Available online at www.sciencedirect.com

Future Generation Computer Systems 21 (2005) 1333–1344

FGCS Future Generation Computer Systems

www.elsevier.com/locate/fgcs

Efficient and accurate derivatives for a software process chain in airfoil shape optimization

C.H. Bischof[a], H.M. Bücker[a], B. Lang[b], A. Rasch[a,*], E. Slusanschi[a]

[a] *Institute for Scientific Computing, RWTH Aachen University, D-52056 Aachen, Germany*
[b] *Applied Computer Science Group, University of Wuppertal, D-42097 Wuppertal, Germany*

Available online 16 December 2004

Abstract

When using a Newton-based numerical algorithm to optimize the shape of an airfoil with respect to certain design parameters, a crucial ingredient is the derivative of the objective function with respect to the design parameters. In large-scale aerodynamics, this objective function is an output of a computational fluid dynamics program written in a high-level programming language such as Fortran or C. Numerical differentiation is commonly used to approximate derivatives but is subject to truncation and subtractive cancellation errors. For a particular two-dimensional airfoil, we instead apply automatic differentiation to compute accurate derivatives of the lift and drag coefficients with respect to geometric shape parameters. In automatic differentiation, a given program is transformed into another program capable of computing the original function together with its derivatives. In the problem at hand, the objective function consists of a sequence of programs: a MATLAB program followed by two Fortran 77 programs. It is shown how automatic differentiation is applied to a sequence of programs while keeping the computational complexity within reasonable limits. The derivatives computed by automatic differentiation are compared with approximations based on divided differences.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Automatic differentiation; Forward mode; Seeding; Concatenated programs; Shape optimization

1. Introduction

Demand for air travel is growing rapidly. Airline costs and passenger convenience have led to an increasing interest in maximizing take-off and landing frequencies, as these heavily affect the efficiency of airport and aircraft operations. A detailed knowledge of the wake flow field is essential to estimate safe-separation distances between aircraft in take-off and landing. However, the complex flow field around an

* Corresponding author. Present address: Institute for Scientific Computing, RWTH Aachen University, Seffenterweg 23, D-52074 Aachen, Germany. Fax: +49 241 8022241.
E-mail addresses: bischof@sc.rwth-aachen.de (C.H. Bischof); buecker@sc.rwth-aachen.de (H.M. Bücker); lang@math.uni-wuppertal.de (B. Lang); rasch@sc.rwth-aachen.de (A. Rasch); slusanschi@sc.rwth-aachen.de (E. Slusanschi).

aircraft is still not completely understood. In the context of a collaborative research center (SFB 401) at RWTH Aachen University, a team of engineers, mathematicians, and computer scientists is investigating the fluid–structure interaction at airplane wings to further broaden our knowledge of the physical phenomena underlying the aerodynamics of cruise and high lift configurations. Much of this work relies on numerical simulations. As one of the first preparatory steps toward the long-term goal of optimizing the wake flow field, one of the projects currently aims at optimizing an airfoil with respect to certain design parameters. Traditionally, finding a suitable set of parameters is carried out by running a simulation code over and over again with perturbed inputs. However, this approach may consume enormous computing time and may also require an experienced user to select appropriate sets of parameters just to achieve improvement, even without optimality. Here, numerical optimization techniques can help reduce the number of simulation runs, and in particular provide more goal-oriented computational support for a design engineer.

Typically, a simulation code is written in a high-level programming language such as Fortran, C, C++, or MATLAB. The simulation may also involve running several codes written in different programming languages. When embedding such a “pure” simulation code in a gradient-based optimization framework, the derivatives of the simulation output with respect to the design parameters are needed. A common misconception is that these derivatives can only be obtained from divided differences with all the disadvantages of numerical differentiation including truncation and cancellation errors and the associated problem of finding a suitable step size. Fortunately, there is a remedy for efficiently getting reliable and accurate derivatives in the situation where the function is given in the form of computer codes. This technique is called algorithmic or automatic differentiation (AD) [1,2] and provides easy access to derivatives. AD comprises a set of techniques where a given computer program is automatically transformed into another program capable of evaluating not only the original simulation but also derivatives of selected outputs with respect to selected inputs. In contrast to numerical differentiation, derivatives computed by AD are free from truncation and cancellation errors.

Applying AD in the context of computational fluid dynamics (CFD) is not new; see the recent book [3] for a general overview. The goal of this note is to demonstrate the feasibility of AD for a particular aerodynamic sensitivity study of a two-dimensional airfoil. More precisely, we are interested in the derivatives of the lift and drag coefficients with respect to eight geometric shape parameters. An important issue of the problem at hand is that the function to be differentiated consists of a sequence of three programs: a MATLAB program generating an airfoil from a given base airfoil and some geometric shape parameters, a Fortran 77 program that takes an airfoil and some control parameters to generate a mesh around the airfoil, and a Navier–Stokes flow solver, also written in Fortran 77, to compute the flow on a given mesh. A similar approach is studied in [4] where AD is applied to the WTCO wing grid generator and the TLNS3D Navier–Stokes flow solver, both Fortran 77 programs. Details on developing an AD-enhanced version of TLNS3D are given in [5,6]. Another application of AD applied to a grid generator called CSCMDO which is written in C is described in [7]. An airfoil optimization is performed in [8] where the drag coefficient is minimized for constant lift using an Euler solver. Further references where automatic differentiation has been applied to problems from computational fluid dynamics include [9–11].

The structure of this note is as follows. Section 2 briefly reviews the basics of AD. In Section 3, the actual simulation consisting of a sequence of three codes is described, and the procedure of applying AD to this sequence is outlined in Section 4. Numerical experiments including a comparison with a numerical differentiation approach based on divided differences are reported in Section 5. Directions for future research are outlined in Section 6.

2. Automatic differentiation

The term “Automatic Differentiation (AD)” comprises a set of techniques for automatically augmenting a given computer program with statements for the computation of derivatives. That is, given a computer program C that computes a function $f : \mathbb{R}^n \to \mathbb{R}^m$, automatic differentiation yields another “differentiated” program C' that, for any input $x \in \mathbb{R}^n$, not only evaluates f but also its Jacobian $J \in \mathbb{R}^{m\times n}$ at the same input

x. The AD technology is applicable whenever derivatives of functions given in the form of a programming language, such as Fortran, C, C++, or MATLAB, are required. The reader is referred to the recent book by Griewank [2] and the proceedings of AD workshops [12–14] for details on this technique. In automatic differentiation the chain rule of differential calculus is applied over and over again to elementary statements such as binary addition or multiplication for which derivatives are known, combining these step-wise derivatives to yield the derivatives of the whole program.

One well-known strategy for applying the chain rule is the so-called forward mode of AD. Program output variables whose derivatives are to be computed by AD are called *dependent variables*; program input variables with respect to which one is differentiating are known as *independent variables*. If one is interested in obtaining n directional derivatives of f, a derivative object u^{∇} (a length-n row vector) is associated with every intermediate variable u involved in the evaluation of the function f. If u is a compound object, its associated derivative object u^{∇} needs to store n scalar derivative values for every scalar entry of u. For every operation of the original code C involving a variable u, an additional operation is performed in the differentiated code C'. For instance, let the original code C perform a multiplication $u = vw$. Then, the differentiated code C' is given by

$$u = vw, \qquad u^{\nabla} = v^{\nabla} \odot w + v \odot w^{\nabla},$$

where the symbol $\odot$ denotes an appropriate multiplication with a derivative object.

The actual derivative information computed this way is determined by the initialization of the derivative objects of the independent variables, x_j^{∇}. Setting x_j^{∇} to the jth row of the $n \times n$ identity matrix makes the Jacobian $J = \partial f/\partial x \in \mathbb{R}^{m\times n}$ available after the computation. If, instead, x_j^{∇} is initialized to the jth row of some *seed matrix* $S \in \mathbb{R}^{n\times k}$ with $k < n$ then the product $J \cdot S$ is returned, allowing k directional derivatives to be computed very efficiently since all intermediate derivative objects are only of length k. Appropriate seeding can be critical in terms of performance under certain circumstances. Note that issues such as sparsity of the Jacobian or high-level knowledge of the function to be differentiated may also be exploited to significantly increase the performance of AD-generated code [2].

Another AD strategy is the so-called reverse mode. In contrast to the forward mode propagating derivatives along with the control flow of the original function, the reverse (or backward) mode of AD generates derivatives by reversing the control flow of the underlying function. As the reverse mode involves derivative objects of length m, it is attractive in terms of computation time if the number of dependent variables is significantly less than the number of independent variables [2]. Reversing the flow for the derivative computations requires saving intermediate results on a so-called *tape* or recomputing them.

Except for very simple cases, the function f cannot be evaluated within a single routine. Instead, evaluating f typically involves a large subtree of the whole program, the “top-level” routine of this tree invoking a multitude of lower-level routines and finally providing the function value. In such a case automatic differentiation must be applied to the whole subtree, often totaling several hundreds of routines and tens or even hundreds of thousands lines of code, in order to obtain the function’s derivatives.

Implementing AD for a certain programming language is usually done using either the source transformation approach or the operator overloading approach. The latter is applicable only in object oriented languages that support overloading of operators. Then each overloaded operator executes the derivative statements in addition to the operation in the original code. Usually, this approach has a myopic view regarding the data dependencies, because an operator has only access to its own operands. The operators have no knowledge about the dependencies between variables, and computing the dependencies at runtime is complicated. This limits the inherent flexibility provided by the associativity of the chain rule and the potential performance gains thereby possible [15]. Several AD tools available for various widely used programming languages implement the operator overloading approach without a complete dependency analysis (e.g., ADOL-C [16] and FADBAD [17] for C++, and ADOL-F [18] for Fortran 90).

By contrast, a tool implementing the source transformation approach analyzes the whole code and can identify those variables that need derivative objects and the statements that need derivative computations associated to them. The source code is then augmented, potentially using AD rules whose scope extends beyond a single operation, and optimizations may be applied. The result is a new code which computes the original function and its derivatives. AD tools based on source transformation include Adifor [19,20], TAF (formerly TAMC) [21], Odyssée [22], Tapenade [23] (all for Fortran code), and ADIC [24] for C/C++.

An overview of available AD tools and of applications where AD methods have been applied may be found at http://www.autodiff.org.

3. The computational chain

The optimization problem considered in this work consists of maximizing an objective function $f(\xi_1, \ldots, \xi_n)$, where the ξ_j denote geometric parameters defining the airfoil (cross-section of the wing), and f is some integral quantity obtained from the flow around the airfoil, e.g., the ratio of the lift and drag coefficients, c_L/c_D. The objective function f is computed in three stages; see Fig. 1.

The first stage generates the actual airfoil by superposing a given “base airfoil” A_0 with so-called “mode functions” A_j:

$$A(\xi) = A_0 + \sum_{j=1}^{n} \xi_j A_j.$$

The number of mode functions is small, a typical value being $n = 8$. The airfoil A is described via the coordinates of roughly 200 points lying on its boundary, and these coordinates are then fed into a second program, an elliptic grid generator, which produces a multi-block structured grid G. The grid in Fig. 1 consists of four blocks whose boundaries are also shown in the bottommost picture. Finally, a CFD solver is used to determine the pressure (p) and velocity fields of the flow around the airfoil, from which the objective value f is easily obtained. The flow solver in our experiments is the TFS program [25,26], which has been developed at the Aerodynamics Institute, RWTH Aachen University. TFS solves the Navier–Stokes equations of a compressible ideal gas in two or three space dimensions with a finite volume formulation. The spatial discretization of the convective terms follows a variant of the advective upstream splitting method (AUSM$^+$) proposed by Liou [27,28]. The viscous stresses are centrally discretized to second-order accuracy. The implicit method uses a relaxation-type linear solver whereas the explicit method relies on a multigrid scheme.

For efficiency reasons gradient-based numerical optimizers should be used to solve the problem. These algorithms require the derivatives $\partial f/\partial \xi$ of the whole computational chain.

4. Automatic differentiation of the computational chain

As the computational chain represents a composition of three functions,

$$f = f(G(A(\xi))),$$

its derivative can be obtained by differentiating each of the three stages and combining these derivatives according to the chain rule:

$$\frac{\partial f}{\partial \xi} = \frac{\partial f}{\partial G} \cdot \frac{\partial G}{\partial A} \cdot \frac{\partial A}{\partial \xi}.$$

A naive approach would first explicitly generate the three Jacobian matrices $\partial f/\partial G$, $\partial G/\partial A$, and $\partial A/\partial \xi$ by applying automatic differentiation to each of the three programs separately. The second step would then perform two matrix–matrix multiplications. The problem with this approach when using a forward mode-based AD tool is the large number of independent variables of the flow solver. Recall that, for computing $\partial f/\partial G$, the number of independent variables equals the number of grid points. That is, the execution time of the AD-generated code for the computation of $\partial f/\partial G$ would be proportional to the execution time of the flow solver times the number of grid points. In order to compute $\partial f/\partial \xi$ more efficiently we employ the following approach.

The airfoil generator is written in the MATLAB language, for which only an AD tool based on operator overloading has been available [29]. For the reasons given in Section 2, AD tools based on source transformation can yield derivatives more efficiently, and therefore we have begun developing a new AD tool for MATLAB, called ADiMat [30], which combines

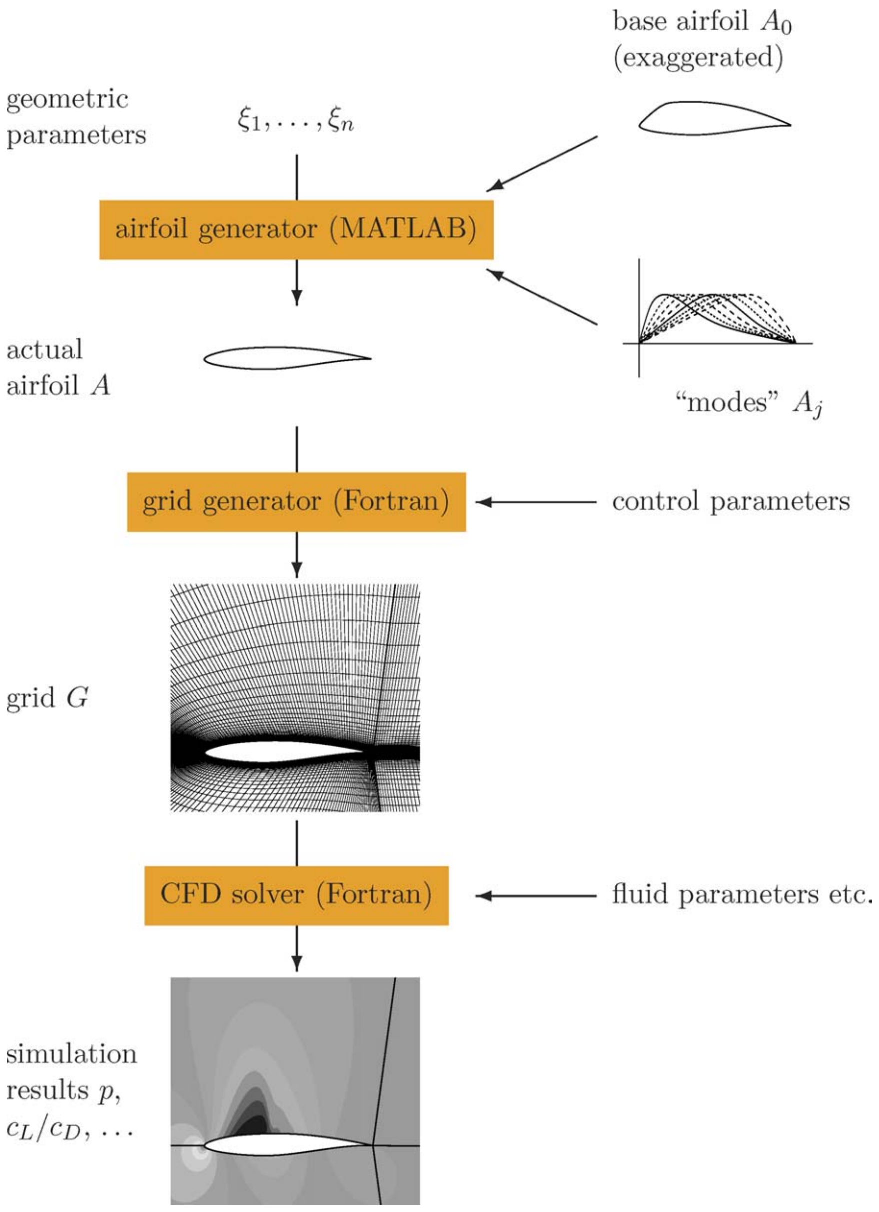


Fig. 1. Computational chain from the geometric parameters to the simulation results.

source transformation techniques with overloaded operators. Currently ADiMat is able to augment the airfoil generator with reasonably efficient derivative computations, but the efficiency of the augmented code will be further increased in future releases of the tool.

The grid generator and the TFS flow solver are written in Fortran, comprising roughly 5000 and 24,000 lines of code, respectively. After some preparations in order to force strict adherence to the Fortran 77 standard, Adifor 2.0 was used to augment these two codes with derivative computations. That is, the transformed grid generator returns not only the coordinates of the grid points but also their derivatives with respect to the coordinates of the roughly 200 points on the airfoil, and the augmented TFS program computes the derivatives of p, c_L/c_D, etc., with respect to the grid coordinates in addition to the original results. Since Adifor analyzes the whole data flow of each code, it automatically detects that only 112 out of the 227 subroutines of TFS can contribute to the requested derivatives and must be transformed. The augmented routines contain approximately 19,000 lines of code, which —together with the remaining 115 original routines—give a total of roughly 30,000 lines of code for the augmented flow solver.

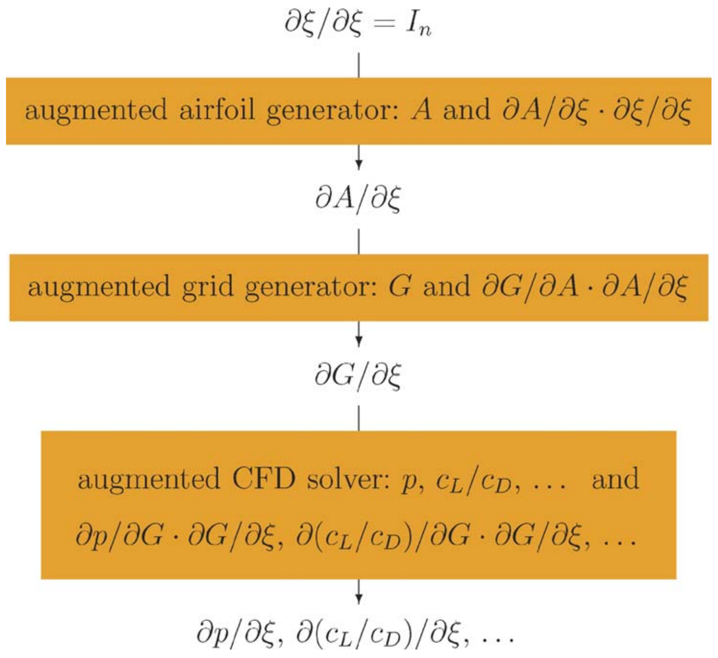
As the number of grid points is very large, naive forward mode AD of the grid generator and the flow solver would involve very long derivative objects, thus increasing the overall computing time and memory requirements by a factor of several thousands. Since we are only interested in the gradient of the scalar function c_L/c_D, the reverse mode of AD would alleviate this problem for the flow solver, but the derivative objects in the grid generator remain long in either forward or reverse mode.

The solution to this problem is appropriate *seeding*, cf. Section 2. If the gradient objects A^∇ in the augmented grid generator are initialized with the matrix $\partial A/\partial\xi$ instead of the identity matrix then this program computes the derivative $(\partial G/\partial A)\cdot(\partial A/\partial\xi) = \partial G/\partial\xi$ instead of $\partial G/\partial A$, and length-n derivative objects are used throughout, where the number of geometric parameters, n, is small. Analogously, initializing the derivative objects G^∇ in the augmented flow solver with $\partial G/\partial\xi$ leads to the required derivatives with respect to ξ being computed, and again all derivative objects are of length n. This procedure is summarized in Fig. 2.

In our implementation, files are used to transfer the derivative information from one stage to the next in the computational chain.

5. Numerical experiments

In order to illustrate the above discussion, we report on experiments performed with the computational chain described in the preceding section, on a 1733 MHz Pentium 4 Linux machine. An implicit method was used for the solution of a steady state flow problem. In this context, the iteration steps of the method are called "(local) time steps". All



$\partial\xi/\partial\xi = I_n$
augmented airfoil generator: A and $\partial A/\partial\xi\cdot\partial\xi/\partial\xi$
$\partial A/\partial\xi$
augmented grid generator: G and $\partial G/\partial A\cdot\partial A/\partial\xi$
$\partial G/\partial\xi$
augmented CFD solver: p, c_L/c_D, ... and
$\partial p/\partial G\cdot\partial G/\partial\xi$, $\partial(c_L/c_D)/\partial G\cdot\partial G/\partial\xi$, ...
$\partial p/\partial\xi$, $\partial(c_L/c_D)/\partial\xi$, ...

Fig. 2. Derivative computation in the chain.

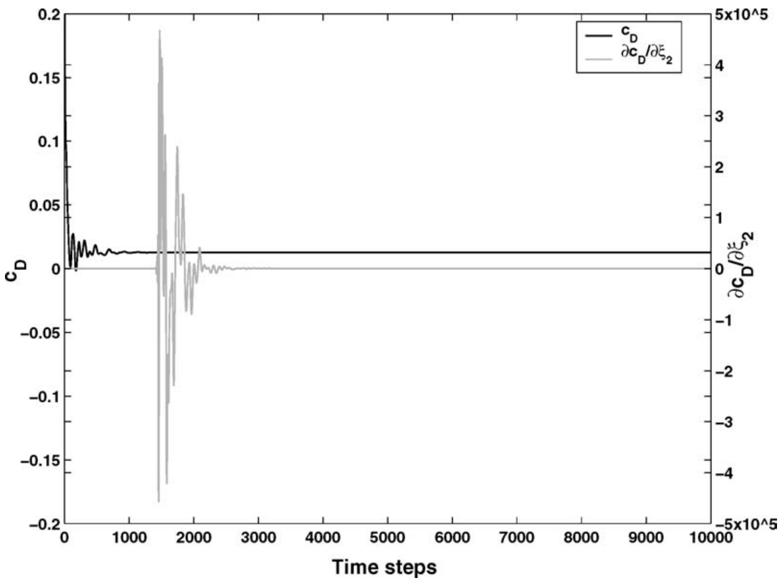


Fig. 3. Original function c_D and AD-computed derivative $\partial c_D/\partial \xi_2$.

computations were performed in double precision. In the experiments, a grid with 17,428 points is used. The transonic flow conditions involve a Mach number of 0.74 and a Reynolds number of 2.4×10^6. The angle of attack is given by 1.5593° corresponding to a lift coefficient of 0.5. On the airfoil a no slip condition and an adiabatic wall is assumed, the flow variables on the far field boundary are computed with one-dimensional linearized characteristics.

In Fig. 3 the convergence history of the computed drag coefficient, c_D, and of its derivative with respect to the second geometry parameter, $\partial c_D/\partial \xi_2$, are shown. In contrast to the original function c_D, which has converged to four decimal digits after about 2000 time steps, the AD-computed derivative takes roughly 9000 steps to achieve four correct digits. Note that the derivatives take very large intermediate values between time steps 1000 and 3000 before converging to the moderate result $\partial c_D/\partial \xi_2 \approx 0.27$. Therefore the actual convergence takes place in the grey horizontal line and is not visible in this scaling; the convergence behavior is monitored in more detail in the third column of Table 1. The reason for this convergence behavior is still an open research problem. First investigations indicate that there is no connection between the range of iterations in which the shock is set up in the original function and the iteration range in which the derivatives take the large intermediate values.

Traditionally, the derivatives of large-scale numerical simulations have been approximated with divided differences. In the following, DD will be used as a shorthand for first-order forward divided differences. Thus, the DD approximation of $\partial c_D/\partial \xi_i$ with step size h is given by

$$\frac{\partial c_D(\xi)}{\partial \xi_i} \approx \frac{c_D(\xi + he^{(i)}) - c_D(\xi)}{h}, \tag{1}$$

where $e^{(i)} = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ denotes the ith unit vector.

From a first glance at the convergence history of a DD approximation this approach might seem superior to AD because the DD values achieve four converged digits after only 4000 time steps (for $h = 10^{-4}$), whereas AD takes twice as many steps to achieve the same number of digits; see Table 1.

But a closer look reveals that the DD values for different h agree only in the first digit; see Fig. 4, which shows time steps 5000 through 10,000 of the AD-computed value $\partial c_D/\partial \xi_2$ and the DD approximations for step sizes $h = 10^{-4}$, 10^{-5}, and 10^{-6}. Thus the earlier convergence of the DD approximations, as compared with the AD values, is highly deceptive,

Table 1
Convergence history of the drag coefficient c_D and its derivative $\partial c_D/\partial \xi_2$, computed with AD and DD ($h = 10^{-4}$)

Time step	c_D	AD	DD
1000	0.0122944011	1.8358356514	0.1584697771
2000	0.0126006465	4362.8269850564	0.2447809025
3000	0.0125963656	118.8660242812	0.2338688469
4000	0.0125963622	2.3906303951	0.2341026512
5000	0.0125963642	0.0597423708	0.2341090900
6000	0.0125963642	0.2488406523	0.2341100080
7000	0.0125963642	0.2674666395	0.2341101676
8000	0.0125963642	0.2696612808	0.2341101949
9000	0.0125963642	0.2699437903	0.2341101999
10000	0.0125963642	0.2699839801	0.2341102009

veiling the fact that the limit itself agrees with the required derivative only in the first digit. As one of the DD values (for $h = 10^{-5}$) agrees well with the AD-computed derivative we have confidence to the latter being correct.

It is a well known problem that the accuracy of the DD approximation is highly sensitive to the choice of the step size h. If h is too large then the truncation error dominates, whereas the cancellation error in the numerator of the right-hand side of (1) prevents achieving high accuracy for small h. Finding an appropriate value for h may be a very time-consuming process because it involves running the function—in our case the whole computational chain—in a black-box fashion, for a number of different step sizes.

Fig. 5 shows the differences between the AD-generated derivative and DD approximations for both scalar functions of interest, c_D and c_L, with step sizes ranging from 10^{-8} to 10^{-2}. The derivative values used in the figure are those from time step 10,000, when all methods have converged to more than six digits.

Two points are worth noting. First, the *best* DD step sizes are different for c_L ($h_{\text{opt},c_L} \approx 1.1 \times 10^{-5}$) and c_D ($h_{\text{opt},c_D} \approx 5 \times 10^{-6}$). Furthermore, h_{opt,c_D} yields a bad approximation for $\partial c_L/\partial \xi_2$. For this reason the best DD approximation to the corresponding partial

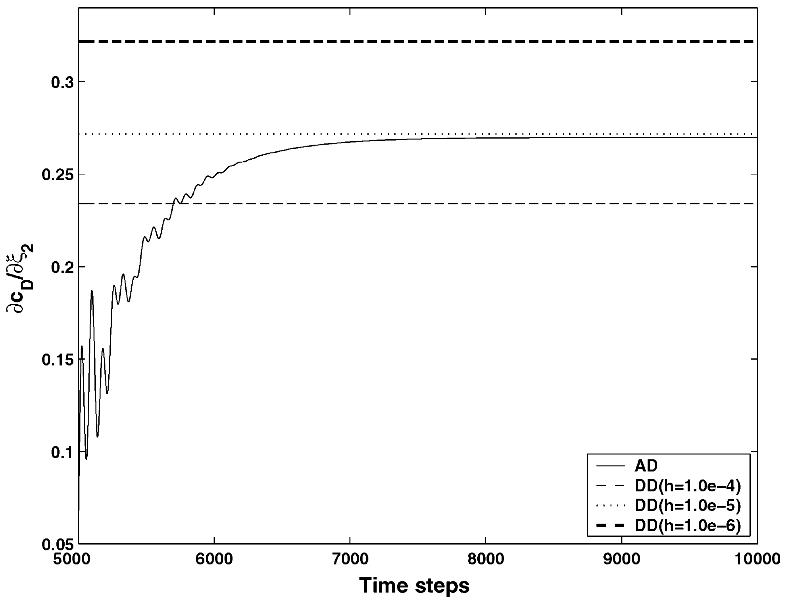


Fig. 4. Comparison between $\partial c_D/\partial \xi_{2\text{AD}}$ and $\partial c_D/\partial \xi_{2\text{DD}}$ for step sizes $h = 10^{-4}$, 10^{-5}, and 10^{-6}.

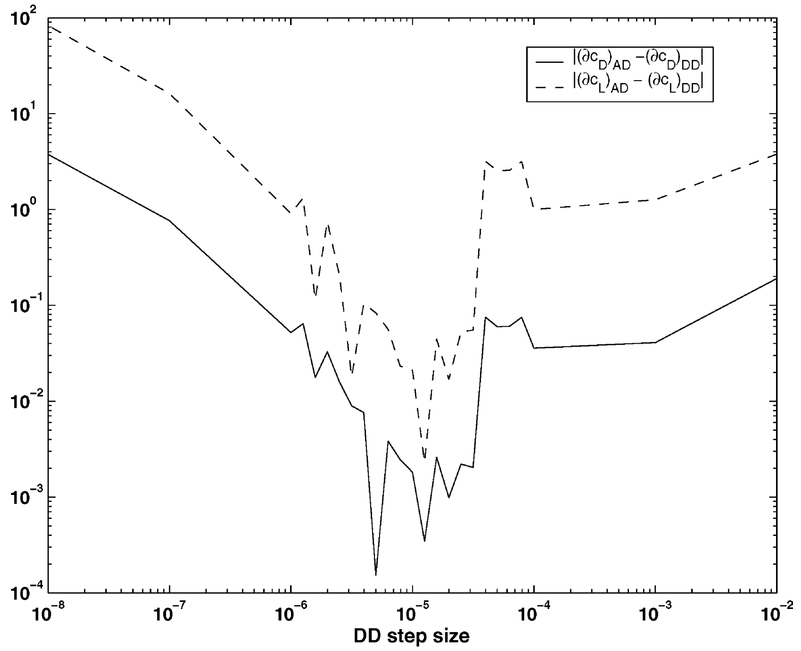


Fig. 5. Absolute difference between AD- and DD-computed derivatives $\partial c_D/\partial\xi_2 \approx 0.27$ and $\partial c_L/\partial\xi_2 \approx 6.92$.

derivative of the objective function, $\partial(c_L/c_D)/\partial\xi_2$, has only three correct digits. Second, even if a range of "appropriate steps sizes" is known (10^{-6} to 10^{-4} in our example), the quality of the DD approximations may vary widely within this range, even for very small variations of h. This fact further complicates finding an optimal step size, in particular since without the AD value we do not know which of the DD approximations is most accurate.

By contrast, computing the derivatives with AD does not require experimenting with step sizes and yields results that are free from truncation and cancellation errors.

In the present study, we are interested in the feasibility of applying AD to the computational chain rather than generating a performance-oriented AD code. Despite the lack of tuning the application to increase performance, the differentiated code is reasonably efficient. Since, compared to flow solver TFS, the storage requirements and execution times of the airfoil and grid generators are negligible, we report on the performance of the differentiated TFS program. Recall that the number of independent variables is eight throughout the computational chain and, therefore, a storage increase of a factor of 8.3 which was observed for the differentiated flow solver is acceptable. However, there is room for improvements with respect to the execution time as the factor by which the AD code is slower than the original TFS code is about 16. Note that forward divided differences for eight partial derivatives need nine evaluations of the original function. Current activities aimed at increasing the performance of the differentiated code include shared memory parallelization and aggressive compiler optimization techniques. For instance, by unrolling the loops over the derivative objects, the above-mentioned factor 16 can be reduced to about 14. Also, we are investigating to let the iteration initially run without the derivatives until a certain degree of convergence is obtained and then switch to the propagation of derivatives along with the original iteration [31]. In the framework of optimization, there is also the option to terminate the computation before the derivatives are fully converged. These approaches are explored in [3–5,8,32].

6. Conclusions and future directions

Accurate derivatives of mathematical functions given in the form of computer programs or sequences of computer programs can be obtained from automatic differentiation (AD). The idea behind automatic differentiation is to view a program execution as a composition of a—potentially large—number of elementary functions such as binary addition or multiplication. Then, the chain rule is applied to accumulate the overall derivatives from the known partial derivatives of the elementary functions. Sophisticated AD tools are capable of generating derivative code that computes the product of the Jacobian matrix and a user-specified seed matrix. The concept of seeding is important when derivatives of a sequence of different computer programs are to be computed because it enables the propagation of directional derivatives from one program to the following. In the forward mode of AD, the execution time of a differentiated program is roughly given by the product of the execution time of the original program and the number of independent variables, i.e., the scalar variables with respect to which derivatives are computed. Thus, by using an appropriate seeding, the execution time of an AD code of a sequence of programs is approximately given by the execution time of the sequence of the original programs and the number of independent variables of the first program. The crucial issue is that the execution time of the AD code is independent of the number of inputs of all the remaining programs.

In an application from aerodynamics, the derivatives of $c_\mathrm{L}/c_\mathrm{D}$ with respect to some design parameters determining the shape of a two-dimensional airfoil are of interest. Here, c_L and c_D denote the lift and drag coefficients, respectively, of the airfoil. The underlying mathematical function is defined by executing a MATLAB program followed by two Fortran 77 programs. The MATLAB program computes the shape of the airfoil from a given initial shape and the design parameters. In a second step, a Fortran 77 program is used to generate a multi-block structured grid around the given airfoil. In a third step, a Navier–Stokes solver computes the flow field including $c_\mathrm{L}/c_\mathrm{D}$. Automatic differentiation is applied to these three computer codes to obtain the desired derivatives. The numerical values of the AD-generated derivatives are compared to approximations of the derivatives based on divided differences. It is shown that divided differences heavily depend on an appropriate step size and that the best step size varies with the choice of the derivative whereas AD computes reliable values for any of the desired derivatives.

For the near future we plan to embed the differentiated computational chain into a hybrid optimization framework combining evolutionary and gradient-based techniques. The overall idea behind this hybrid approach is to achieve global and fast local convergence behavior. The next step involves the transition from two to three dimensions so that a parallel approach is crucial in order to keep the execution times within reasonable limits. Moreover, three-dimensional problems lead to a significant increase of the number of design parameters. Then, the use of the reverse mode of automatic differentiation is required because its time complexity is independent from the number of design parameters. Preliminary results with the reverse mode applied to an advanced CFD solver are reported in [32]. The long-term goal is to understand the wake flow field making it possible to reduce the dominating turbulences, for instance, by changing the position of the engines or the shape of the airfoil. A precise specification of an appropriate objective function is still an open problem.

Acknowledgments

The authors would like to thank Jakob Risch for his notable contribution during the initial phase of this project and Andre Vehreschild for providing the ADiMat tool. We would also like to thank the Aerodynamics Institute, RWTH Aachen University, for making available the source code of the flow solver TFS. In particular, Matthias Meinke and Ehab Fares deserve special recognition. We are grateful to the referees for several helpful comments leading to an improvement of the quality of the manuscript. This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB 401 “Modulation of flow and fluid–structure interaction at airplane wings”, RWTH Aachen University, Germany.

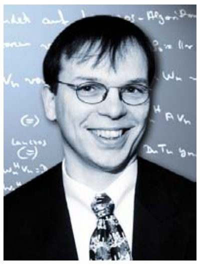
References

[1] L.B. Rall, Automatic Differentiation: Techniques and Applications, vol. 120 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1981.

[2] A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, Philadelphia, 2000.

[3] B. Mohammadi, O. Pironneau, Applied Shape Optimization for Fluids, Oxford University Press, Oxford, 2001.
[4] A. Carle, L.L. Green, C.H. Bischof, P.A. Newman, Applications of automatic differentiation in CFD, in: Proceedings of the 25th AIAA Fluid Dynamics Conference, Colorado Springs, CO, USA, June 20–23, AIAA Paper 94-2197, 1994.
[5] L.L. Green, P.A. Newman, K.J. Haigler, Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via automatic differentiation, J. Comput. Phys. 125 (2) (1996) 313–324.
[6] L. Green, P. Newman, K. Haigler, Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via automatic differentiation, in: Proceedings of the 11th AIAA Computational Fluid Dynamics Conference, Orlando, FL, USA, July 6–9, AIAA Paper 93-3321, 1993.
[7] C.H. Bischof, A. Mauer, W.T. Jones, J. Samareh, Experiences with automatic differentiation applied to a volume grid generation code, J. Aircraft 35 (4) (1998) 569–573.
[8] S.A. Forth, T.P. Evans, Aerofoil optimisation via AD of a multigrid cell-vertex Euler flow solver, in: G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (Eds.), Automatic Differentiation of Algorithms: From Simulation to Optimization, Springer, New York, 2002, Chapter 17, pp. 153–160.
[9] P. Aubert, N. Di Césaré, O. Pironneau, Automatic differentiation in C++ using expression templates and application to a flow control problem, Comput. Visual. Sci. 3 (2001) 197–208.
[10] C. Bischof, G. Corliss, L. Green, A. Griewank, K. Haigler, P. Newman, Automatic differentiation of advanced CFD codes for multidisciplinary design, J. Comput. Syst. Eng. 3 (6) (1992) 625–637.
[11] C. Bischof, L. Green, K. Haigler, T. Knauff, Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation, in: Proceedings of the Fifth AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, AIAA Paper 94-4261, American Institute of Aeronautics and Astronautics, 1994, pp. 73–84.
[12] A. Griewank, G. Corliss (Eds.), Automatic Differentiation of Algorithms, SIAM, Philadelphia, 1991.
[13] M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, 1996.
[14] G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (Eds.), Automatic Differentiation of Algorithms: From Simulation to Optimization, Springer, New York, 2002.
[15] C.H. Bischof, M.R. Haghighat, Hierarchical approaches to automatic differentiation, in: M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, 1996, pp. 83–94.
[16] A. Griewank, D. Juedes, J. Utke, ADOL-C, a package for the automatic differentiation of algorithms written in C/C++, ACM Trans. Math. Softw. 22 (2) (1996) 131–167.
[17] C. Bendtsen, O. Stauning, FADBAD, a flexible C++ package for automatic differentiation, Technical Report IMM-REP-1996-17, Technical University of Denmark, IMM, Department of Mathematical Modeling, Lyngby, 1996.
[18] D. Shiriaev, ADOL-F automatic differentiation of Fortran codes, in: M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, 1996, pp. 375–384.
[19] C. Bischof, A. Carle, P. Khademi, A. Mauer, ADIFOR 2.0: Automatic differentiation of Fortran 77 programs, IEEE Comput. Sci. Eng. 3 (3) (1996) 18–32.
[20] M. Fagan, A. Carle, Adifor 3.0 overview, Technical Report of CAAM-TR00-03, Rice University, Department of Computational and Applied Mathematics, 2000.
[21] R. Giering, T. Kaminski, Recipes for adjoint code construction, ACM Trans. Math. Softw. 24 (4) (1998) 437–474.
[22] N. Rostaing, S. Dalmas, A. Galligo, Automatic differentiation in Odyssée, Tellus 45A (5) (1993).
[23] The TAPENADE tutorial, http://www-sop.inria.fr/tropics/.
[24] C.H. Bischof, L. Roh, A. Mauer, ADIC—An extensible automatic differentiation tool for ANSI-C, Software Pract. Exper. 27 (12) (1997) 1427–1456.
[25] E. Fares, M. Meinke, W. Schröder, Numerical simulation of the interaction of flap side-edge vortices and engine jets, in: Proceedings of the 22nd International Congress of Aeronautical Sciences, Harrogate, UK, August 27–September 1, ICAS 0212, 2000.
[26] E. Fares, M. Meinke, W. Schröder, Numerical simulation of the interaction of wingtip vortices and engine jets in the near field, in: Proceedings of the 38th Aerospace Sciences Meeting and Exhibit, Reno, NV, USA, January 10–13, AIAA Paper 2000-2222, 2000.
[27] M.S. Liou, C.J. Steffen, A new flux splitting scheme, J. Comput. Phys. 107 (1993) 23–39.
[28] M.S. Liou, A sequel to AUSM: AUSM$^+$, J. Comput. Phys. 129 (1996) 164–182.
[29] T.F. Coleman, A. Verma, ADMIT-1: Automatic differentiation and MATLAB interface toolbox, ACM Trans. Math. Softw. 26 (1) (2000) 150–175.
[30] C.H. Bischof, H.M. Bücker, B. Lang, A. Rasch, A. Vehreschild, Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs, in: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, Montreal, Canada, October 1, IEEE Computer Society, Los Alamitos, CA, 2002, pp. 65–72.
[31] H.M. Bücker, A. Rasch, E. Slusanschi, C.H. Bischof, Delayed propagation of derivatives in a two-dimensional aircraft design optimization problem, in: D. Sénéchal (Ed.), Proceedings of the 17th Annual International Symposium on High Performance Computing Systems and Applications and OSCAR Symposium, Sherbrooke, Québec, Canada, May 11–14, NRC Research Press, Ottawa, 2003, pp. 123–126.
[32] A. Carle, M. Fagan, L.L. Green, Preliminary results from the application of automated adjoint code generation to CFL3D, in: Proceedings of the Seventh AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, USA, September 2–4, AIAA Paper 98-4807, 1998.

Christian Bischof is head of the Institute for Scientific Computing and of the Center for Computing and Communication of RWTH Aachen University. He received a PhD in computer science from Cornell University in 1988. His interests lie in semantic transformation, especially automatic differentiation, high-performance computing, virtual reality, and problem solving environments.



H. Martin Bücker received the Dipl.-Ing. degree in electrical engineering, the Dipl.-Inform. degree in computer science, and the PhD degree in electrical engineering from RWTH Aachen University in 1993, 1994, and 1997, respectively. He worked for the Central Institute for Applied Mathematics, Research Centre Jülich, from 1993 to 1998, participating in the design of parallel algorithms for sparse matrix problems. Since 1998 he has been with the Institute for Scientific Computing, RWTH Aachen University, where he received his Habilitation in 2003. His current areas of research include high-performance computing, numerical linear algebra, and automatic differentiation.

Bruno Lang is a professor in the Applied Computer Science Group, University of Wuppertal. He received his diplomas in Computer Science (1989) and Mathematics (1990) and his PhD (1991) from Technical University Karlsruhe and his Habilitation (1997) from the University of Wuppertal. He was a senior research assistant at the Institute for Scientific Computing, Aachen University, from 1998 to 2002. His current activities include teaching and several research projects touching on parallelism, verified computing, and automatic differentiation.

Arno Rasch received his Dipl.-Inform. in computer science from RWTH Aachen University in 2000. He is currently working as research assistant and PhD student at the Institute for Scientific Computing, RWTH Aachen University. His research interests include parallel computing, automatic differentiation, and engineering applications.

Emil-Ioan Slusanschi received his Dipl.-Ing. in computer science from University Politechnica of Bucharest at the Faculty of Automatic Control and Computers in 2000. He also received his Advanced Studies Diploma at the same University in 2001. He is currently working as research assistant and PhD student at the Institute for Scientific Computing, RWTH Aachen University. His research interests include parallel computing and parallel architectures, automatic differentiation, and engineering applications.