V European Conference on Computational Fluid Dynamics
ECCOMAS CFD 2010
J. C. F. Pereira and A. Sequeira (Eds)
Lisbon, Portugal,14-17 June 2010

TIME-STEPPING FOR ADJOINT CFD CODES FROM AUTOMATIC DIFFERENTIATION

Faidon Christakopoulos*, Dominic Jones and Jens D. Müller

The School of Engineering and Materials Science,
Queen Mary,
University of London,
Mile End Road, London, E1 4NS, United Kingdom
e-mails: {*f.christakopoulos, dominic.jones, j.mueller}@qmul.ac.uk

Key words: Optimization, Discrete, Adjoint, Automatic Differentiation

Abstract. *In the case of aerodynamic shape optimization problems, the adjoint formulation has always been favored due to its properties. The past decade lots of effort has been drawn on the development of continuous adjoint solvers, which have though presented important disadvantages. The development and improvement of automatic differentiation tools in the last years on the other hand, has enabled the exploitation of the discrete adjoint formulations, which present some great advantages and seem to overcome the problems of the continuous approach. These formulations can even be coupled with acceleration methodologies such as multigrid and one-shot to produce even faster and more efficient adjoints and optimization codes.*

1 INTRODUCTION

The property of adjoint CFD codes to be practically independent of the number of design variables makes the implementation of such codes essential for efficient gradient-based optimization, in problems were more than very few design variables have to be considered, e.g. in real industrial 3D problems. Early implementations [1] have favored continuous adjoint codes, where the adjoint equations are derived, then discretised. Such codes though have been proven to be rather difficult to maintain and often present convergence problems due to errors in the transposition of the Jacobian matrix $\mathbf{A}$.

On the other hand, in discrete adjoint codes, the adjoint code is derived as a differentiation of the code statements of a CFD code. This presents a number of advantages. First of all, the adjoint Jacobian $\mathbf{A^T}$ is guaranteed to be the exact transpose of the primal Jacobian $\mathbf{A}$ and thus convergence loss problems are avoided. Most importantly, the discrete approach is straightforward, albeit tedious, but can be automated using automatic differentiation tools (AD).

A number of applications of AD to CFD codes have been published before [2, 3, 4] and have been shown for some types of discretisation to be able to obtain a performance comparable to continuous adjoint codes [5].

In this paper, emphasis is placed on presenting the way and the simplicity, in which an AD derived adjoint code can be validated and assembled from the original CFD code. A different approach to the adjoint pseudo-time-step is presented and then the whole primal and adjoint code is coupled with one-shot and multigrid methods, in order to acquire an even faster optimization process.

2 METHODOLOGY

2.1 Automatic Differentiation

Automatic differentiation (AD) is the process through which the code that calculates the gradient of a primal (code written) function is automatically generated. The main logic used by the various AD tools (e.g. *Tapenade, TAMC, etc*) is that the chain rule can be used to derive the gradient of a primal function, no matter how complicated this function may be. This gradient can be computed either in forward (tangent linearisation) or reverse (adjoint) mode.

Therefore, provided a primal CFD code, its equivalent tangent linearisation and adjoint codes can be straightforwardly derived using the AD tools and be coupled with the original code so that the gradients are calculated. The ability of generating both versions is of great importance, as it enables the validation of the adjoint in every iteration towards convergence, as it will be shown later on in the paper.

This provides a great advantage to the discrete adjoint optimization CFD codes, as the process of deriving the adjoint code can be fully automated and can also be easily checked, debugged and maintained, considering that the generation of the AD routines is practically costless in time for the CFD code programmer.

Furthermore, the use of AD guarantees the exact transposition of the Jacobian $\mathbf{A}$, thus convergence loss problems are avoided. These problems are very often in continuous adjoint codes, due to transposition errors, which require lots of effort to be traced and is highly time consuming.

2.2 Multigrid

Multigrid methods have proven to provide to major advantages, which are convergence frequency damping and acceleration. They follow four main steps, that can be summarized as :

1. Remove fine grid oscillations with the smoother.
2. Restrict fine grid solution and residuals to the coarse grid.
3. Remove coarse grid oscillations with the smoother.
4. Prolongate the coarse grid corrections to the fine grid.

The main multigrid cycles used, are the V and the W cycles, presented in the following figure :

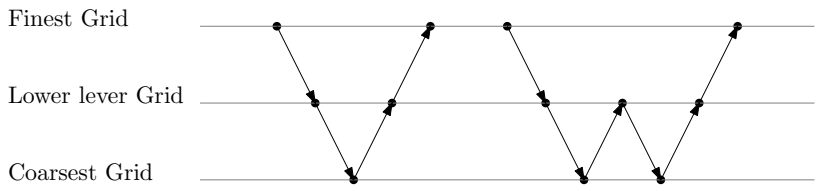

Figure 1: Two basic multigrid cycles, the V and the W cycle.

In this paper, the V cycle is used, but some extra attention should be drawn to the adjustment of the multigrid process, when the altered adjoint pseudo-time-step (3.2) is used.

2.3 One-shot

In one-shot methods [6], the solution to the KKT system is obtained by solving the primal, adjoint and design simultaneously. Such methods have been shown to be able to converge the primal, adjoint and design in 5-10 times the cost of computing the primal. On the other hand, the classic approach of converging primal and adjoint fully for each design step, typically results in factors of 50-100 times the primal for a converged design, which is not acceptable in large optimization problems, as the ones in an industrial design environment.

The simplest one-shot strategy is to use a fixed number of iterations on the primal and adjoint and a Wolfe-condition to select the step-size. There is a more effective way

though, that uses the linear relation that had been proven to exist between the convergence of primal and adjoint and the convergence of the design. Based on this relation, a convergence criterion can be used to stop iterating on primal and adjoint, which can be described by the equation :

$$g_{RMS} = \frac{|\nabla J|}{C} \tag{1}$$

where J the functional and C a user defined constant, that controls the accuracy of the primal and adjoint solution at each design step. This second methodogy is adopted in this paper.

3 THEORY

3.1 Adjoint pseudo-time-stepping validation theory

Every automatically generated subroutine in reverse (adjoint) mode can be validated by using the equivalent forward mode (tangent linearisation) subroutine. The results of these subroutines should be identical. For example, considering the subroutine `calclift` that calculates the lift coefficient of an airfoil, the equivalent automatically generated subroutines in forward and reverse mode are respectively :

$$\texttt{calclift}_\texttt{d}(\alpha, \rightarrow \alpha_\texttt{d}, \texttt{C}_\texttt{L}, \leftarrow \texttt{C}_{\texttt{L}_\texttt{d}}) \text{ and } \texttt{calclift}_\texttt{b}(\alpha, \rightarrow \alpha_\texttt{b}, \texttt{C}_\texttt{L}, \leftarrow \texttt{C}_{\texttt{L}_\texttt{b}})$$

where `d` and `b` denote forward and reverse differentiation respectively.

Setting $\alpha_\texttt{d} = 1$ and $\texttt{C}_{\texttt{L}_\texttt{b}} = 1$, the two subroutines compute the same derivative in two different ways :

$$\texttt{C}_{\texttt{L}_\texttt{d}} = \frac{\partial C_L}{\partial \alpha}$$
$$\alpha_\texttt{b} = \frac{\partial C_L}{\partial \alpha}^T$$

These derivatives should match exactly for the two subroutines. The value of these derivatives can be validated using a simple finite differences calculation.

In the same logic, but one step further ahead, the whole adjoint time-stepping loop can be validated versus the equivalent tangent linearisation loop. This can become clearer, if one considers a typical CFD shape optimization problem. The primal would be the solution of the system of state equations :

$$R(U, \alpha) = 0 \tag{2}$$

where U the state and α the design variables.

Considering a cost function J, its derivative with respect to the design variables would be :

$$\frac{dJ}{d\alpha} = \frac{\partial J}{\partial \alpha} + \frac{\partial J}{\partial U}\frac{\partial U}{\partial \alpha} \tag{3}$$

The term $\partial U/\partial\alpha$ is calculated by differentiating the the state equations (2) with respect to α :

$$\begin{aligned} & \frac{\partial}{\partial\alpha}\left(R\left(U,\alpha\right)\right)=0 \\ \Rightarrow\quad & \frac{\partial R}{\partial U}\frac{\partial U}{\partial\alpha}+\frac{\partial R}{\partial\alpha}=0 \\ \Rightarrow\quad & \mathbf{A}u=f \\ \Rightarrow\quad & u=\mathbf{A}^{-1}f \end{aligned} \tag{4}$$

where $\mathbf{A}$ the Jacobian, $u=\partial U/\partial\alpha$ the perturbation field and $f=-\partial R/\partial\alpha$ a source term, which is added to the conservation equations by the shape change. Through (4), (3) can be written :

$$\frac{dJ}{d\alpha}=\frac{\partial J}{\partial\alpha}+g^{T}u \tag{5}$$

where $g^{T}=\partial J/\partial U$. The variable u is the output of the tangent linearisation version of the primal pseudo-time-stepping loop.

On the other hand, the adjoint variables v are the solution of the system :

$$\begin{aligned} & \mathbf{A}^{T}v=g \\ \Rightarrow\quad & v=\mathbf{A}^{-T}g \\ \Rightarrow\quad & v^{T}=g^{T}\mathbf{A}^{-1} \end{aligned} \tag{6}$$

Through (4) and (6), (5) can be written as :

$$\frac{dJ}{d\alpha}=\frac{\partial J}{\partial\alpha}+v^{T}f \tag{7}$$

from which the *adjoint - tangent linearisation equivalence* is obvious :

$$g^{T}u=v^{T}f \tag{8}$$

Equation (8) must be satisfied at every iteration for the forward and reverse AD derived code.

Of course, from the automatic differentiation point of view, the transpose of $dJ/d\alpha$ is being calculated from (7) as :

$$\frac{dJ}{d\alpha}^{T}=\frac{\partial J}{\partial\alpha}^{T}+f^{T}v \tag{9}$$

but exactly the same relation must be satisfied. Here, it should be mentioned that the terms $(\partial J/\partial\alpha)^{T}$ and f^{T} can be calculated either in forward or in reverse mode.

3.2 Adjoint pseudo-time-stepping alteration

In one-shot adjoint optimization methods, the optimum sought is the solution of the system :

$$\mathbf{R}(Q, \alpha) = 0 \tag{10}$$

$$\mathbf{A}^T v = g \tag{11}$$

$$\frac{\partial J}{\partial \alpha} + v^T f = 0 \tag{12}$$

where the solution of (10) are the state variables (*primal*), of (11) the adjoint variables (*dual*) and of (12) the optimal shape.

The primal solution is acquired using a pseudo-time stepping scheme of the form :

```
do nIter = 1,mIt
  call residual ( →Q, ←R, →Nrm )
  call update ( →R, ←Q )
end do
```

with "→" and "←", inputs and outputs are implied respectively.

Typically, in order to solve the dual, one would supply this loop to an AD tool (e.g. *Tapenade*) for differentiation, producing, therefore, an adjoint pseudo- time-stepping loop that has the reserve code statements order than the primal :

```
do nIter = mIt,1,-1
  call update_b ( Q, →Q̄, R, ←R̄) )
  call residual_b ( Q, ←Q̄, R, →R̄, Nrm, ←Nrm̄ )
end do
```

where the overline implies adjoint quantities and reverse differentiated subroutines. In this case, in every iteration, an updated value of the adjoint residuals is calculated calculated as :

$$\overline{R}^{i+1} = \overline{R}^{i} - \delta t \cdot \overline{Q}^{i} \tag{13}$$

It can be observed that this form of the adjoint pseudo-time-stepping loop contains the calculation of the derivative of the residuals with respect to the normals' perturbation ($\overline{\texttt{Nrm}}$). This computation though only needs to be performed once, after the dual is converged, avoiding in this way the extra computational cost. Furthermore, it is obvious that adjoint variables $\overline{\texttt{q}}$ need to be initialized, as the first important operation is updating. This initialization is performed by reverse differentiating the cost function.

Considering the disadvantages of this "brute-force" application of automatic differentiation and trying to avoid them, the pseudo-timesteppig loop could be altered so as to take the form :

```
do nIter = 1,mIt
  call residual_b ( Q, ←R_b, R, →Q_b, Nrm )
  R_b = R_b + g
  call update ( ←Q_b, →R_b )
end do
```

where only the subroutine `residual` has been supplied for automatic differentiation. Here, in every iteration, the updated adjoint variables are calculated as :

$$\overline{Q}^{i+1} = \overline{Q}^{i} - \delta t \cdot \overline{R}^{i} \tag{14}$$

In this adjoint pseudo-time-stepping loop, the same update subroutine with the primal is used (self- adjoint subroutine), the order of the arguments $\overline{\mathtt{q}}$ and $\overline{\mathtt{r}}$ has been changed and the adjoint source term `g` must be added to the adjoint residuals, before the solution update. In this way, the adjoint pseudo-time stepping loop follows the same calculation order with the primal and the same acceleration techniques can be used. Furthermore, there is no need of initialization of the adjoint variables $\overline{\mathtt{q}}$, as the first important operation is the residual computation and not the updating.

Therefore, the whole computational process to calculate the gradient of the functional using the altered pseudo-time stepping, has the form :

```
do nIter = 1,mIt
  call residual_b ( Q, ←R_b, R, →Q_b, Nrm )
  R_b = R_b + g
  call update ( ←Q_b, →R_b )
end do
call residual_nrm_b ( Q, →Q_b, R, Nrm, ←Nrm_b )
call metrics_b ( →X, ←X_b, →Nrm, →Nrm_b )
```

whereas the equivalent "brute force" process is :

```
call cost_fun_b ( →X, →Q, ←Q_b, ← cost, → 1 )
do nIter = mIt,1,-1
  call update_b ( Q, →Q_b, R, ←R_b) )
  call residual_b ( Q, ←Q_b, R, →R_b, Nrm, ←Nrm_b )
end do
call metrics_b ( →X, ←X_b, ←Nrm, →Nrm_b )
```

4 RESULTS

4.1 Validation results

For the validation of the altered adjoint time-step, a test case of calculating the sensitivity of the lift of an airfoil NACA 0012 with respect to the angle of attack is used. The far-field conditions are $Ma = 0.43$ and $\alpha = 2^o$. At every iteration, the gradient is

computed. Considering the adjoint equivalence, tangent linearisation and adjoint should match in every iteration. The following table shows this relation :

It.	Tangent linearisation	Altered adjoint time-step
1	4.6767895663117161	4.6767895663117169
2	4.6025485792291176	4.6025485792291221
3	4.3965883079495329	4.3965883079495303
4	4.1346945501831556	4.1346945501831573
5	3.8504192698104562	3.8504192698104585
6	3.5618266817047339	3.5618266817047397
7	3.2884820797512821	3.2884820797512835
8	3.0396872942877495	3.0396872942877549
...	...	...
1700	3.1451761587343876	3.1451761587343960

Table 1: Tangent linearisation vs adjoint gradient

As mentioned before, the final value of the gradient computed via tangent linearisation or adjoint can be compared and validated versus the equivalent value computed via finite differences. For this test case and using central finite difference, the comparison with tangent linearisation and adjoint is presented in the following table :

	Gradient
Central Difference	3.1451761487549104
Tangent linearisation	3.1451761587343876
Adjoint	3.1451761587343960

Table 2: Finite difference vs tangent linearisation and adjoint

The gradient via central finite difference matches the values of the tangent linearisation and adjoint down to an expected accuracy of seven decimal digits, that is enough to verify the value of the gradient computed via AD.

4.2 Altered adjoint pseudo-time step performance results

By using the altered adjoint time-step, one avoids the computational cost of evaluating the perturbation of the residuals with respect to the normals, as described before. Also, as less primal code is being differentiated, there are overall less operations to be performed in the AD code (e.g. less calls to the help-subroutines push/pop, when using *Tapenade*). All these result in a runtime decrease, which is presented in the following table :
The cost of the two adjoint versions is therefore (compared to the cost of the primal) :

	Runtime (sec)
Primal	9.6886053
Altered adjoint	25.517594798
"Brute force" adjoint	40.826551312

Table 3: Runtime improvement for the pseudo-time stepping loop.

	Cost
Altered adjoint	2.6337738
"Brute force" adjoint	4.2138729

Table 4: Cost of the adjoints' pseudo time-stepping loop compared to the cost of the primal loop.

Even faster adjoint code can be produced by using the AD tool's pragmas in all the possible positions in the primal code and all the AD code generation improvement options (e.g. pragma *C$AD II-LOOP* and *adjointliveness* option for *Tapenade*).

4.3 Multigrid results

As mentioned before, one of the advantages of multigrid is the convergence frequency damping. Coupling the multigrid theory with the primal and adjoint code described above, this virtue of multigrid is presented in the following figure :

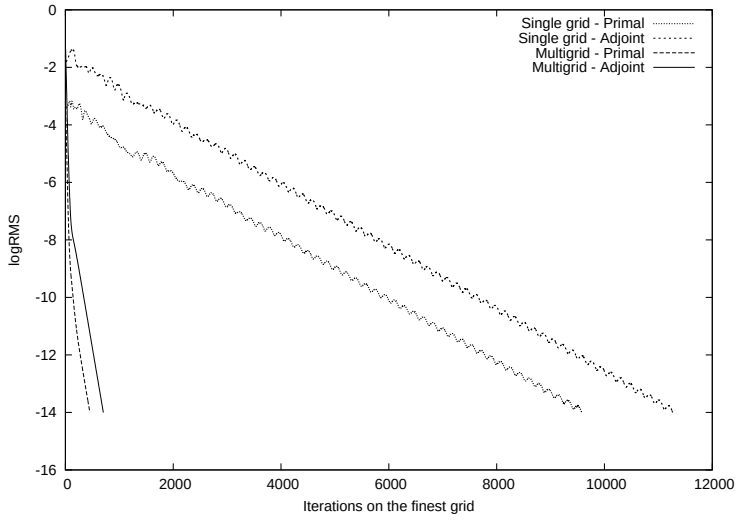

Figure 2: Convergence of primal and adjoint for single and multi grid.

Because of the frequency damping and less iterations on the "expensive" grid, multigrid

achieves a large runtime acceleration. The run-times for primal and adjoint for single and multi- grid, as well as the acceleration achieved, are presented in the next table :

	Single grid	Multigrid	Acceleration
Primal	48.563034s	11.612725s	$\sim 76\%$
Adjoint	165.65434s	51.003185s	$\sim 69\%$

Table 5: Multigrid runtime acceleration

5 DISCUSSION & CONCLUSIONS

In the previous sections, it has been shown that discrete adjoint CFD codes present a number of significant advantages compared to the continuous ones. Moreover, the way, in which the AD generated adjoint code can be validated, has been discussed. The simplicity and the exactness of this validation process really brings forward the discrete adjoint approach, compared to the continuous implementation. Last but not least, the option of coupling the discrete adjoint with solving and optimizing acceleration techniques, such as multigrid and one-shot, has been unfolded, that opens the way to even faster and more efficient discrete adjoints.

Apart from the above, the present paper makes clear that the nowadays AD tools are in the position of generating reliable discrete adjoints with high performance. Of course, as the development and improvement of the AD tools is still an on going process, even more efficient adjoint codes can be expected in the near future.

6 ACKNOWLEDGEMENTS

This research is part of the European project **FLOWHEAD** (Fluid Optimisation Workflows for Highly Effective Automotive Development Processes), funded by the European Commission under THEME SST.2007-RTD-1. http://flowhead.sems.qmul.ac.uk/

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] L. Hascoët, M. Vázquez, and A. Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V.Kumar et al., editor, *Proceedings of the International Conference on Computational Science and its Applications, ICCSA'03, Montreal, Canada*, pages 85–94. LNCS 2668, Springer, 2003.

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

[4] R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems*, 21(8):1345–55, 2005.

[5] P. Cusdin and J.-D. Müller. On the performance of discrete adjoint CFD codes using automatic differentiation. *IJNMF*, 47(6-7):939–945, 2005.

[6] A. Jaworski, P. Cusdin, and J.-D. Müller. Uniformly converging simultaneous time-stepping methods for optimal design. In R. Schilling et. al., editor, *Eurogen*, Munich, 2005. Eccomas.

[7] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical Report 0300, INRIA, 2004.