Ecinads-D5.1:Direct differentiation of first version

H. Alcin,V. Pascual, L. Hascoet

Abstract The ECINADS projet aims at the efficient solution of state and adjoint systems for mesh adaptation and control. *In fine*, the adjoint is obtained by reverse differentiation. We first consider and shortly synthetise here the application of Automatic Differentiation in direct mode to the CFD code AIRONUM.

1 INTRODUCTION

For a long time, industrial CFD numerical studies have been addressing simulation quasi-exclusively. Now more and more studies address the next step, i.e. Optimal Control problems, starting from some existing simulation code. This can be done with any kind of simulation code, although nowadays this is still limited in most cases to steady models. Optimal Control on genuine unsteady simulations still proves very expensive in an industrial context. Furthermore, Optimal Control methods can also address other questions, such as the control of numerical errors that will be describe in a forthcoming report.

Development of an Optimal Control application starts from an existing Simulation application, and reuses key parts of it. Let us identify these parts. Given a set of parameters, a simulation software gives a prediction of a physical process. For instance a "numerical wind tunnel" predicts the flow of air around a plane shape, using only computation. The predicted flow is the unique solution of a (set of) mathematical equation which we call the *state equation*, according to Optimal Control

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.

terminology. Numerical resolution of the discretized state equation involves in fact two important parts.

- One is the assembling part: for given arbitrary values of the state variables W, and using values of external parameters γ (e.g. the geometry), it computes a *residual* array, which reflects how the state variables satisfy the state equation. It is therefore

  $$\textbf{state-assembler: } (\gamma, W) \mapsto \Psi(\gamma, W)$$

  where Ψ is in fact the left-hand side of the discretized state equation

  $$\Psi(\gamma, W) = 0 \ \ .$$

- The other is the resolution algorithm: for the given fixed external parameters γ, it uses the residual returned by the state-assembler to produce the state solution $W(\gamma)$ that nullifies the residual (or at least makes it sufficiently small).

  $$\begin{aligned} &\textbf{state-resolution: } \gamma \mapsto W(\gamma) \\ &\qquad\qquad\qquad\qquad \text{such that } \Psi(\gamma, W(\gamma)) = 0. \end{aligned}$$

  This is most often done iteratively, by incremental modifications of an arbitrary initial state, each modification driven by the residual for the current state.

To turn a simulation application into an Optimal Control application requires an additional ingredient, the *objective functional* that will evaluate a scalar cost for any possible parameters and state. Given a set of parameters γ, which we now view as *control parameters*, and given the corresponding solution state $W(\gamma)$, it computes one (or several) optimization criterion, i.e. the value of the objective functional for these control parameters and state. This objective functional takes into account all industrial targets and constraints for a given process or product.

$$\textbf{objective-assembler:} (\gamma, W) \mapsto J(\gamma, W)$$

The goal of Optimal Control is to find control parameters γ which will make the objective functional smaller.

When the size of the parameter array is small, many methods can find the minimum easily enough. However for large numbers of control parameters, approaches that use analytic gradients become necessary. We advocate the following strategy to obtain these gradients:

- from the state-assembler, develop a **gradient-assembler** that computes the residual corresponding to derivatives of the state equation. This part of the strategy relies on *Automatic Differentiation* (AD).
- develop an adequate resolution algorithm for the gradient, that uses the gradient-assembler to iteratively find the requested gradient.

In this deliverable, we analyze this strategy to compute gradients, which ultimately yields a mesh adaptation and control loop. One objective is to maximize the re-use of code from simulation codes into the optimization code. This strategy strongly relies on Automatic Differentiation. The direct differentiation mode of TAPENADE has been applied to a whole software written in F90 and mpi. This section addresses two issues in this differentiation. First we describe one among the important difficulties we encountered, the problem of managing dynamic allocations. Second we give in short the accuracy validation of the code obtained.

2 The TAPENADE program differentiator

TAPENADE is an Automatic Differentiation Engine developed at INRIA Sophia-Antipolis by the TROPICS team. TAPENADE can be utilized as a server (JAVA servlet), which runs at INRIA Sophia-Antipolis. The current address of this TAPENADE server is

`http://www-tapenade.inria.fr:8080/tapenade/index.jsp,`

The Tapenade server. TAPENADE can also be downloaded and installed locally as a set of JAVA classes (JAR archive). In that case it is run by a simple command line, which can be included into a Makefile. It also provides you with a user-interface to visualize the results in a XHTML browser.

In the work presented here, we shall use the direct mode of differentiation. 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 direct 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

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

Using the chain rule, the direct mode gives the Jacobian F' of F as:

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

In direct AD in general and in particular in TAPENADE, each instruction of the form:

$$y \mapsto f_p(y)$$

is replaced by the coucle ogf instructions:

$$y \mapsto f_p(y)$$

$$(y,dy) \mapsto f'_p(y)dy$$

3 Differentiation of a complex F90 software

3.1 The code AIRONUM

AIRONUM is CFD experimental software written in FORTRAN 95 and using the mpi communication library. It computes turbulent compressible flows with a Large Eddy Simulation model based on a Variational Multiscale formulation and a statistical model based on the k-epsilon Reynolds Average Navier-Stokes model. Both modelisation are combined in a hybrid RANS-LES formulation. The state equation is written:

$$\Psi_h(W_h) = 0$$

where Ψ_h holds for the equation residual, driven to zero after resolution, and W_h for the state variable, solution of the state equation after the residual is driven to zero. Then we compute a scalar output functional :

$$j_h = (g, W_h)$$

where g is a prescribed observation vector.

For differentating AIRONUM, we have introduced a source term κ in the state equation residual

$$\Psi_h(W_h) - \kappa = 0$$

therefore the state variable is an implicit function of κ:

$$W_h = W_h(\kappa) \Leftrightarrow \Psi_h(W_h) - \kappa = 0$$

We differentiate the scalar product j_h of the state variable by vector g:

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

The differentiation of AIRONUM has permitted to identify a particular problem related to allocation, which is addressed now.

3.2 A F90 difficulty: calling context and allocations

Tapenade differentiates the call tree that is under the root procedure that you have specified. The result is a set of differentiated modules and/or differentiated standalone procedures, that do not execute alone but must be used/called from a certain context. It is up to you to write this calling context, and Tapenade can help you only to a certain point. The following discussion focuses on the tangent mode. Be aware that things can be slightly harder in the adjoint mode.

You may use as a basis a calling context designed for the original, non-differentiated root procedure. Notice that even if you provided Tapenade with the complete calling context code as well, it will differentiate only the call graph below the root procedure. The calling context will not be modified nor adapted to the differentiated root. **So even if there is a calling context available for the root, you will have to modify it by hand so that it can call the differentiated root**.

There are many possible cases, so we'd rather use an example. We'll try to extract general rules later. Here's a small example source program. The differentiation root procedure is ROOT. The first column contains the code that is not in the call graph below ROOT.

```
PROGRAM MAIN
  USE MMA
  USE MMB
  REAL, DIMENSION(:), ALLOCATABLE :: x
  INTEGER :: i
  REAL :: cva(2)
  COMMON /cc1/ cva
  EXTERNAL GETFLAG
  CALL GETFLAG(flag)
  ALLOCATE(x(30))
  CALL INITA()
  DO i=1,20
    aa(i) = 100.0/i
    x(i) = 3.3*i
  END DO
  cva(1) = 1.8
  cva(2) = 8.1
  CALL ROOT(x)
  PRINT*, 'zz=', zz
  DEALLOCATE(aa)
  DEALLOCATE(x)
END PROGRAM MAIN
```

```
MODULE MMA
  REAL, DIMENSION(20), ALLOCATABLE :: aa

CONTAINS
```

```
    SUBROUTINE INITA()
      ALLOCATE(aa)
    END SUBROUTINE INITA
  END MODULE MMA

  MODULE MMB
    INTEGER :: flag
    REAL :: zz
  END MODULE MMB

  SUBROUTINE ROOT(x)
    USE MMA
    USE MMB
    REAL, DIMENSION(*) :: x
    REAL :: v
    REAL :: cv1, cv2
    COMMON /cc1/ cv1, cv2
    zz = cv1*cv2
    v = SUM(aa)
    IF (flag .GT. 2) zz = zz*v
    zz = zz + SUM(x)
  END SUBROUTINE ROOT
```

We will differentiate in tangent mode, with head ROOT. We will not specify the name of the output file, so that every module or isolated procedure of the differentiated code will be put into a separate file. The differentiated call graph is returned in files

```
  root_d.f90
```

,

```
  mma_d.f90
```

, and

```
  mmb_d.f90
```

.

```
  root_d.f90 defines procedure ROOT_D
```

```
!  Diff. of root in tangent mode:
!   variations of useful results: zz
!   with respect to varying inputs:
!    *aa cv1 cv2 x
!   RW status of diff variables:
!    zz:out *aa:in cv1:in cv2:in x:in
SUBROUTINE ROOT_D(x, xd)
  USE MMB_D
  USE MMA_D
  IMPLICIT NONE
  REAL, DIMENSION(*) :: x
  REAL, DIMENSION(*) :: xd
  REAL :: v
  REAL :: vd
  REAL :: cv1, cv2
  REAL :: cv1d, cv2d
  COMMON /cc1/ cv1, cv2
  INTRINSIC SUM
  COMMON /cc1_d/ cv1d, cv2d
  zzd = cv1d*cv2 + cv1*cv2d
  zz = cv1*cv2
  vd = SUM(aad)
  v = SUM(aa)
  IF (flag .GT. 2) THEN
    zzd = zzd*v + zz*vd
    zz = zz*v
  END IF
  zzd = zzd + SUM(xd)
  zz = zz + SUM(x)
END SUBROUTINE ROOT_D
```

```
mma_d.f90, mmb_d.f90 define modules MMA_D, MMB_D
```

```
MODULE MMA_D
  IMPLICIT NONE
```

```
    REAL, DIMENSION(:), ALLOCATABLE :: aa
    REAL, DIMENSION(:), ALLOCATABLE :: aad

  CONTAINS
    SUBROUTINE INITA()
      IMPLICIT NONE
      ALLOCATE(aad(20))
      ALLOCATE(aa(20))
    END SUBROUTINE INITA
  END MODULE MMA_D

  MODULE MMB_D
    IMPLICIT NONE
    INTEGER :: flag
    REAL :: zz
    REAL :: zzd
  END MODULE MMB_D
```

The calling context of the original ROOT is not sufficient to call

```
ROOT_D
```

and must be adapted. Here this context consists of program MAIN, but there can be more modules and procedures in it. The new program

```
MAIN_CD
```

produced by Tapenade in file

```
main_cd.f90
```

is a possible starting point, but it must be modified by hand. Or you can write it from scratch. For instance:

```
  PROGRAM MAIN_CD
    USE MMB_D
    USE MMA_D
    REAL, DIMENSION(:), ALLOCATABLE :: x
    REAL, DIMENSION(:), ALLOCATABLE :: xd
    INTEGER :: i
    REAL :: cvad(2)
    COMMON /cc1_d/ cvad
    REAL :: cva(2)
    COMMON /cc1/ cva
    EXTERNAL GETFLAG
    CALL GETFLAG(flag)
    ALLOCATE(xd(30))
    ALLOCATE(x(30))
    CALL INITA()
    initialize aad,xd,cvad,zzd
    DO i=1,20
      aa(i) = 100.0/i
      x(i) = 3.3*i
    END DO
    cva(1) = 1.8
    cva(2) = 8.1
    CALL ROOT_D(x, xd)
    PRINT*, 'zz=', zz, 'and zzd=', zzd
    DEALLOCATE(aad)
    DEALLOCATE(aa)
    DEALLOCATE(xd)
    DEALLOCATE(x)
  END PROGRAM MAIN_CD
```

Some comments:
* the context must call the differentiated root, hence the call to

```
ROOT_D
```

. * the context must provide

```
ROOT_D
```

with all the derivatives of its arguments, whether formal parameters or globals. Hence

```
xd
```

, but also

```
aad, /cc1_d/, zzd
```

. * These new

```
xd, aad, /cc1_d/, zzd
```

must be declared, allocated if their original variable is, and initialized with the derivatives you want. * After return from

```
ROOT_D
```

, the derivatives in these variables can be used, and the variables must be deallocated if their original variable is. * One must make sure that these variables are effectively the same in the context and in

```
ROOT_D
```

, which means that when they are globals from modules, the same modules must be used on both sides. Hence the

```
USE MMA_D
```

and

```
USE MMB_D
```

. This also ensures that the ”good”

```
INIT_A()
```

is called. * All these operations need not be done only in

```
MAIN_CD
```

. If

```
MAIN_CD
```

calls utility procedures, you may choose to modify these procedures instead. Make sure to keep all these modified procedures separated from their original definition, and link with the original code only when it was not modified. This also applies to

```
MAIN_CD
```

itself. * To make sure you don’t mix the original context procedures and the new context procedures adapted to

```
ROOT_D
```

, why not give them the same procedure name? This way, the compiler should complain if you mix them. Here

```
MAIN_CD
```

could just remain MAIN. The same could be said for module names

```
MMA_D
```

and

```
MMB_D
```

but that’s another story as currently Tapenade systematically appends a

```
_D
```

to them.

3.3 Differentiated code validation

The software AIRONUM evaluating $j_h(\kappa)$ implements an unsteady, turbulent Navier-Stokes simulation. The code is more that 100,000 lines long, and SPMD parallelization is necessary for most of its appli- cations. Message-passing is done with MPI calls.

In the software AIRONUM, communications does not use only point-to-point non- blocking communication MPI ISEND, MPI IRECV, and MPI WAIT, but also many collective communication MPI BCAST, MPI GATHER, and MPI ALLREDUCE. Given the current stage of development in Tapenade about message-passing communication, we first apply tangent differentiation on the code. The re- sulting derivatives were validated by comparison of the parallel tangent code with divided differences between two runs of the original code, each of them parallel. At the source level, 10 of the 32 calls to MPI were detected active, causing 10 differentiated message-passing calls.

The propose validation is performed by computing the derivative with respect to κ on a particular context, the turbulent flow around a cylinder, ECINADS Test Case 1.

On this relatively small test case, average run time per time step and per processor of the tangent code was 0.49 second, compared to an original run time per processor of 0.38 second. This increase of 30% is in line with what we observe on sequential codes.

The accuracy assessment is performed as follows: From one side, we apply the differentiated code and obtain a unique derivative $j_h'(\kappa)$ of j_h with respect to κ which we suppose to be a vector with a constant value, so that we can consider κ as a scalar. From the other side we can use the orignal code for an evaluation of this derivative by finite differences, *i.e.* by the application of small perturbations of κ of different sizes, and making the difference with the computation of j_h without perturbation.

We display in Fig.1 the variation of the difference:

$$\delta(\varepsilon) = |\frac{j_h(\kappa+\varepsilon) - j_h(\kappa)}{\varepsilon} - j_h'(\kappa)|.$$

We observe the classical V-curve, expressing that for decreasing ε, the finite difference deviation to exact derivative approaches zero, for a not too small ε. When the ε is too small, the finite difference is polluted by roundoff errors and its accuracy degrades again.

Here, we admit that the best finite difference is less accurate than the symbolic calculation, which is extremely probable, according to the curve.

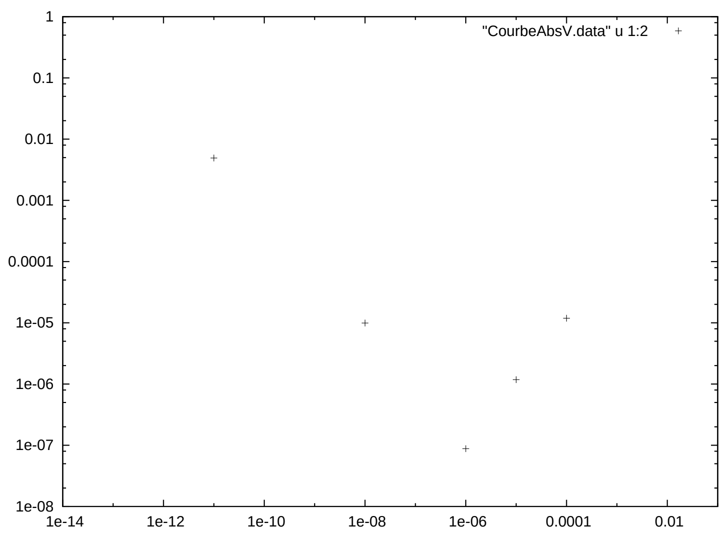

Fig. 1 The V-curve for validation of diferentiation for AIRONUM

4 Conclusion

After the allocation problems have been solved as explained here, TAPENADE could be applied to differentiate on direct mode a complete parallel CFD code. The next step is the application of reverse mode.

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.