OPT-*i*
An International Conference on
Engineering and Applied Sciences Optimization
M. Papadrakakis, M.G. Karlaftis, N.D. Lagaros (eds.)
Kos Island, Greece, 4-6, June 2014

DEVELOPMENT OF A DISCRETE ADJOINT CFD CODE USING ALGORITHMIC DIFFERENTIATION BY OPERATOR OVERLOADING

Z. Dastouri[1], J. Lotz[2], and U. Naumann [2]

[1] RWTH Aachen University, Software and Tools for Computational Engineering
D-52062 Aachen, Germany
e-mail: dastouri@stce.rwth-aachen.de

[2] RWTH Aachen University, Software and Tools for Computational Engineering
D-52062 Aachen, Germany
e-mail: {lotz,naumann}@stce.rwth-aachen.de

Keywords: Algorithmic Differentiation, Operator Overloading, Discrete Adjoint, Computational Fluid Dynamics(CFD) Code, dco/fortran, Sensitivity Analysis

Abstract. *Design optimization for fluid flow recently has drawn a great deal of attention in the industrial design area including aeronautic[1, 2], turbo-machinery[3] and automotive design[4, 5]. The optimization problem is subject to a large number of design variables that makes calculating the sensitivity derivatives by finite differences a costly procedure. Algorithmic differentiation(AD) is a well known method to differentiate the computer program with the aim of obtaining the sensitivity of the objective with respect to design variables. There are two different AD approaches, operator overloading and source transformation. In this paper we describe a design framework for application of the algorithmic differentiation tool by operator overloading in Fortran dco/fortran[1] to CFD analysis solver called GPDE [2] . GPDE is an unstructured pressure-based steady Navier-Stokes code with finite volume spatial discretization, which is based on the SIMPLE pressure-correction scheme for the incompressible viscous flow computation. This approach yields a discrete tangent-linear and adjoint version of the CFD code. Moreover, we address typical implementation issues and complexities in the differentiation procedure by AD tool in the CFD code. The numerical results of a relevant test case by our overloading tool in forward and reverse mode are compared with their corresponding results of the previously differentiated code by source transformation that is generated by TAPENADE. We show that in terms of ease of implementation and ability to handle arbitrary functions, dco/fortran provides the differentiated code with a greater flexibility and robustness in comparison with AD by source transformation. This research is aimed toward the application of AD tools by operator overloading on a legacy industrial incompressible flow solver that is in our context ESI's ACE+.*

[1] Derivative Code by Overloading in Fortran
[2] General Partial Differential Equation solver developed by CFD group of QMUL(http://www.qmul.ac.uk)

1 INTRODUCTION

In this paper, we discuss the application of automatic differentiating tool by operator overloading on a CFD code called GPDE[6] that is developed in QMUL. GPDE is an unstructured pressure-based steady Navier-Stokes solver with finite volume spatial discretization, for the incompressible viscous flow computation. A design framework to get adjoint sensitivity in a CFD solver is established emphasizing the simplicity and efficiency of the overloading tool. This tool called `dco/fortran` is a development at the institute *Software and Tools for Computational Engineering*[3] at RWTH Aachen University implementing Algorithmic Differentiation (AD) by overloading in Fortran [7].
In section 2 and 3, we give brief introductions to CFD background, the used simulation program, and basics of algorithmic differentiation. In section 4, we present the implementation of `dco/fortran` on original CFD solver from scratch to get the tangent-linear and adjoint version of the CFD code. In section 5 we address the implementation issues and typical complexities with respect to memory consumption and computational time. Moreover, we discussed and introduced proper solution algorithms including a checkpointing scheme for iterative solver and development of a continuous approach for linear solver inside the code. In section 6, we present the numerical results and sensitivity derivatives that is obtained for a test case . These results are compared with the differentiated code by source transformation that is generated by TAPENADE[8, 9]. Finally in section 7, we point out further possible improvements of the efficiency for calculation of the adjoint and also we present the future work.

2 CFD BACKGROUND

In general, a CFD analysis process provides a qualitative (or quantitative) prediction of fluid flows by means of mathematical modeling (partial differential equations), numerical methods (discretization and solution techniques), and software tools (solvers, pre and post processing utilities). For solving the steady incompressible flow, the governing equations (Navier-Stokes' equations) can be expressed as:

$$\int_{S_{cv}} (\rho\phi_i)\mathbf{v}\cdot\mathbf{n}d\mathbf{S} = \int_{S_{cv}} \nabla(\rho\phi_i)\cdot\mathbf{n}d\mathbf{S} + \int_{S_{cv}} (-\nabla\mathbf{p})\cdot\mathbf{n}d\mathbf{S} \tag{1}$$

$$\int_{S_{cv}} \nabla\cdot\mathbf{v} = 0 \tag{2}$$

where Equation (1) is the momentum equation, and ρ is the density of fluid; S_{cv} is the control volume surface with $d\mathbf{S}$ as surface element and $\mathbf{n}$ as normal surface vector; $\mathbf{v}$ stands for fluid velocity and ϕ is the velocity component in different direction; Equation (2) is the continuity equation [10]. In three dimensions, discretization of momentum and pressure equations leads to four equations with four unknowns. Pressure as a part of the source term is shown in the momentum equation, however, the equations do not provide an independent equation for the rate of change of pressure. Therefore, instead of using actual pressure value, Semi-Implicit Method for Pressure-Linked Equations or SIMPLE algorithm developed by Patankar and Spalding in 1972 [11] is applied to solve coupling problem of the velocity and the pressure via applying pressure and velocity correction equations.
In design optimization process, the goal is to minimize or maximize objective function J with

[3] https://www.stce.rwth-aachen.de

respect to a set of design parameters α. For a shape optimization problem, the design parameters control the geometry variables $\mathbf{x}$. Given a set of discrete flow solution F over a computational grid in a flow model, the objective function can be expressed as $J(F(\alpha), \mathbf{x}(\alpha))$. In CFD application the flow solution is defined implicitly through the solution of a set of nonlinear discrete flow equation of the form $R(F, \alpha) \equiv R(F, \alpha) = 0$ that is derived from Navier Stokes equations for a fixed-point iteration to a steady case.

2.1 Description of CFD Code

GPDE is an unstructured pressure-based steady Navier-Stokes solver with finite volume spatial discretization, which is based on the SIMPLE pressure correction scheme for the incompressible viscous flow computation. It is written in Fortran 90-95 (5,000 lines) as a test-bed for developing adjoint Navier-Stokes fields. The major com- ponents of the code includes:

- The *Mesh* module in GPDE can read mesh files generated by GMSH[4] and build the data structure for further flow calculation;
- The *Base* module provides sorting and listing functions as basic parts for efficient search;
- The *Geometry* module computes geometric quantities like face areas, volumes, normal vectors etc. across the grid;
- The *Pdes* module undertakes the main component in GPDE for fluid flow computation via generating the coefficient matrices and right hand sides of momentum equation, continuity equation and arbitrary scalar equation;
- The *Utils* module contains all the basic mathematical functions (vectors cross product, dot product and so on), fix-point control for the outer iteration, linear solvers for algebra equations and other miscellaneous subroutines.

In CFD calculation, using Equation (1) and Equation (2), the pseudo-time stepping is applied to achieve the steady flow field. Each pseudo-time step is one outer iteration for the computation of velocity and pressure field and the outer iteration is different from the inner iteration of iterative methods for solving momentum and pressure equations. The algorithm in the code computes laminar/turbulent flow through two and three dimension geometries. It needs to indicate that in GPDE the density of fluid is constant and equals to $1.0kg/m^3$. All the variables adopt SI (standard international) units. For numerical simulation, the similarity number (e.g. Reynold number or Mach number) should satisfy certain values according to the physical problems. GPDE is more suitable for flow within Mach number smaller than 0.3, where the compressibility of fluid can be neglected (More detail of code background is given in [12]).

3 ALGORITHMIC DIFFERENTIATION

For a given implementation of the flow solution $F : \mathbb{R}^n \to \mathbb{R}^m$ over a computational grid in a flow model as a computer program, AD enables us to compute derivatives of arbitrary order with an accuracy up to machine precision. Derivatives are meant as the sensitivities $\frac{\partial y}{\partial x}$ of the output $\mathbf{y} \in \mathbb{R}^m$ m with respect to the inputs $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} = F(\mathbf{x})$, yielding the Jacobian $\nabla F(\mathbf{x})$.

[4]http://geuz.org/gmsh/

The first-order tangent-linear routine $(\mathbf{y}^{(1)}, \mathbf{y}) = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)})$ computes the directional derivative $\mathbf{y}^{(1)}$ of F in direction $\mathbf{x}^{(1)}$ in addition to the function value:

$$\begin{aligned} \mathbf{y}^{(1)} &= \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \\ \mathbf{y} &= F(\mathbf{x}) \quad , \end{aligned} \tag{3}$$

where $\nabla F(\mathbf{x}) \equiv \nabla_{\mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{m \times n}$ denotes the Jacobian (matrix) of F.

The function $F^{(1)} : \mathbb{R}^{2.n} \rightarrow \mathbb{R}^m$, defined as

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv < \nabla F, \mathbf{x}^{(1)} > \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)}, \tag{4}$$

is referred to as the *tangent-linear* model of F. This produces the matrix-vector product of the derivative with a direction $\mathbf{x}^{(1)}$. Dense Jacobians can be obtained in tangent-linear mode at $O(n)Cost(F)$, where $Cost(F)$ denotes the computational cost of a single evaluation of F .

The first-order adjoint routine $(\mathbf{x}_{(1)}, \mathbf{y}) = F_{(1)}(\mathbf{x}, \mathbf{x}_{(1)}, \mathbf{y}_{(1)})$ increments the input value of $\mathbf{x}_{(1)}$ with the adjoint of F in direction $\mathbf{y}_{(1)}$ in addition to the computation of the function value:

$$\begin{aligned} \mathbf{x}_{(1)} &= \mathbf{x}_{(1)} + (\nabla F(\mathbf{x}))^T \cdot \mathbf{y}_{(1)} \\ \mathbf{y} &= F(\mathbf{x}) \end{aligned} \tag{5}$$

The function $F_{(1)} : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^n$, defined as

$$\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv < \mathbf{y}_{(1)}, \nabla F > \equiv \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}, \tag{6}$$

is referred to as the *adjoint model* of F. Therefore, the transposed Jacobian $\nabla F^T = \nabla F(\mathbf{x})^T$ is obtained in adjoint mode at $O(m)Cost(F)$. For the purpose of our CFD code optimization, we applied both model, however the adjoint model is preferred since a large number of inputs (n) are mapped to a rather small amount of outputs (m).

There are two main methods for implementing algorithmic differentiation: by source code transformation or by using derived data types and operator overloading. To implement algorithmic differentiation by source transformation, the code is parsed at a compile time and a new code is generated that computes its tangent or adjoint derivatives. While in another approach, the existing operators are re-defined (overloaded) to additionally calculate the derivative portion using the definition of the derivative of the operator. For more detailed information about Algorithmic Differentiation refer to the text books [7, 13].

4 DEVELOPMENT OF INCOMPRESSIBLE ADJOINT SOLVER USING `dco/fortran`

The source transformation AD tool that is previously applied on GPDE CFD code for differentiating purposes is Tapenade[9]. Here, the operator overloading AD approach is applied to obtain Tangent and Adjoint codes using `dco/fortran`. The objective function in GPDE flow solver can be defined as pressure loss, wall forces, etc to get the *Surface mesh sensitivity* or *Flow variable sensitivity* with respect to corresponding independent design variables such as surface node coordinate, flow velocity or pressure. Reading the mesh information from the mesh files and the flow equations including continuity, momentum, turbulence and other flow solvers are called via a main file. In main_gpde, the boundary condition, initialization of flow

variables and the input variables for number of iteration, density, convection and diffusion coefficients and other related data are determined. The detail of implementation for generating the tangent and adjoint code using `dco/fortran` is given below.

Tangent-Linear Code During the execution of the differentiated routine by the overloaded operators and intrinsic functions from the module `DCO_T1S`, tangents of all activated variables and compiler generated active temporaries are propagated together with the function value. A tangent-linear routine of a given implementation of CFD Fortran program F is geven below:

```
subroutine gpde_main

!1-USE-ing the module DCO_T1S
use dco_t1s

!2-Activation: Re-declaring all floating point variables
!  relevant to the calculation of the cost function with
!  the TYPE(DCO_T1S_TYPE)
type(dco_t1s_type),dimension(n_dx,n_vrt) :: geom
type(dco_t1s_type),dimension(n_dx,n_vrt) :: vel,pres,init_vel
type(dco_t1s_type) :: obj
double precision,dimension (n_vrt) :: Jac

!3-Reading Mesh data, Boundary Condition and Initialization
call read mesh(msh,geom%x)
call connectivity (msh)
...
init_vel =1*(-1)*geom%norm
...
!4-Seeding: SET-ing the derivatives x^1 of input design
!   variable x as 1.0 before calling the tangent-linear routine
call dco_t1s_set(geom%dx,0.d0,1)
do i =1, n_vrt
  call dco_t1s_set(geom%dx(1,i),1.d0,1)

!5-Calling the driver code consist of iterative loop for
!  momentum, continuity, scaler and flow equations.
  call state objective(msh,vel,geom,pres,obj,..)
  {do fp-iter=1,vel%n_iter
    call momentum equation ( )
    call continuity equation ()
    call scalar equation ()
    ...
   enddo}.
!6-Harvest objective tangent
  call dco_t1s_get(obj,Jac(i),1)
!7-Reset tangent of input to initial
  call dco_t1s_set(geom%dx(1,i),0.d0,1)
!8-Reseting initial value and tangents for new solution
```

```
  pres%phi=0
  vel%phi=init_vel
  spal%phi=init_spal
end do
print *,"FWD sensitivity:", Jac
end subroutine
```

Adjoint Code In reverse mode, *activation* is slightly same procedure as tangent-linear code by re-declaring all relevant variables with the `DCO_A1S_TYPE` by `USE`-ing the module `DCO_A1S`. But in contrast to the simultaneous computation of tangents and values of the tangent-linear version of F, an adjoint computation by `DCO_A1S` consists of setting the value component of x the input data and registering x in the tape in *forward sweep* and computing the product of the transposed Jacobian ∇F^T of F with the adjoint $y_{(1)}$ of output y in reverse sweep. The steps of implementation of the adjoint code for a CFD Fortran program are explained below:

```
subroutine gpde_main
!****Forward sweep****
!1-USE-ing the module DCO_A1S
use dco_a1s

!2-Activation: Re-declaring all floating point variables
!  relevant to the calculation of the cost function with
!  the TYPE(DCO_A1S_TYPE)
type(dco_a1s_type),dimension(n_dx,n_vrt) :: geom
type(dco_a1s_type),dimension(n_dx,n_vrt) :: vel,pres,init_vel
type(dco_a1s_type) :: obj
double precision,dimension (n_vrt) :: Jac

!3-Reading Mesh data, Boundary Condition and Initialization
call read mesh(msh,geom%x)
call connectivity (msh)
...
init_vel =1*(-1)*geom%norm
...
!4-Initialize tape, activating the tape recording
call dco_a1s_tape_create ()
!5-Registering the design variable x on the tape
call dco_a1s_tape_register_variable(geom%dx)
!6-Calling the driver code consist of iterative loop for
!  momentum, continuity, scaler and flow equations.
  call state objective(msh,vel,geom,pres,obj,..)
  {do fp-iter=1,vel%n_iter
    call momentum equation ( )
    call continuity equation ()
    call scalar equation ()
    ...
   enddo}.
!****Reverse sweep****
```

```
!7-Set-ing the objective adjoint to 1.D0.
call dco_a1s_get( obj,1.d0,   1 )
!8-Propagate adjoints in tape
call dco_a1s_tape_vinterpret_adjoint
!9-Harvest surface sensitivity derivatives
call dco_a1s_get( geom%dx ,Jac ,   1 )
!10-Release memory allocated by tape machinary
call dco_a1s_tape_remove
print *,"REV sensitivity:", Jac
end subroutine
```

5 IMPROVING DIFFERENTIATED CODE PERFORMANCE

In structure of a CFD code, there are outer iterative loop for updating velocity and pressure field or inner loop for linear solver of flow equations. These iterative loop may not lend themselves to Automatic Differentiation efficiently. In this section, we address the implementation issues and proper solution algorithm regarding to convergence and memory consumption of the differentiated code.

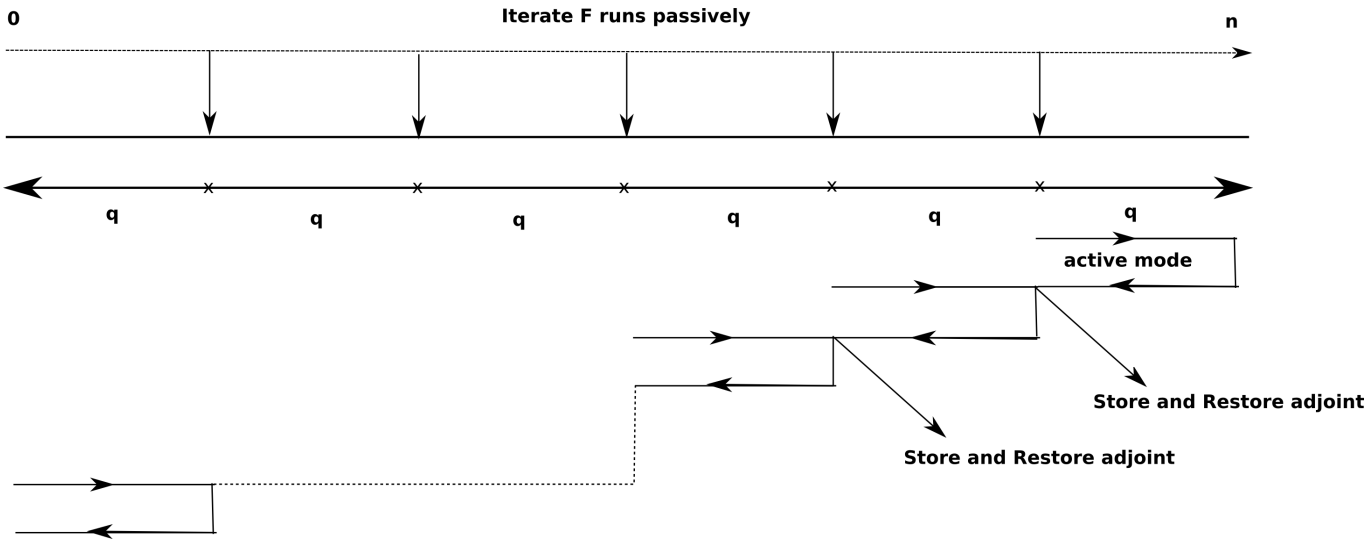

Figure 1: Visualization of equidistant checkpointing scheme

5.1 Iterative Solver; Checkpointing Scheme

Simplicity of implementation of the adjoint method comes with a price. In a CFD solver, all inner iterations for linear solver and outer iterations for calculating the velocity and pressure have to be taped. The needed size of this tape grows dynamically proportional to the number of operations in the calculation that leads to considerable memory consumption. The solution to this problem is to employ a checkpointing scheme. We use an equidistant checkpointing scheme as visualized in Figure 5.

Rather than storing and taping every variable and operation during the forward run, checkpoints are stored at strategically chosen intervals q for n_chk (number of checkpoint steps) where $n\_chk = n\_iter/q$ and n_iter is the total number of iterations. The implementation steps for

applying equaldistance checkpointing in adjoint code is given below:

Forward Sweep

1. Reading Mesh data, Boundary Condition and Initialization
2. Determining q as a checkpoint step
3. Calling the driver code passively consist of iterative loop for momentum, continuity, scalar and flow equations and Recording flow variables (velocity, pressure and dependent variables) in each q step.
   `Do` $n = 1 : n\_chk :$
4. Restoring checkpointed variables from n_iter to $n * q$ iterations.
5. Initializating the tape, activating the tape recording for the last interval $(n\_iter-(n * q))$ to $(n\_iter - (n - 1) * q)$
6. Registering the flow variables (velocity, pressure and dependent variables) to tape
7. Calling the driver loop in $(n\_iter-(n * q))$ to $(n\_iter-(n-1)*q)$ interval for registering the operation.

Reverse Sweep

1. Seeding: `SET`-ing the objective adjoint of y to $y_{(1)}$,
2. Tape interpretation: Propagate the adjoints $y_{(1)}$ of output y to adjoints of input reversely
3. Harvesting: `GET`-ing the adjoint of the input variable $\mathbf{x}$ in $n - 1$ interval
4. Recording the adjoint of inputs for the last interval as $y_{(1)}$
5. Releasing memory allocated by tape machinery and Resetting the tape

   `End do`

5.2 Linear solver; Continuous Approach

In the SIMPLE algorithm, the momentum equations and the continuity equations are solved by two types of linear solvers. Because of the non-symmetric coefficient matrix A, the momentum equation is solved via Biconjugate gradient method (BiCG-solver in GPDE)[10]. On the other hand, the continuity equation (pressure-correction equation) is symmetric and is solved by conjugate gradient method (CG-solver in GPDE)[10].
Since AD uses the same control data flow as the original program, when the linear solver is iterative, the number of iterations required to fully converge the solution x of $A\mathbf{x} = \mathbf{b}$ might not be enough to converge its derivative $\mathbf{x}^{(1)}$ or $\mathbf{x}_{(1)}$. Moreover, in discrete adjoint of a linear solver, every basic mathematical calculation has to be adjoint. These intermediate values do not need to be calculated when using a continuous adjoint version. This continuous version will be inserted in the discrete adjoint of the rest of the program.

Tangent-Linear Code According to [16] the tangent-linear projection $s^{(1)}$ of the solution $s = L(A, \mathbf{b})$ in directions $A^{(1)}$ and $\mathbf{b}^{(1)}$ is implicitly given as the solution of the linear system

$$A \cdot s^{(1)} = \mathbf{b}^{(1)} - A^{(1)} \cdot s \tag{7}$$

The treatment for linear solver $A\mathbf{x} = \mathbf{b}$ for tangent linear code is given blow:

1. `GET`-ing the tangent of A and $\mathbf{b}$ before iterative loop as $A^{(1)}$ and $\mathbf{b}^{(1)}$
2. `SET`-ing tangent of s to zero
3. `CALL`-ing linear solver $A\mathbf{x} = \mathbf{b}$ to get s
4. Building new right hand side as $rhs = \mathbf{b}^{(1)} - A^{(1)} \cdot s$
5. `CALL`-ing linear solver $A \cdot s^{(1)} = rhs$ to get $s^{(1)}$
6. `SET`-ing the tangent of s as $s^{(1)}$

Adjoint Code According to [16] the the adjoint projections $A_{(1)}$ and $\mathbf{b}_{(1)}$ for given adjoints $s_{(1)}$ can be computed as

$$A^T \cdot \mathbf{b}_{(1)} = s_{(1)} \tag{8}$$

$$A_{(1)} = -\mathbf{b}_{(1)} \cdot s^T \tag{9}$$

`dco/fortran` provides an interface to use external functions for the calculation of the adjoint. Instead of recording the operations of such a function inside the forward-run, this function is registered in the tape to call and execute in reverse run of the adjoint code. The treatment for linear solver for adjoint linear mode is:

1. Forward sweep
   (a) `USE`-ing the module `DCO_A1S`.
   (b) Creating the checkpoint interval (cp)
   (c) Writing A and $\mathbf{b}$ values to the checkpoint
   (d) Calling the inner linear solver passively $A.s = \mathbf{b}$
   (e) Registering output s to tape
   (f) Registering the `External function` to tape to point out the function which should be called in the reverse sweep
2. Reverse sweep
   (a) Reading from the checkpoint A and $\mathbf{b}$ values
   (b) Reading the incoming adjoint $s_{(1)}$ from the tape
   (c) Transposing matrix A
   (d) Calculating $A^T.\mathbf{b}_{(1)} = s_{(1)}$
   (e) Returning outgoing adjoint $\mathbf{b}_{(1)}$ to tape
   (f) Calculating $A_{(1)} = -\mathbf{b}_{(1)}.s^T$
   (g) Returning outgoing adjoint $A_{(1)}$ to tape

6 NUMERICAL RESULTS

In this section, numerical results from a relevant test case are presented for flow simulation model, tangent linear and adjoint code and the comparison is made with sensitivity results of adjoint code that is generated by Tapenade.

6.1 Case Study-Simulation of S-bend channel flow (3D)

The case study that is used for flow model simulation and sensitivity studies, is the S-bend channel flow case from Volkswagen which is a simplified vehicle climatisation duct[17]. Test case is carried out at *Reynold = 500* on hexahedral mesh with 41044 elements. The boundary condition is defined as uniform flow at inlet. The simulation in GPDE is performed for 112 iteration in the primal code until we attain the convergence with the tolerance of 0.0001 for velocity reduction. The velocity vector field along the channel is shown in Figure 2. Because cross section of the S-bend channel, the bend area is not axial symmetric and therefore, the outlet velocity field expresses the asymmetric distribution.

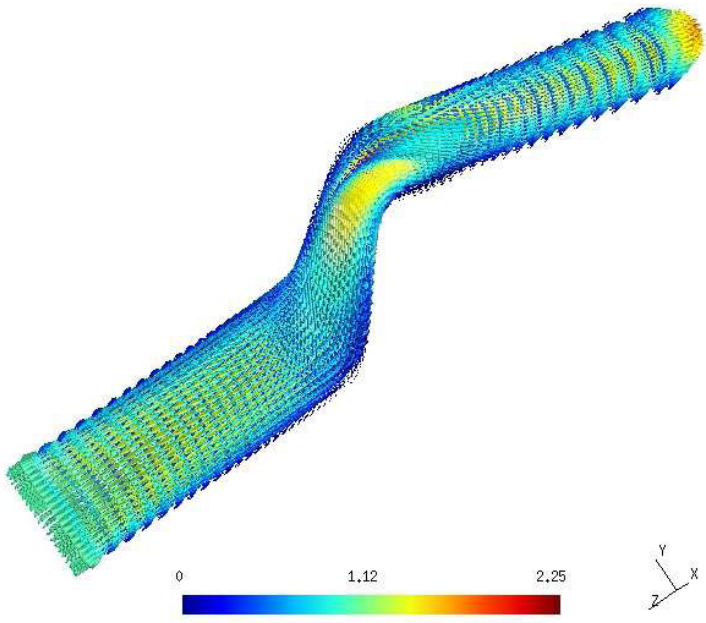

Figure 2: **Velocity vector field though the geometry center of the channel;** The velocity shows nearly parabolic distribution between inlet and bend areas. Due to the bend of channel, the velocity increase through the narrow part of channel, and small vortex appears after the bend area because of the flow separation.

6.2 Sensitivity Analysis Results

Sensitivity analysis results are presented for pressure loss objective function with respect to surface boundary nodes of S bend in three dimensions. The first order sensitivity derivatives show the direction of surface change that leads to shape modification in the bend. These results are illustrated in Figure 3. The numerical sensitivity results using *dco/fortran* are compared with results of Tapenade tool along desired lines of geometry that determined in Figure 4. The results of comparison presented in Figures 5 and 6 and Table 1 for surface sensitivity.

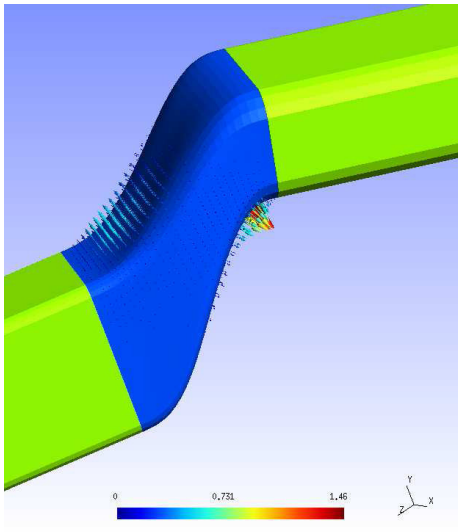

Figure 3: **Sensitivity analysis results for pressure loss objective function with respect to surface boundary node;** The first order sensitivity derivatives show the direction of surface change that leads to shape modification in the bend, These results are obtained for 112 iteration.

It shows for overloading approach, results of forward and reverse adjoint sensitivity algorithm match perfectly up to machine accuracy, and also the pressure function values is the same with primal output. The surface sensitivity results are verified with finite difference method for optimum difference. The comparison in Figure 5 shows the same behavior qualitatively for three approaches. The difference in gradient values for source transformation and overloading approaches AD tools might be due to the different function values of pressure output in AD Tapenade code compare to the primal code as it explained in Table 1.

	Max Surface Sen.Top	Max Surface Sen.Bot	Pressure loss value
Primal	$---$	$---$	$6.47982977E+03$
Tapenade	0.0986663660	-0.2448022212	$6.48227704E+03$
Tangent linear *dco*	0.1047474744	-0.2026309214	$6.47982977E+03$
Adjoint *dco*	0.1047474744	-0.2026309214	$6.47982977E+03$
Finite Difference	0.1010346202	-0.1157092919	$6.47982977E+03$

Table 1: Gradient Comparison of Tapenade and *dco/fortran* AD tools

6.3 Overloading vs Tapenade

Both target of AD tool by operator overloading and source transformation approaches are implemented to generate sensitivity algorithm for CFD code. The relative advantages and disadvantages of using the two different tools are summarized in Table 2. The + sign indicates the AD tool has the desired characteristics in this respect while the indicates the undesirable one.

For complex CFD code and large-scale sensitivity analysis, using a hybrid (source transformation/overloading) discrete adjoint approach is intended. As explained before, due to the memory consumption of the taping process in adjoint code by overloading tool, an equal distance checkpointing scheme is implemented. It results in computational time as 120.212 seconds when the primal runs in 7.356 seconds for number of iteration 60. This time for finite

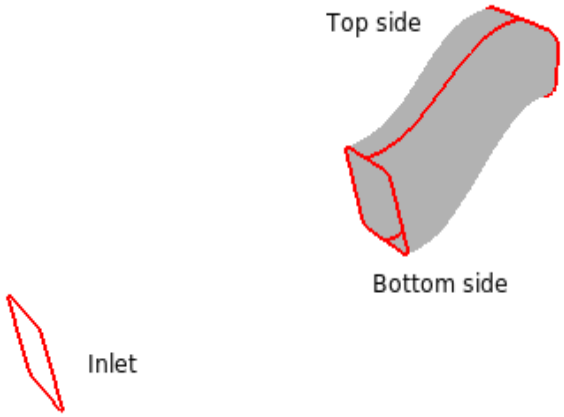

Figure 4: Reference of comparison

AD tool	O−O	S−T
Ease of implementation	+	−
Changing the original source code	+	−
Memory consumption	−	+
Compile time optimizations	−	+
Compatibility of numerical gradients with the discrete PDE	+	+
Flexibility to handle arbitrary functions	+	−
Robustness	+	−

Table 2: Comparison operator overloading and source transformation AD tools

difference approach with respect to the number of nodes, is $2\times$ number of nodes $\times$ computational time for running primal$\approx$426 seconds. The checkpointing scheme applied in adjoint code with Tapenade is binomial that is the optimal memory/runtime tradeoff for iterative loops developed by [18] and reduces the computational time to 40.334 seconds. The true comparison of different tools can be made when a similar checkpointing scheme is applied to both adjoint codes. The expected slowdown by using `dco/fortran` is 1.5-2 times when compared to the source transformation approach with a considerable reduction in development time.

7 CONCLUSIONS AND OUTLOOK

In this paper generation and performance of a discrete adjoint for an unstructured mesh flow solver using AD tool by operator overloading is described. The accuracy of the adjoint sensitivities in discrete forward and reverse version has been verified and compared with a previously validated sensitivity obtain from a source transformation approach for a S-bend test case. To improve the adjoint performance, an equaldistances checkpointing scheme is applied. Another area for improvement is to replace the used equidistant checkpointing scheme by revolve [18]. Numerical experiments indicate that the `dco/fortran` adjoint approach can be used to estimate the sensitivity, showing generally good agreement with Tapenade tool. In terms of robustness, ease of development and ability to handle arbitrary functionals, `dco/fortran` is more efficient and its application can be extended to more complex CFD code. However a

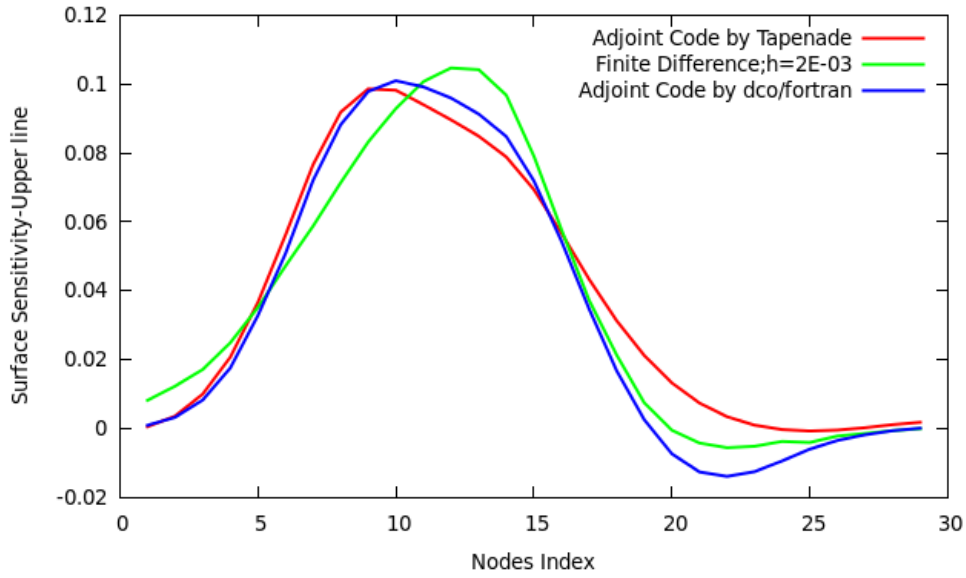

Figure 5: **Sensitivity analysis results for pressure loss objective function with respect to top side surface node**; (refer to Figure 4); The iteration converged after 112 iteration. The surface sensitivity results are verified with finite difference method. The comparison shows the same behavior qualitatively for three approaches.

hybrid operator-overloading/source-transformation approach will remain a worthwhile tool to reduce the memory requirements for the differentiating purposes. Our next step in this research is to extend the development and application of `dco/fortran` to generate the sensitivity algorithm for the industrial-commercial CFD solver, ACE+ developed by ESI group and relevant industrial test cases. In this context, pre-processing steps of differentiating the source code and compilation the overloading datatype and functions in ACE+ modules is performed and the framework will be presented in the next paper.

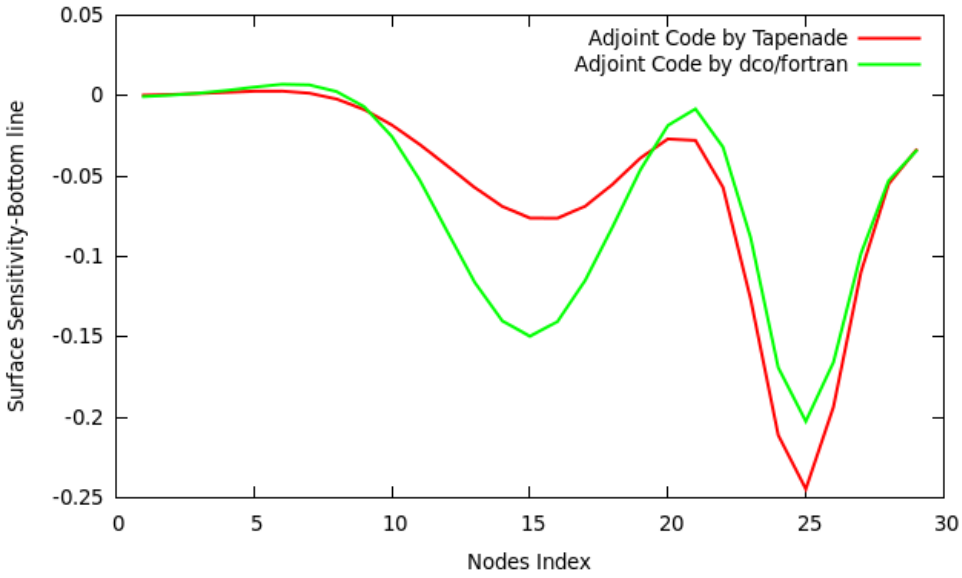

Figure 6: **Sensitivity analysis results for pressure loss objective function with respect to bottom side surface node**; (refer to Figure 4). The positive sensitivity indicates that the surface area should potentially be pulled out for optimization the objective function.

ACKNOWLEDGEMENT

The presented research is supported by the project **aboutFlow**, funded by the European Commission under FP7-PEOPLE-2012-ITN-317006.

REFERENCES

[1] A. Jameson, L. Martinelli, and N.A. Pierce. *Optimum aerodynamic design using the Navier-Stokes equations*. Theor. Comp. Fluid. Dyn., 10:213-237, 1998.

[2] M. B. Giles, M. C. Duta, J.-D Muller, and N. A. Pierce. *Algorithm developments for discrete adjoint methods*. AIAA Journal, 41(2):198-205, 2003.

[3] S. Kammerer, J.F., M. Paffrath, U. Wever, and A.R. Jung. *Three-dimensional optimization of turbomachinery bladings using sensitivity analysis*. AIAA paper, (GT2003-38037), 2003.

[4] C. Othmer and T. Grahs. *Approaches to fluid dynamic optimization in the car development process*. In R. Schilling et. al., editor, Eurogen, Munich, 2005. Eccomas.

[5] C. Othmer, T. Kaminski, and R. Giering. *Computation of topological sensitivities in fluid dynamics: cost function versatility*. In Eccomas CFD 2006. Eccomas, P. Wesseling and E. Onate and J. Periaux, 2006

[6] D. Jones, http://cfdp[ack.net, *dominic.jones@gmx.co.uk*, London, 2012.

[7] U. Naumann, *The Art of Differentiating Computer Programs*, SIAM, 2011.

[8] TAPENADE, On-line Automatic Differentiation Engine, *www-tapenade.inria.fr*. 2010.

[9] EU Research Projects, *Fluid Optimisation Workflows for Highly Effective Automotive Development Processes(FLOWHEAD)* Project reference: 218626, Funded under: FP7-TRANSPORT , July 2012

[10] J.H.Ferziger, M.Peric. *Computational Methods for Fluid Dynamics*. Springer, 2002.

[11] S.V. Patankar and D.B. Spalding. *A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows*. International Journal for Heat Mass Transfer, Number 15:1787-1806, 1972.

[12] Y. Wang. *Shape-Optimisation based on 3D Unstructured Grid with Adjoint Method*. Technical Report, School of Engineering and Materials Science Queen Mary, University of London, 2013.

[13] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[14] U. Naumann, AD-enabled NAG Fortran compiler, dco/fortran:User Guide, 2013

[15] U. Naumann, J. Lotz. *Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations*. Technical Report, AIB-2012-10, 2012

[16] M. B. Giles. *Collected matrix derivative results for forward and reverse mode algorithmic differentiation*. See Bischof et al. 2008, 3544

[17] C. Othmer. *A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows*. International Journal for Numerical Methods in Fluids, 58(8):861, 2008. DOI 10.1002/fld.1770.

[18] A. Griewank and A. Walther. *revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation*. Algorithm 799: ACM Transactions on Mathematical Software, 26(1):1945, March 2000.