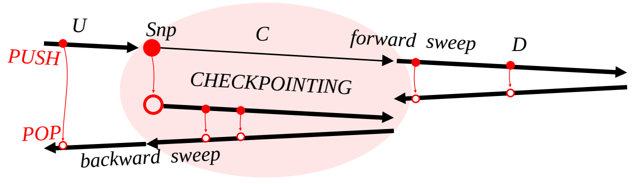

Fig. 1 The Checkpointing time-memory tradeoff

forward sweep goes from left to right, the backward sweep goes from right to left. Each I_k is vertically aligned with its derivative I_k'. Suppose P is split in three successive fragments U, C, and D. Checkpointing C amounts to running C without any restoration push. When the backward sweep reaches back fragment C, the intermediate values are missing. To keep things going, C is run a second time, now like a real forward sweep with the push statements, and then the backward sweep can resume execution till the end.

Duplicated execution of `C` obliges us to save a sufficient number of variables, called a *Snapshot* (*Snp*). Snapshots are usually smaller than the total number of push performed by `C`. All in all a good choice of checkpoints, most probably nested, results in a stack size that grows only like the logarithm of the size of `P`, at the cost of repeated executions that make the execution time increase, by a factor which is also of the order of the logarithm of the size of `P` [1].

Because it is essential to keep the stack size low, we studied the checkpoint mechanism, looking for minimal *snapshots*. Let's go back to Fig. 1 and find what must be in the snapshot *Snp*. The goal is that the second execution of `C` runs exactly like the first. Using standard data-flow analysis terminology, a sufficient condition is that the **use** set of C is not overwritten between its two executions. In other words, execution of $\mathtt{push}(Snp);\mathtt{C};\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}};\mathtt{pop}(Snp)$, must modify no variables in $\mathbf{use}(\mathtt{C})$. Introducing the **out** set of variables possibly modified by a piece of code, the constraint writes:

$$\mathbf{out}\left(\mathtt{push}(Snp);\mathtt{C};\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}};\mathtt{pop}(Snp)\right)\cap\mathbf{use}(\mathtt{C})=\emptyset \qquad (3)$$

Classically, the **out** sets of successive code fragments accumulate. However, the push and pop pairs remove variables from the **out** sets. Therefore eq.(3) rewrites as:

$$\left(\mathbf{out}(\mathtt{C})\cup\mathbf{out}(\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}})\right)\setminus Snp\cap\mathbf{use}(\mathtt{C})=\emptyset \qquad (4)$$

And the smallest *Snp* that obeys this constraint is:

$$Snp=(\mathbf{out}(\mathtt{C})\cup\mathbf{out}(\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}}))\cap\mathbf{use}(\mathtt{C}) \qquad (5)$$

Now, we observe that the **out** set of a forward-backward pair such as $\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}}$ depends on the set **req** of required variables imposed on it by the *TBR* analysis. Indeed, if variable v is added into **req**, then if `D` modifies v, $\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}}$ must restore it. In any case, v is removed from $\mathbf{out}(\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}})$. Therefore we have two options:

- **eager snapshot:** we keep the **req** set before `D` to the **req** before `C`, i.e. the variables required by $\overleftarrow{\mathtt{U}}$ derivatives of `U`.
- **lazy snapshot:** we add to the **req** set before `D` all the variables in $\mathbf{use}(\mathtt{C})$, and then we know that

$$\mathbf{out}(\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}})\cap\mathbf{use}(\mathtt{C})=\emptyset \qquad (6)$$

  and *Snp* is reduced to $\mathbf{out}(\mathtt{C})\cap\mathbf{use}(\mathtt{C})$, at the expense of more restorations inside $\overrightarrow{\mathtt{D}};\overleftarrow{\mathtt{D}}$.

Experimental measurements show that the lazy snapshot option generally performs better, although this depends on the code. Table 1 shows the effect of the two options on the computation of the gradient of a classical 2D Navier-Stokes solver. We observe an interesting 25% gain in memory for the lazy snapshot option. CPU time is also improved marginally, probably because less memory traffic also means less CPU time.

Snapshot:	**eager**	**lazy**
Memory (Mbytes)	248.1	184.7
CPU (seconds)	25.2	22.3

Table 1 Comparison of **eager** and **lazy** snapshot strategies on the gradient of a 2D Navier-Stokes solver

Whatever the option chosen, this definition of the snapshot correctly handles successive checkpoints. Suppose that the D program fragment is split again, to feature a second checkpoint C2. Suppose that C uses a variable v but doesn't modify it, whereas C2 uses and modifies v. v is not modified elsewhere. In other words:

$$v \in \mathbf{use}(\texttt{C});\;\; v \notin \mathbf{out}(\texttt{C});\;\; v \in \mathbf{use}(\texttt{C2});\;\; v \in \mathbf{out}(\texttt{C2})$$

Equation (5) tells us that $v \in Snp(\texttt{C2})$. If we use eager snapshots, then with the help of a good "**out**" analysis, we find that $v \notin \mathbf{out}(\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}})$ because $v \in Snp(\texttt{C2})$, and thus $v \notin Snp(\texttt{C})$. On the other hand if we use lazy snapshots, $v \notin Snp(\texttt{C})$ simply because $Snp(\texttt{C})$ is now only $\mathbf{out}(\texttt{C}) \cap \mathbf{use}(\texttt{C})$, even without the need for a good "**out**" analysis on $\overrightarrow{\texttt{D}};\overleftarrow{\texttt{D}}$.

This is particularly important for the derivatives of the assembly phases, in which the current state variables are in general used at several places to compute the residuals, and are only modified once, at the end of the (pseudo-)time step, to hold the next state.

3 Why and how shall we get the adjoit?

3.1 *Problème continu*

We consider a system governed by the Navier-Stokes equations of a compressible gas, with a state variable W:

$$W = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{pmatrix}.$$

The state equation writes:

$$\Psi(W) = 0$$

with

$$\Psi(W) = W_t + div\,(\mathscr{F}_E(W) + \mathscr{F}_V(W)) \quad sur\;\; \Omega \times]0,T[$$

+ initial and boundary conditions.

We also introduce a functional

$$j = (g, W) \quad , \quad g = g(x,t).$$

3.2 Discrete problem

Let us denote by W_h the discrete state solution.

The discretised Navier-Stokes system is written:

$$\Psi_h(W_h) = 0$$

3.3 The mesh adaptation problem

We want to finde a fixed optimal mesh $\mathscr{M}_{opt}$

- Minimising the error on functional: $\delta j(\mathscr{M}) = |j(W) - j(W_{\mathscr{M}})|$.

  The optimal metric method
  We define a *continuous mesh* on a computational domain $\Omega \subset R^3$

$$M = (\mathscr{M}(x))_{x \in \Omega}$$

from a positive symetric matrix $\mathscr{M}(x)$ defining a Riemannian metric.

Ther the initial problem reduces to the research of a continuous optimal metric.

Optimal metric Lemma : The optimal metric is expressed in terms of state and adjoint state:

$$\mathscr{M}_{opt} = Funct(W, W^*)$$

where W^* is the adjoint state, solution of:

$$(\frac{\partial \Psi}{\partial W}|_W)^* W^* = (\frac{\partial J}{\partial W}|_W)^*$$

In practise, W and W^* are approximated by their discretised counterpart W_h and W_h^*

Optimal metric for the Euler model Let us introduce

$$\mathbf{H}(\mathbf{x}) = \sum_{n=1}^{m} \sum_{j=1}^{5} \left([\Delta t]_j(\mathbf{x})) + [\Delta x]_j(\mathbf{x}) + [\Delta y]_j(\mathbf{x}) + [\Delta z]_j(\mathbf{x}) \right),$$

$$[\Delta t]_j(\mathbf{x})) = \int_0^T \left| W_j^*(\mathbf{x},t) \right| \cdot \left| H((W_{j,t}))(\mathbf{x},t) \right| \, dt,$$

$$[\Delta x]_j(\mathbf{x}) = \int_0^T \left| \frac{\partial W_j^*}{\partial x}(\mathbf{x},t) \right| \cdot \left| H(\mathscr{F}_1(W_j))(\mathbf{x},t) \right| \, dt,$$

$$[\Delta y]_j(\mathbf{x}) = \int_0^T \left| \frac{\partial W_j^*}{\partial y}(\mathbf{x},t) \right| \cdot \left| H(\mathscr{F}_2(W_j))(\mathbf{x},t) \right| \, dt,$$

$$[\Delta z]_j(\mathbf{x}) = \int_0^T \left| \frac{\partial W_j^*}{\partial z}(\mathbf{x},t) \right| \cdot \left| H(\mathscr{F}_3(W_j))(\mathbf{x},t) \right| \, dt \; .$$

where W_j^* is the j-th component of the adjoint array W^* and $H(\mathscr{F}_i(W_j))$ the Hessian of the j-th component of the array $\mathscr{F}_i(W)$.

The optimal metric writes:

$$\mathscr{M}_{opt}(\mathbf{x}) = C\,\det(|\mathbf{H}(\mathbf{x})|)^{-\frac{1}{5}}\,|\mathbf{H}(\mathbf{x})|$$

where the constant C depends on the prescribed number N of nodes:

$$C = N^{\frac{2}{3}} \left(\int_\Omega \det(|\mathbf{H}(\mathbf{x})|)^{\frac{1}{5}} d\mathbf{x} \right)^{\frac{2}{3}} .$$

3.4 Evaluation of adjoint through AD

We describe now how the adjoint will be obtained from particular diffferentiated code. The adjoint state W_h^* is the solution of the linear system:

$$(\frac{\partial \Psi_h}{\partial W_h}|_{(\kappa,W_h)})^* W_h^* = (\frac{\partial J}{\partial W_h}|_{(\kappa,W_h)})^* .$$

Computation of the adjoint We introduce ab artificial parameter κ:

$$j_h(\kappa) = (g, W_h(\kappa))$$
$$\bar{\Psi}_h = \Psi_h(\kappa, W_h(\kappa)) + \kappa$$

- $\frac{\partial j}{\partial \kappa} = (g, \frac{\partial W(\kappa)}{\partial \kappa})$
- By the implicit function theorem:

$$\frac{dW(\kappa)}{d\kappa} = - \frac{\Psi(\kappa, W(\kappa))}{\partial W}^{-1} \frac{\Psi(\kappa, W(\kappa))}{\partial \kappa}$$

thus
$\frac{\partial j}{\partial \kappa} = (g, -\frac{\Psi(\kappa,W(\kappa))}{\partial W}^{-1} \frac{\Psi(\kappa,W(\kappa))}{\partial \kappa})$

- Then putting $\bar{\Psi} = \Psi + \kappa$ on a
  $\frac{\partial j}{\partial \kappa}.\delta = (g, \frac{\partial W}{\partial \kappa}\delta) = -(\frac{\partial \bar{\Psi}}{\partial W}^{-*} g, \frac{\partial \bar{\Psi}}{\partial \kappa}\delta)$
  now $W^* = \frac{\partial \Psi}{\partial W}^{-*} g = \frac{\partial \bar{\Psi}}{\partial W}^{-*} g$ d 'où
  $\frac{\partial j}{\partial \kappa}.\delta = -(W^*, \delta)$

Lemma : with :

$$j_h(\kappa) = (g, W_h(\kappa))$$

$$\Psi_h(\kappa, W_h(\kappa)) = \kappa$$

we have: $w_h^* = -\frac{dj_h}{d\kappa}|_{\kappa=0}$.

Structure of the initial code
We need to define (if not already available) a routine, the "top routine" having:

- g as an input parameter, but not to be differentiated,
- κ as an"independent input variable" for the differentiation,
- $j_h = (g, W_h)$ as "dependent output variable", *i.e.* the output the derivative of which with respect to the independent input variable is to be computed.

Size of arrays
Assuming that the state W is made of *ktmax* instantaneous fields, each of them being of dimension $ns \times 5$, where *ns* is the number of nodes,

- the parameter g is of dimension $ktmax \times ns \times 5$.
- κ is of dimension $ktmax \times ns \times 5$.
- the adjoint state W_h^* is of dimension $ktmax \times ns \times 5$.
- the state W_h will need to be stored, with a dimension $ktmax \times ns \times 5$.

Choice of inverse mode It is justified by the following remarks:

- j_h maps R^p into R and therefore so does its differential applied to a given value.
- The computation of the adjoint- differential of j_h is efficient since it maps R^p into R.

4 PRELIMINARY CONCLUSIONS

Several important milestones in the obtention of an adjoint by reverse differentiation have been identified. New difficulties arose in the application of the novel F90 version of TAPENADE to a new but complex CFD software. The next step will address the efficient adjoint differentiation of the main routines of mpi library.

5 Acknowledgements

This work has been supported by French National Research Agency (ANR) through COSINUS program (project ECINADS n^o ANR-09-COSI-003). HPC resources from GENCI-[CINES] (Grant 2010-x2010026386 and 2010-c2009025067) are also gratefully acknowledged.

References

1. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.
2. L. Hascoet and V. Pascual. Tapenade 2.1 user's guide. Technical Report 0300, INRIA, 2004.