ECCOMAS Congress 2016
VII European Congress on Computational Methods in Applied Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.)
Crete Island, Greece, 5–10 June 2016

A CODE-COUPLING APPROACH TO THE IMPLEMENTATION OF DISCRETE ADJOINT SOLVERS BASED ON AUTOMATIC DIFFERENTIATION

Jan Backhaus[1], **Anna Engels-Putzka**[1] **and Christian Frey**[1]

[1]German Aerospace Center (DLR)
Institute of Propulsion Technology
Linder Höhe, D-51147 Cologne, Germany
e-mail: jan.backhaus, anna.engels-putzka, christian.frey@dlr.de

Keywords: discrete adjoint; automatic differentiation; RANS; turbomachinery

Abstract. *We propose a method for selectively applying automatic differentiation (AD) by operator overloading to develop the discrete adjoint of a turbomachinery flow solver. A fully differentiated version of the solver is generated by operator overloading using the tapeless tangent mode of ADOL-C. The differentiated solver is coupled to an undifferentiated version of the same code using message passing. The automatic differentiation is used to calculate derivatives of the flux calculation routines. The flux derivatives depending on inner cell states are sparse, and this sparsity is exploited using analytical differentiation of the spatial discretization scheme. Subsequently the sparse matrix is communicated to the undifferentiated code for solution. Turbomachinery boundary conditions may have dense Jacobians and are therefore only evaluated during the solution process. The solution of the adjoint system of equations is achieved through a preconditioned GMRES, implemented inside the undifferentiated code. A modern three dimensional contra-rotating fan stage with engineering parameterization serves as application example in order to demonstrate the technique and to perform numerical validations. The validation of gradient results is performed by comparing against results from finite differences, and the tangent forward mode.*

1 INTRODUCTION

The adjoint method is a key technology to enable gradient based optimization methods with expensive fluid dynamics simulations. It also has interesting applications in mesh adaptation, robust design and the evaluation of shape variations either due to manufacturing tolerances or wear during operations. Since the adjoint method gained popularity much later than the application of flow simulations, it is often necessary to develop consistent adjoint solvers for long existing CFD solvers. One way to obtain an adjoint solver is to adjoin the underlying PDEs and discretize these; this is called the continuous approach. The discrete approach in contrast is to take the discretized PDEs which are implemented as the primal flow solver and adjoin these. We follow the discrete adjoint approach in this publication. There are two basic techniques for developing a discrete adjoint solver: the white box and the black box approach. For the white box approach, the developer reads a calculation routine, writes down the computation in analytical form, derives its adjoint and subsequently implements the derived calculation. The black box approach is to use a tool for automatic differentiation (AD) in reverse mode, which yields the exact adjoint calculation routine. Both methods have benefits and shortcomings. The white box approach has the potential for the most efficient result, but it requires a high level of insight into the code and is susceptible for inconsistencies between the primal and the adjoint code. While the black box approach solves these white box issues it can lead to inefficient results, especially when applied to iterative solution schemes for PDEs. In practice, a combination of both techniques is often used with success. Especially the selective application of AD inside a white box framework is described by a number of publications [1, 2, 3, 4]. Automatic differentiation can be achieved in two ways: source to source transformation and operator overloading. While the source to source technique is quite popular for solvers written in Fortran, the operator overloading approach is found more advantageous for solvers written in C. The following is a description of how selective application of AD by operator overloading may be achieved in the frame of a parallel solver, starting from the adjoint solver described in [5]. TRACE is a simulation suite for internal flows, with focus on turbomachinery applications; it is developed at DLR, in productive use at MTU AeroEngines and a research tool at several universities.

2 FLOW SOLVER

The most frequently used tool for industrial turbomachinery design optimizations are the compressible steady Reynolds-averaged Navier-Stokes (RANS) equations in a rotating frame of reference, discretized by the finite volume approach. These equations have the general form

$$\frac{\partial q}{\partial t} + \operatorname{div} F(q) - S(q) = 0 \tag{1}$$

with the conservative state in each cell denoted by $q = (\rho, \rho u, \rho v, \rho w, \rho E)$, the fluxes in three directions $F = (F^1, F^2, F^3)$ with $F^i : \mathbb{R}^5 \to \mathbb{R}^5$ and the source terms $S(q)$. For the discretization of convective fluxes we use Roe's TVD upwind scheme in combination with the MUSCL approach after van Leer, viscous fluxes are calculated by a central difference scheme. The steady solution is obtained by implicit pseudo-time marching.

Turbomachinery simulations are characterized by a set of specialized boundary conditions which influence how an adjoint solver can be constructed. Boundary conditions are applied by prescribing values on extra layers, called ghost cells, which extend the mesh beyond the boundary of the physical domain. We therefore divide the flow state into states in internal and

external cells

$$q = (q_{int}, q_{ext}) \tag{2}$$

where the external cells values are prescribed by a functional relationship

$$q_{ext} = \mathcal{F}(q_{int}, q_{ext}). \tag{3}$$

The class of non-local boundary conditions poses special challenges here. In the case of non-reflecting boundary conditions after Giles [6] the ghost-cell update is defined as the fixed point iteration

$$\Delta q_{ext}^{n+1} = \mathcal{F}(q_{int}^{n}, q_{ext}^{n}). \tag{4}$$

Whereas $\mathcal{F}$ involves a Fourier transformation in circumferential direction in order to suppress incoming waves at interfaces of the computational domain.

Blade rows with different rotational velocities are, in the steady case, coupled by Denton's mixing plane approach [7], which requires circumferentially mixed out states to become identical on each side of the mixing plane. Circumferential averaging of states is performed by band-wise integration over fluxes and then applying the inverse flux function to obtain the integral state. This reads in cylindrical coordinates

$$\bar{q}^F = F_c^{-1}\left(\frac{1}{\Delta\vartheta}\int_0^{\Delta\vartheta} F_c(q)d\vartheta\right), \tag{5}$$

where Δq_{ext}^{n+1} vanishes in the case of a converged flow solution.

The boundary conditions mentioned before depend on flow states in a circumferential direction. A span-wise dependency of states may additionally occur at exit surfaces, when the static pressure at the exit is prescribed by the radial equilibrium condition of pressure and centrifugal forces

$$dp = \rho V_\vartheta^2 \frac{dr}{r}, \tag{6}$$

whereas p denotes the static pressure, V_ϑ the circumferential component of the flow velocity and r the radius.

3 ADJOINT SOLVER

For the fields of application mentioned in the introduction it is desirable to calculate partial derivatives of cost functions $I(q)$, calculated from the flow solution q which depends on many design parameters α. An efficient way to evaluate $\frac{\partial I}{\partial \alpha_i}$, with α_i denoting the i-th design parameter, for a large number of parameters is the adjoint method. Since the adjoint approach is well documented in the literature, e.g. in [8], only the basic relations are repeated here. We want to evaluate

$$\frac{\partial I(q(\alpha))}{\partial \alpha_i}, \tag{7}$$

from the flow state q which is defined implicitly by fulfilling the discretized Navier-Stokes equations

$$R(\alpha, q(\alpha)) = 0. \tag{8}$$

One way this can be calculated is solving the adjoint system for the cost function I

$$\left(\frac{\partial R}{\partial q}\right)^t \psi = \left(\frac{\partial I}{\partial q}\right)^t \tag{9}$$

and afterwards evaluate the scalar product

$$\frac{\partial I}{\partial \alpha_i} = -\psi^t \frac{\partial R}{\partial \alpha_j} \tag{10}$$

for each design parameter. The computational costs of solving Equation (9) are about as high as those for solving the nonlinear system of Equations (1) while evaluating the matrix vector product in Equation (10) is comparatively cheap. The adjoint method is therefore efficient when the number of parameters is larger than the number of cost functions.

3.1 Flux linearization

A central point in the implementation of the adjoint solver is the exact calculation of $\frac{\partial R}{\partial q}$. The linearization of the discretized steady Eqn. (1) may be written as

$$\frac{\partial R_i}{\partial q_k} = \sum_j \frac{\partial F_j}{\partial q_k} + \delta_{ik} \frac{\partial S_i}{\partial q} = 0 \tag{11}$$

where i runs over all cells, j over the cell's faces and k over all states. Since the flux at a cell face depends only on a few flow states in neighboring internal cells q_{int}, the Jacobian matrix $\frac{\partial F}{\partial q_{int}}$ can be efficiently calculated by exploiting the sparsity pattern of the spatial discretization scheme.

While the nonlinear solver can be used with a variety of turbulence models, it is assumed here, that the eddy viscosity is not influenced by small geometric changes and may therefore be ignored for the adjoint system of equations. This is called the constant-eddy-viscosity (CEV) assumption. The validity of the CEV assumption for design optimizations is discussed by various authors, e.g. [9, 10].

3.2 Adjoint boundary conditions

While the flux Jacobian for inner cell states may be calculated with the above scheme, the fluxes on cell faces close to the boundary of the computational domain depend on external cell states and through Eqn. (3) on the boundary conditions. When differentiating Eqn. (11) with respect to internal and external flow states, one obtains the residual Jacobian matrix

$$\frac{\partial R}{\partial (q_{int}, q_{ext})} = (A_{int}\ A_{ext}). \tag{12}$$

Since q_{ext} depends on q_{int} through Eqn. (4) in the case of non-reflecting boundary conditions, one would have to differentiate this giving a fixed point iteration for the linear boundary condition

$$\Delta(\delta_{ext}^{n+1}) = \frac{\partial \mathcal{F}}{\partial (q_{int}, q_{ext})} \begin{pmatrix} \delta q_{int}^n \\ \delta q_{ext}^n \end{pmatrix} \tag{13}$$

However, the adjoint system of equations is, in contrast to the primal system, not solved by pseudo-time marching. Consequently, we cannot evolve Eqn. (13) parallel to the solution process. Instead we use

$$\frac{\partial \mathcal{F}}{\partial (q_{int}, q_{ext})} \begin{pmatrix} \delta q_{int}^n \\ T \delta q_{ext}^n \end{pmatrix} = 0, \tag{14}$$

with the adjoint boundary operator T. Since no routine in the primal solver exist which can be differentiated to obtain T, the operator is manually derived and implemented. The linear

operators T for the turbomachinery boundary conditions, employed here, are derived in [5, 11] and not repeated for brevity. Using the above equations, the linearized residual reads

$$\mathscr{L} = \begin{pmatrix} A_{int} & A_{ext} \end{pmatrix} \begin{pmatrix} Id \\ T \end{pmatrix}. \tag{15}$$

The adjoint is obtained by transposition

$$\mathscr{L}^* = \begin{pmatrix} Id & T^t \end{pmatrix} \begin{pmatrix} A_{int}^t \\ A_{ext}^t \end{pmatrix}. \tag{16}$$

3.3 Solution scheme

The adjoint system of equations (9) is solved by a preconditioned GMRES solver with restarts [12]. The available preconditioners are successive over-relaxation, (SSOR) and incomplete LU-decomposition with limited level-of-fill (ILU).

3.4 Selective use of AD

The introduction of AD into TRACE is described in [13]. The results of this work are used here as a starting point to describe how an AD based adjoint solver can be implemented in a grey box fashion.

More specifically we apply white box techniques to the spatial discretization stencil, objective functions and boundary conditions, while the flux calculation routines are differentiated in a black box fashion through AD in forward mode. From forward derivatives of the flux calculation routines we then build the adjoint system matrix. This grey box approach was chosen in order to obtain an efficient adjoint solver but at the same time avoid the cumbersome differentiation of the sophisticated flux computation routines. The reasons for treating boundary conditions differently from the rest of the flux computations are:

1. The primal boundary conditions are not implemented in simple function calls which could be treated by AD. They are constituted by a composition of functions controlled by the different solver modes and choices of models.

2. The fixed-point property of non-reflecting boundary conditions as discussed in section 3.2.

3. Non-local boundary operators, such as the non-reflecting boundary conditions, radial equilibrium boundary conditions as well as row coupling interfaces depend on a large number of cell states. The differentiation of these would add large dense contributions to the residual Jacobian matrix. In order to keep the memory consumption manageable they must therefore be evaluated on-the-fly during the adjoint solution process.

We employ the tangent forward mode of automatic differentiation which calculates directional derivatives by attaching to each variable in the calculation an additional derivative variable and propagating this derivative value alongside the computation:

$$y = f(x),\ \dot{y} = \frac{df}{dx}\dot{x}, \tag{17}$$

where $\frac{df}{dx}$ is analytically defined by the chain rule of differentiation for each elementary operation, i.e., all floating operations intrinsically provided by the C language. For a more comprehensive description of AD, the reader is referred to [14]. Since TRACE is written in the C

language, more specifically the C99-standard, we apply automatic differentiation by operator overloading, due to the easier implementation and to avoid the dependency on another compiler and the restrictions to supported language constructs. The operator overloading approach works by providing a class which re-defines all elementary operations to calculate y and $\dot{y}$ as in Eqn. (17). Here we use ADOL-C [15] in tapeless tangent mode. The general workflow for applying AD by operator overloading is to change the definition of the floating point datatype used for calculations from the standard floating point datatype to the AD class (called `adouble` in ADOL-C). During the execution one sets the differentiation seed $\dot{x}$ via a special member function (called `setADValue` in ADOL-C) of the independent variable x and runs the function to be differentiated. The directional derivative is then computed alongside the computation and its derivatives $\dot{y}$ can afterwards be obtained by calling another member function (called `getADValue` in ADOL-C) for the desired output variable. While this is the simplest way of applying AD, there are a few obstacles observed during the introduction of AD in TRACE:

- Operator overloading is not defined for C, only for C++. Even though C++ is sometimes called a superset of C, this is not exactly true. Not all programs conforming to the C99-standard are also valid C++ programs. See [16] for a more detailed view on this issue. In this context the most important issues are variable length arrays (VLA), which are not supported in C++, and unions containing active floating point variables[1]. After eliminating incompatibilities it must be communicated to all developers which constructs must be avoided and how these should be replaced. Automated testing of each new code revision is necessary to ensure that no incompatibilities are introduced.

- External libraries receiving and returning floating point numbers must be differentiated, by either automatic differentiation or, if possible, by high-level manual differentiation[2].

- All floating point variables passed to an external library for output, including C's standard libraries, for which no overloaded equivalent is provided by the AD tool, must be converted to standard datatypes. This requires wrapper-code around the actual library call. For often recurring calls this may be facilitated by using macros. Functions with variable number of arguments, e.g. `printf` type functions, require more elaborate solutions, e.g. variadic templates.

- Some computations are not differentiable, e.g. calculating the length of a vector which may become zero at some point during the calculation. For such results the AD value $\dot{y}$ takes the special values NaN or inf which propagate through the whole derivative computation. While most of the time it is simple to circumvent such calculations by improvements to the primal code (cf. [14] for a list of applicable techniques), finding and identifying such computations requires thorough testing.

3.5 Interfacing differentiated with non-differentiated code

When differentiating an existing code base using AD, one has to determine which variables are active, i.e., if variables are used inside the dependency path between the selected inputs and outputs. In the operator overloading approach a variable is marked as active by changing its datatype from the standard arithmetic type to the respective AD class. It is necessary to correctly

[1]The C++11 standard improves on some of these issue, but it is still necessary to define constructors for such unions

[2]The fast fourier transformation is a candidate for this

identify all variables that have to be active. Any missed out declaration either leads to compile time errors, or to wrong gradient results. Superfluously marking a passive variable as active is less problematic; this only increases the computational effort and memory requirements. Since correctness has to be reached before performance optimizations can be considered, one often starts by changing all floating point type declarations to the AD type and handle only the interface between this active types and the external libraries.

Selectively changing variables reduces the performance overhead to only those routines one wants to differentiate. However this requires more work from the developer to handle the interfaces between active and inactive variables. At these interfaces, all conversions from active to inactive variables have to be explicitly implemented by introducing conversion calls. Composite types complicate things further, since they may be used in contexts where their member variables are active in others where they are passive. Even combinations where only parts of the composite type have to be active may occur. A solution would be to create copies of the data type for each pattern of activation, copy and adapt the calling routines. Since the activation pattern may depend on usage scenarios, this would lead to a lot of copied routines with slightly varying activation patterns which is undesirable for code-maintainability reasons. This approach may be automatized by a script. However that script has to mimic parts of a source-to-source AD tool and therefore may become complicated, depending on the range of supported language constructs.

In order to circumvent the development of such a script another approach for the selective application was chosen here: Two executables are generated from the same code base using a compile time switch. In the unmodified executable, all variables are defined as standard-arithmetic types. In the differentiated executable, all variables are defined as the AD class. Both contain the same set of functions and may be tested independently from each other to ensure binary identical results from their primal calculations. In order to use exact derivatives inside the faster unmodified executable, the unmodified launches the differentiated executable and both perform the basic flow solver initialization, c.f. Fig. 1. The flux Jacobian matrix is generated inside the differentiated solver by iterating over each component of the flow state q^i in each internal cell, setting $\dot{q}^i$ to 1. After calling the flux-functions for the respective cell, the derivatives are obtained by reading all $\dot{F}^j$ as standard floating point values and storing these inside the flux Jacobian matrix in block compressed row storage (BCRS) format, where each submatrix is a 5x5 matrix. This matrix is subsequently communicated to the non-differentiated executable, where it constitutes the system matrix for the adjoint system of equations. Since the differentiated executable is terminated at this point, the additional memory for storing the differentiated flow field is freed. The adjoint system of equations is solved by inside the non-differentiated code as described in section 3.3.

4 VALIDATION

4.1 Procedure

For the validation of the sensitivities we now have three different validation procedures available. The first two are performed by running the complete code for a deformed mesh, and these are therefore primal procedures. One can run the undifferentiated executable calculating finite differences from the results, or one can run the differentiated executable and use the deformation at each mesh node as a seeding direction. These procedures are called FD-primal and AD-primal respectively. The other means of validation is to use finite differences instead of AD in the grey box adjoint code. This is called FD-adjoint in the following text. The solver to be

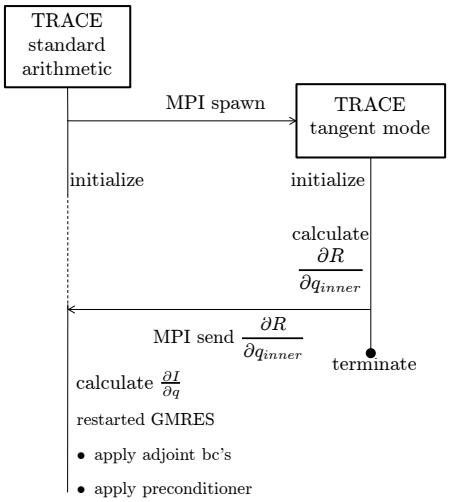

Figure 1: Sequence diagram for the code coupling strategy.

validated is the AD based adjoint, called AD-adjoint, consequently.

For the FD-primal evaluation first order accurate forward differences are obtained from the first terms of the Taylor series expansion

$$\frac{dI}{d\alpha_i} = \frac{I(\mathbf{x} + \mathbf{p}h) - I(\mathbf{x})}{h} + \mathcal{O}(h). \tag{18}$$

The grid coordinates $\mathbf{x}$ are perturbed by a deformation vector $\mathbf{p}$ which is obtained by modifying one design parameter α_i and generating the corresponding computational grid

$$\mathbf{p} = \mathbf{x}(\boldsymbol{\alpha} + \mathbf{e}_i\epsilon_i) - \mathbf{x}(\boldsymbol{\alpha}) \tag{19}$$

The choice of h is a difficult task, due to the linear dependence of the truncation error on the one hand, and cancellation errors on the other hand. Therefore different values of h are examined to find a range where the result is insensitive to the choice of h. Finite differences have the advantage of being very simple and they serve therefore as reference to rule out human error or flaws in the AD tool. However, as the associated approximation error may become very large finite differences cannot provide full confidence in the correctness of the results. The second simple approach is the AD-primal procedure, which serves as an alternative way to calculate derivatives in a forward manner. It has no associated step-width problem. This AD-primal calculates any cost function I available inside the flow solver and $\dot{I}$ with respect to one design parameter provided as deformed mesh in one sweep

$$I(x),\ \dot{I} = \frac{\partial I}{\partial \mathbf{x}}\mathbf{p}. \tag{20}$$

For $\mathbf{p}$ one can use the mesh deformations as in Eqn. (19) and apply these as seed direction for the AD variables on the grid points. In principle, this is the same as the finite difference evaluation described before with a step width approaching zero and no subtractive cancellation present. This is equivalent to the complex step derivative technique [17], which could be used alternatively. Since both primal procedures have a computational cost proportional to the number of design parameters, they are only used for validation purposes.

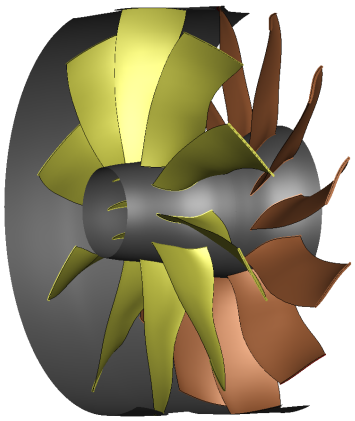

Figure 2: The contra-rotating integrated shrouded propfan CRISP 2.

The third solver is the FD-adjoint, where the flux Jacobian matrix is computed from finite differences. By means of this solver we are able to tell whether the Jacobian matrix calculated by the AD tool is valid and to assess the influence of errors in this matrix on the adjoint sensitivities. The finite difference step-width here is chosen to be proportional to the flow state and the factor of proportionality ε is varied to find a range of insensitivity of the finite differences.

4.2 CRISP 2

The second design of the contra-rotating shrouded propfan called CRISP 2 (Fig. 2) is a multidisciplinary optimization based design conducted at DLR [18] using the optimization framework AutoOpti [19]. The design targets are to develop a new engine concept for extremely high bypass-ratios (>20), in order to achieve a high fan efficiency and to reduce noise emissions for the important operating points. To this end, aerodynamic, acoustic and mechanic objectives or constraints were considered in the optimization [20]. The optimization has been carried out using a combination of evolutionary algorithms and Gaussian process meta models constructed from 3D simulations. The design used here has a mass flow rate of 158 kg/s, a total pressure ratio of 1.3 and an isentropic efficiency of 94 %. With a pre-shock Mach number of about 1.2 the flow is transonic. A validation of the CFD solver on modern contra-rotating fans against experimental results is described in [21]. See [22] for a description of how gradients from an adjoint solver may be included in this optimization. Different mesh resolutions can be generated from this process, here a coarse structured mesh consisting of 500 000 cells is used.

The CAD Model of the fan stage is parameterized by engineering CAD Parameters which are translated into B-Spline tensor product surfaces using the geometry tool Blade-Generator. Based on these surface definitions a structured grid is created using the mesh generator G3DHexa. Deformed meshes for the sensitivity calculation process are created by an elliptic mesh deformation tool [23].

From the 123 parameters of the original optimization we have selected 24 representative parameters. This was necessary to reduce the computational effort spent on the primal validation procedures. The selected parameters are the stagger angle, leading edge angle and trailing edge angle on four profiles of each of the two rotors.

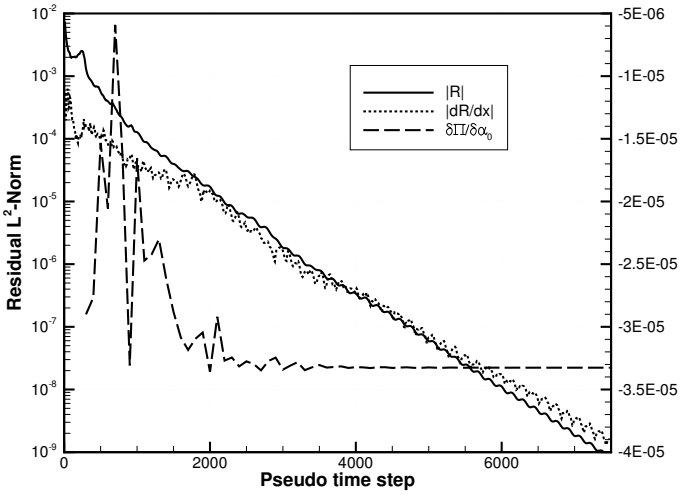

Figure 3: Residual, differentiated residual and convergence of sensitivity of the total pressure ratio for the AD-primal process

4.3 Validation using primal calculations

The convergence of the primal residual and the residual of the forward differentiation, AD-primal, are shown in Fig. 3, additionally for the total pressure ratio Π the evolution of the sensitivity $\frac{\partial \Pi}{\partial \alpha_0}$ is plotted exemplarily. All primal validation calculations are stopped when the residual falls below a threshold of 10^{-9}. While the residual still drops afterwards, the accuracy in all output quantities is sufficient, as can be seen for the value of the sensitivity in Fig. 3.

4.4 Validation using FD-adjoint

The adjoint solver based on finite differences is used to demonstrate the effect of approximations on the residual Jacobian matrix here. From Fig. 4 it can be seen that the convergence of the adjoint residual improves, when a larger finite difference step is used. Errors from the finite differences act as a regularization of the adjoint system matrix. The convergence behavior of the AD-adjoint solver is identical to the convergence with $\varepsilon = 10^{-11}$. However, the smallest error in the sensitivities for the first parameter, comparing AD-adjoint to FD-adjoint, can be observed for $\varepsilon = 10^{-4}$ in Fig. 5.

4.5 Comparison of Gradients

Figure 6 shows the comparison of the calculated gradients for the total pressure ratio from the four different approaches. One can see that these are in very good agreement. Minor deviations can be observed for some parameters, between the sensitivities from the primal and the adjoint approaches. Since the AD approaches show the same results as the FD approaches, an error in the AD tool or its application can be ruled out. The absolute errors of the gradients are calculated as the absolute value of the difference between AD-adjoint and AD-primal and plotted in Fig. 7.

The magnitude of errors is found to be sufficiently small for the optimization procedures described in [22].

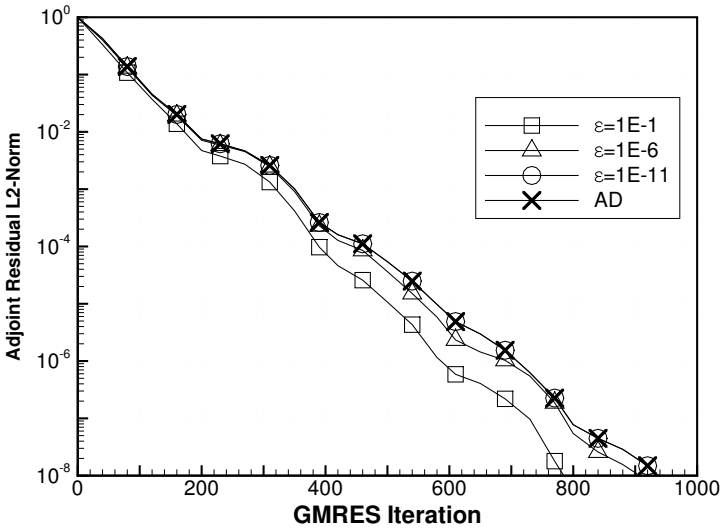

Figure 4: Residuals of the FD-adjoint process with varying step width ϵ.

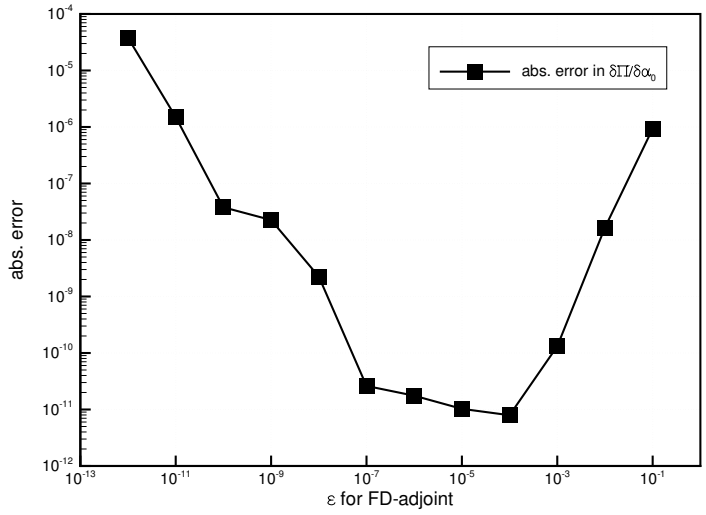

Figure 5: Errors in sensitivities for FD-Adjoint compared to AD-adjoint over step width ϵ.

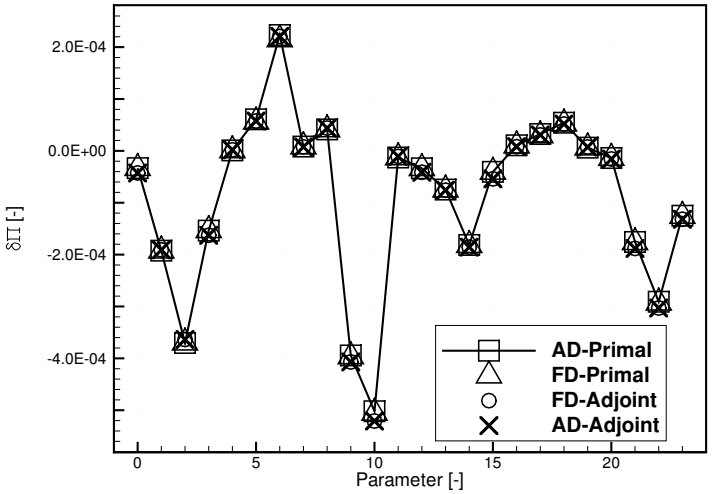

Figure 6: Validation of partial derivatives for the total pressure ratio of CRISP 2 w.r.t 24 CAD parameters.

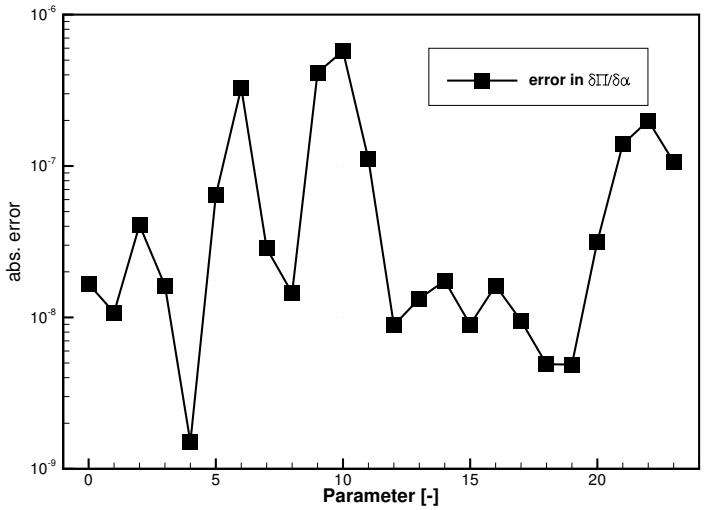

Figure 7: Error of the AD-adjoint sensitivities compared to AD-primal.

5 CONCLUSIONS AND OUTLOOK

Developing the adjoint for an existing CFD solver can become a laborious task which requires good knowledge of the solver and of the equations implemented therein. This amount of knowledge may be replaced, in parts, by the ability to work with AD tools. Correctly applied AD provides exact derivatives which stay consistent to the calculations, even when the solver is modified, given that certain coding guidelines are obeyed. Besides a lot of effort spent on the development of the AD tools, a complete black box approach is still not optimal for the iterative solution processes. The approach presented here shows how AD can be selectively applied for an industrial, parallelized turbomachinery solver written in the C language by using operator overloading. The approach taken here starts from a white box derivation and selectively includes black box techniques. A benefit of the approach is the ability to use the same AD-primal solver which is used to calculate the flux Jacobian for gradient validation purposes and therefore gain confidence in the consistency of the computed flow solution and the derivatives. An alternative approach is to derive an adjoint solver in a complete black box fashion through reverse mode AD and afterwards introduce white box improvements to lower the runtime and memory consumption. While this strategy has been pursued in [13], a thorough comparison of both approaches will be left for future work.

ACKNOWLEDGEMENT

This research was supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) under grant number 20T1104B. We thank the Chair for Scientific Computing (SciComp) of TU Kaiserslautern, especially Max Sagebaum and Dr. Emre Özkaya for the fruitful cooperation in applying automatic differentiation.

REFERENCES

[1] F. Courty, A. Dervieux, B. Koobus, and L. Hascoet, "Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation," *Optimization Methods and Software*, vol. 18, no. 5, pp. 615–627, 2003.

[2] M. Giles, D. Ghate, and M. Duta, "Using automatic differentiation for adjoint cfd code development," *Indo-French Workshop*, 2005.

[3] C. A. Mader, J. R. R. A. Martins, J. J. Alonso, and E. van der Weide, "ADJoint: An approach for the rapid development of discrete adjoint solvers," *AIAA Journal*, vol. 46, pp. 863–873, APR 2008. AIAA/ISSMO 11th Multidisciplinary Analysis and Optimization Conference, Portsmouth, VA, SEP 06-08, 2006.

[4] A. C. Marta, S. Shankaran, D. G. Holmes, and A. Stein, "Development of adjoint solvers for engineering gradient-based turbomachinery design applications," in *Proceedings of the ASME Turbo Expo 2009*, vol. Volume 7: Turbomachinery, Parts A and B, June 2009.

[5] C. Frey, D. Nürnberger, and H.-P. Kersken, "The discrete adjoint of a turbomachinery RANS solver," in *Proceedings of ASME-GT2009*, 2009.

[6] M. B. Giles, "Nonreflecting boundary conditions for Euler calculations," *AIAA Journal*, vol. 28, no. 12, pp. 2050–2058, 1990.

[7] J. Denton and U. Singh, "Time marching methods for turbomachinery flow calculations," VKI Lecture Series 1979-7, von Karman Institute., 1979.

[8] M. B. Giles and N. A. Pierce, "An introduction to the adjoint approach to design," *Flow, Turbulence and Combustion*, vol. 65, pp. 393–415, 2000.

[9] C. S. Kim, C. Kim, and O. H. Rho, "Feasibility study of constant eddy-viscosity assumption in gradient-based design optimization," *Journal of Aircraft*, vol. 40, no. 6, pp. 1168–1176, 2003.

[10] R. Dwight and J. Brezillon, "Effect of approximations of the discrete adjoint on gradient-based optimization," *AIAA Journal*, vol. 44, no. 12, pp. 3022–3071, 2006.

[11] C. Frey, A. Engels-Putzka, and E. Kügeler, "Adjoint boundary conditions for turbomachinery flows," in *ECCOMAS 2012 - European Congress on Computational Methods in Applied Sciences and Engineering, e-Book Full Papers*, 2012.

[12] Y. Saad, *Iterative methods for sparse linear systems. 2nd ed.* SIAM Society for Industrial and Applied Mathematics, Philadelphia., 2003.

[13] M. Sagebaum, E. Özkaya, and N. R. Gauger, "Challenges in the automatic differentiation of an industrial CFD solver," in *Evolutionary and Deterministic Methods for Design, Optimization and Control with Application to Industrial and Societal Problems (EUROGEN 2013)*, 2014.

[14] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Siam, 2008.

[15] A. Walther and A. Griewank, "Getting started with ADOL-C," *Combinatorial Scientific Computing*, pp. 181–202, 2012.

[16] B. Stroustrup, "C and C++: Case studies in compatibility," *C/C++ Users Journal*, vol. 20, no. 9, pp. 22–31, 2002.

[17] J. R. R. A. Martins and J. T. Hwang, "Review and unification of methods for computing derivatives of multidisciplinary computational models," *AIAA Journal*, vol. 51, pp. 2582–2599, November 2013.

[18] T. Lengyel-Kampmann, *Vergleichende aerodynamische Untersuchungen von gegenläufigen und konventionellen Fanstufen für Flugtriebwerke:*. PhD thesis, Ruhr-Universität Bochum, 2015.

[19] C. Voss, M. Aulich, B. Kaplan, and E. Nicke, "Automated multiobjective optimisation in axial compressor blade design," in *ASME Paper*, vol. 90420, 2006.

[20] D. Görke, A.-L. Le Denmat, T. Schmidt, F. Kocian, and E. Nicke, "Aerodynamic and mechanical optimization of CF/PEEK blades of a counter rotating fan," in *Proceedings of the ASME Turbo Expo 2012*, 2012.

[21] T. Lengyel-Kampmann, A. Bischoff, R. Meyer, and E. Nicke, "Design of an economical counter rotating fan: Comparison of the calculated and measured steady and unsteady results," in *ASME Turbo Expo 2012: Turbine Technical Conference and Exposition*, pp. 323–336, American Society of Mechanical Engineers, 2012.

[22] J. Backhaus, M. Aulich, C. Frey, T. Lengyel, and C. Voß, "Gradient enhanced surrogate models based on adjoint cfd methods for the design of a counter rotating turbofan," in *Proceedings of the ASME Turbo Expo 2012*, 2012.

[23] C. Voigt, C. Frey, and H.-P. Kersken, "Development of a generic surface mapping algorithm for fluid-structure-interaction simulations in turbomachinery," in *V European Conference on Computational Fluid Dynamics ECCOMAS CFD 2010* (J. C. F. Pereira, A. Sequeira, and J. M. C. Pereira, eds.), June 2010.