JOURNAL OF AIRCRAFT
Vol. 35, No. 4, July–August 1998

Experiences with Automatic Differentiation Applied to a Volume Grid Generation Code

Christian H. Bischof* and Andrew Mauer†
Argonne National Laboratory, Argonne, Illinois 60439
and
William T. Jones‡ and Jamshid Samareh§
Computer Sciences Corporation, Hampton, Virginia 23666

Automatic differentiation (AD) is a methodology for developing reliable sensitivity-enhanced versions of arbitrary computer programs with little human effort. As such, it can vastly accelerate the use of advanced simulation codes in a multidisciplinary design optimization context because the time for generating and verifying derivative codes is greatly reduced. The application of the recently developed automatic differentiation of C programs (ADIC) prototype tool for ANSI-C programs on the coordinate and sensitivity calculator for multidisciplinary design optimization multiblock three-dimensional volume grid generator are reported. The ADIC-generated code can easily be interfaced with Fortran derivative codes generated with the ADIFOR AD tool for Fortran 77 programs; thus providing efficient sensitivity-enhancement techniques for multilanguage, multidiscipline problems.

I. Introduction

MULTIDISCIPLINARY design optimization (MDO) is a methodology for the design of complex engineering systems and subsystems that coherently exploits the synergism of mutually interacting phenomena. Typically, the MDO process proceeds in an iterative fashion, with each MDO cycle including at least the generation of a numerical solution, determination of associated design sensitivities, and system optimization.

Of these parts, the generation of the numerical solution is the truly domain-dependent piece of this process, where in-depth knowledge of the problem at hand is used to develop the necessary simulation codes. For system optimization, on the other hand, a wide suite of existing optimization algorithms is readily available (see, e.g., Ref. 1).

An MDO problem of particular interest in the aerospace community is the design optimization of the high-speed civil transport. This problem, at the minimum, includes computational fluid dynamics (CFD) and, therefore, numerical grid generation as an integral part of the design process. The possible variation of the simulation method employed for modeling this problem leaves the computation of design sensitivities for the simulation codes in question. Here, we should keep the following issues in mind:

1) *Reliability*: The computed derivatives should be computed accurately.

2) *Computational cost*: The amount of memory and runtime required for the derivative code should be minimized.

3) *Scalability*: Whatever method we choose should be applicable to large codes.

4) *Human effort*: A user should not have to spend much of his or her time on developing computational procedures for computing derivatives.

Traditionally, hand-coding, finite difference (FD) approximations, and symbolic methods have been used for the computation of derivatives. However, these methods fall short with respect to the previously mentioned criteria. The main drawback of FD approximations is their numerical unpredictability as well as their computational cost. In contrast, the hand-coding and symbolic approach cannot be directly applied to large codes and they require considerable human effort. In addition, significant effort has to be expended whenever the analysis code is modified.

Recently, so-called *automatic differentiation* (AD) tools have emerged as a promising approach for computing derivatives. AD techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and cos (see, e.g., Refs. 2 and 3). By repeatedly applying the chain rule of derivative calculus to the composition of those elementary operations, one can compute, in a completely mechanical fashion, derivatives of f that are correct up to machine precision.[4]

In this paper we report on the application of the recently developed automatic differentiation of C papers (ADIC) prototype tool for ANSI-C programs on the coordinate and sensitivity calculator for multidisciplinary design optimization (CSCMDO) multiblock three-dimensional volume grid generator. CSCMDO is a general-purpose grid generator taylored specifically to MDO applications. The next section gives a brief overview of CSCMDO. ADIC provides AD capability for codes written in ANSI-C, and the philosophy and approach underlying ADIC are described in Sec. III. In Sec. IV we report on the results obtained with sensitivity-enhanced versions of CSCMDO generated with ADIC. Lastly, we summarize our results.

Received Dec. 6, 1995; presented as Paper 96-0716 at the AIAA 34th Aerospace Sciences Meeting, Reno, NV, Jan. 15–18, 1996; revision received Jan. 20, 1998; accepted for publication Feb. 3, 1998.

*Computer Scientist, Mathematics and Computer Science Division. E-mail: bischof@mcs.anl.gov.

†Student Research Associate, Mathematics and Computer Science Division; currently Graduate Student, Department of Mathematics, University of Illinois at Urbana–Champaign, Urbana, IL 61801. E-mail: mauer@math.uiuc.edu.

‡Senior Member of Technical Staff, NASA Lewis Research Center's GEOLAB, 3217 North Armistead Avenue. E-mail: w.t.jones@larc.nasa.gov. Member AIAA.

§Computer Scientist, NASA Lewis Research Center's GEOLAB; currently Research Scientist, Multidisciplinary Optimization Branch, 18c West Taylor Street, M/S 159, Hampton, VA 23681. E-mail: j.a.samareh@larc.nasa.gov. Senior Member AIAA.

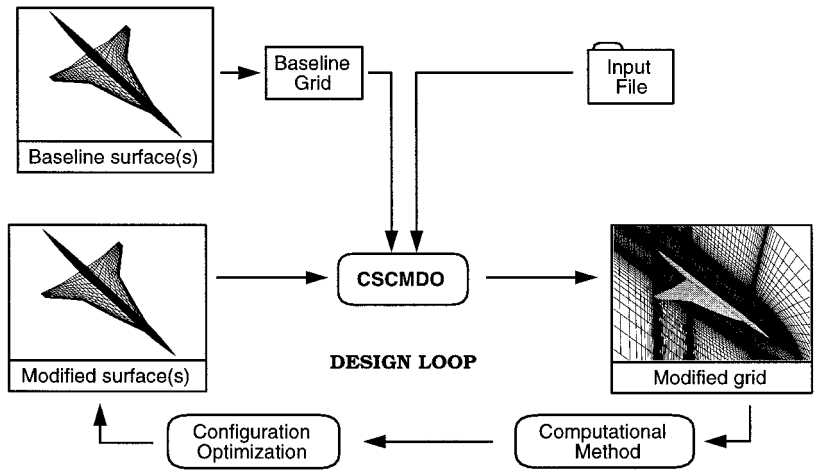

Fig. 1 Integration of CSCMDO into the design loop.

II. CSCMDO Multiblock Three-Dimensional Volume Grid Generator

CSCMDO is a general-purpose, multiblock, three-dimensional, structured volume grid generator with specialized features that are highly suitable for MDO-type grid modifications. The code is designed to execute in a batch environment with control provided via an ASCII user input file. The code is capable of modifying any of the six faces of a block to reflect changes in the optimized geometry as defined by an input surface(s). This section gives a brief overview of CSCMDO; a more detailed discussion can be found in Ref. 5.

With the code executing in a batch mode, it can be incorporated directly into the design loop as shown in Fig. 1. As mentioned earlier, the computational method at the very least contains a CFD analysis. Information input from outside of the loop is generated one time before the loop is initiated. This information includes the baseline geometry surface(s), baseline CFD volume grid, and the user input file. The design loop is then rendered self-sufficient requiring no further human intervention. CSCMDO operates within the design loop to provide automated volume grid-generation/modification for each design cycle.

The surface definition(s) is provided in the form of a structured mesh of discrete point data. The number and distribution of points defining the surface are not required to match those of the desired CFD grid. However, sufficient point resolution must be provided so as to adequately define all surface curvature.

The baseline volume grid may be generated using any structured grid-generation package. CSCMDO supports the file formats commonly used in the field of CFD. No restriction on grid topology is imposed. However, the topology of the volume grid must be consistent with the CFD analysis and capable of incorporating design-cycle geometry modifications. The changes represented by modified geometry surface(s) are assumed to be small, but in the event that geometry changes do violate the volume topology, grid quality checks provide return codes to the software controlling the loop for appropriate action.

Individual block faces may be modified independently using a variety of methods including direct injection of a modified surface, parametric updates to a modified surface, projection to a modified surface, and the deformations conforming to a modified surface. Volume modifications are accomplished for each block using algebraic reinitialization of the block interior, or by deformation of the original block interior based on changes defined on the six block faces.

III. ADIC Prototype Tool

Traditionally, two approaches to AD have been developed: the so-called forward and reverse modes. These modes are distinguished by how the chain rule is used to propagate derivatives through the computation. We briefly summarize the main points about these two approaches, a more detailed description can be found in Ref. 6 and the references therein.

Let us assume that we have a function f that maps an n-vector $\boldsymbol{x}$ into an m-vector $\boldsymbol{y}$. The forward mode propagates derivatives of intermediate variables with respect to the independent variables and follows the control flow of the original program. Exploiting the linearity of differentiation, the forward mode allows us to compute arbitrary linear combinations

$$J \cdot S \tag{1}$$

of columns of the Jacobian

$$J = \begin{bmatrix} \dfrac{\partial y(1)}{\partial x(1)} & \cdots & \dfrac{\partial y(1)}{\partial x(n)} \\ \vdots & & \vdots \\ \dfrac{\partial y(m)}{\partial x(1)} & \cdots & \dfrac{\partial y(m)}{\partial x(n)} \end{bmatrix} \tag{2}$$

For an $n \times p$ matrix S, the effort required is roughly $\mathcal{O}(p)$ times the runtime and memory of the original program. In particular, when S is a vector $\boldsymbol{s}$, we compute the directional derivative

$$J * s = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h * \boldsymbol{s}) - f(\boldsymbol{x})}{h} \tag{3}$$

In contrast, the so-called reverse mode of AD propagates derivatives of the final result with respect to an intermediate quantity, so-called adjoint quantities. To propagate adjoints, one must be able to reverse the flow of the program, and remember or recompute any intermediate value that nonlinearly impacts the final result. This may not be easily achieved for general programs, and these issues are further discussed in Ref. 6. In either case, AD computes derivatives accurate to machine precision[4] and is directly applicable to computer programs of arbitrary length containing branches, loops, and subroutines.

From a user's perspective, AD tools preferably should behave like black boxes, which, given the code describing the function to be differentiated, and an indication of which program variables correspond to the independent and dependent variables with respect to differentiation, generate an efficient sensitivity-augmented code for computing the desired derivatives without any need for human intervention.

Fundamentally, there are two approaches for augmenting a computer code with derivative computations. Languages like C++ or Fortran 90 support a language feature called *operator*

overloading, which allows the redefinition of the behavior of the elementary arithmetic operations and, hence, can be employed to attach, "under the rug," so to speak, derivative objects to original program variables, and apply the rules of differentiation one operation at a time. The ADOL-C tool[7] employs such an approach to compute derivatives of arbitrary order.

ADIC, in contrast, employs a source-transformation approach to directly rewriting the source code to compute first-order derivatives. This approach requires considerable compiler infrastructure, and ADIC employs part of the Sage++ (Ref. 8) source transformation infrastructure for C++ programs to transform ANSI-C programs. With minor restrictions, the current ADIC prototype accepts arbitrary ANSI-C programs and can handle, for example, subroutines, dynamic memory allocation, and pointers. The code generated is a portable ANSI-C code that can easily be modified to, say, print out sensitivities. The features and limitations of ADIC are discussed in detail in Ref. 9, here we briefly summarize the main points.

ADIC, like ADIFOR, employs a hybrid forward-/reverse-mode approach to generating derivatives. For each assignment statement, the reverse mode is used to generate code that computes the partial derivatives of the result with respect to the variables on the right-hand side and then the forward mode is employed to propagate overall derivatives. Hence, ADIC and ADIFOR provide the directional derivative computation possibilities [see Eq. (1)] associated with the forward mode of automatic differentiation. As a result, derivatives generated, for example, by a Fortran code differentiated with ADIFOR, can easily be used as seed matrix for an ADIC-augmented-C code, whose derivatives, in turn, can again easily be ingested by a Fortran code. As seen in the next section, this is of vital importance in the MDO context.

ADIC can also transparently exploit sparsity in derivative computations through the use of the SparsLinC library,[6,9] which, as a byproduct of the computation, will automatically compute the sparsity pattern of large sparse Jacobians.

Fig. 2 Structure of CFD design problem.

IV. Experimental Results

To improve the aerodynamic performance of the high-speed civil transport, we embed the system shown in Fig. 2 in an optimization context. The rapid airplane parametric input design (RAPID) code[10] is written in Fortran 77 and, given geometric design variables (gdv), for example, camber, produces a aircraft surface grid (sfg). From this output, CSCMDO builds a three-dimensional volume grid (vlg). TLNS3D, a three-dimensional Navier–Stokes solver for turbulent flow[11,12] then uses the geometry information as well as the stream parameters (strm) to compute the flow (flw) over the aircraft and from there, measures of performance such as lift or drag. In the MDO design context we then need (∂flw/∂gdv) at every pass through the design cycle.

We mention several issues that are important to keep in mind when approaching this problem:

1) Computationally, this process is dominated by the CFD solver. The runtime of the grid generators is almost insignificant compared to that of TLNS3D.

Table 1 Timing results for RS/6000

	Number of design variables						
	1	2	3	4	5	6	7
Best-case FD	39	59	79	99	118	138	158
CSCMDO.AD (1)	130	174	223	281	324	526	560
CSCMDO.AD (2)	69	84	103	119	144	192	212

[a]Times in user seconds.

Table 2 Timing results for Sun-4

	Number of design variables						
	1	2	3	4	5	6	7
Best-case FD	57	86	115	144	172	201	230
CSCMDO.AS (1)	149	205	278	351	465	546	637
CSCMDO.AD (2)	83	104	131	160	184	213	242

[a]Times in user seconds.

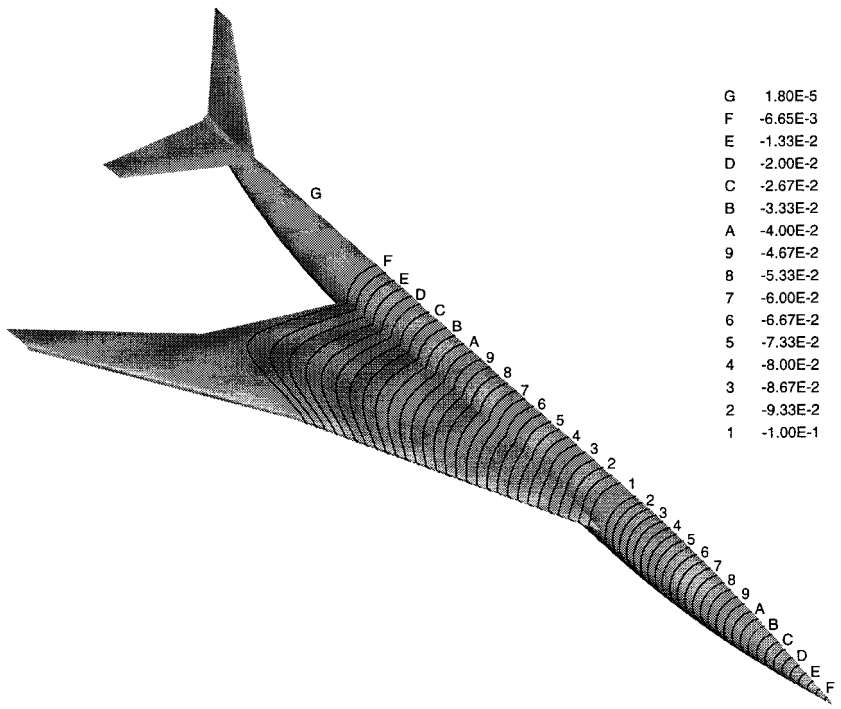

Fig. 3 Sensitivity on the surface of CFD grid X coordinate with respect to wing root chord.

2) Accurate sensitivities are required for all modules, and in particular for the grid generators, as errors in their derivatives would contaminate derivatives of TLNS3D.

3) The number of design objectives (flw) is small, whereas the number of surface grid points (sfg) and certainly the number of volume grid points (vlg) usually is very large.

4) The codes employed continue to be developed and refined.

In our experiments, we only concerned ourselves with computing the derivatives (∂vlg/∂sfg), employing the RAPID and CSCMDO codes. The ADIFOR tool[6,13] was applied to RAPID and a file containing (∂sfg/∂gdv) was written by the sensitivity-enhanced version of RAPID. We also mention that ADIFOR has also been successfully applied to the single-block version of TLNS3D, and the results are summarized in Ref. 14.

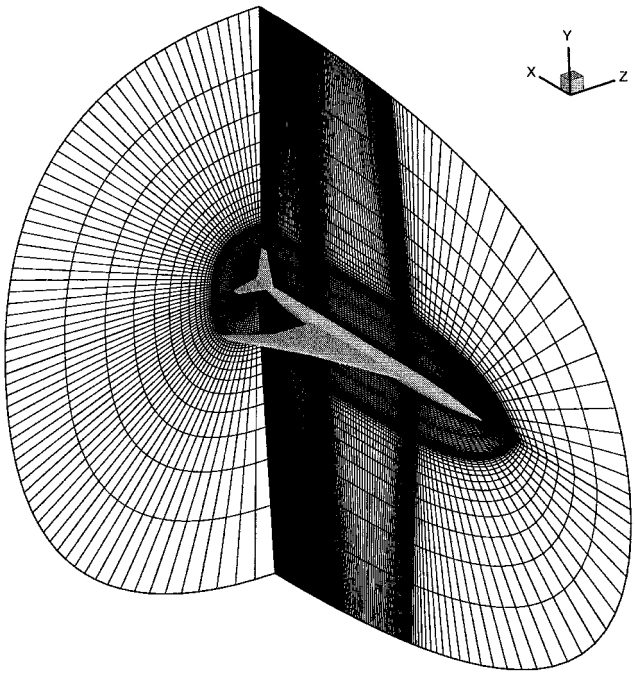

Fig. 4 CFD volume grid produced by CSCMDO.AD.

The ADIC prototype was applied to CSCMDO. The fact that the ADIC interface allows for derivative chaining is essential in this context. That is, instead of computing (∂vlg/∂sfg), which would be a huge (albeit sparse) matrix, which then would be multiplied with (∂sfg/∂gdv) to obtain (∂vlg/∂gdv), we can *directly* compute (∂vlg/∂gdv) by using (∂sfg/∂gdv) as the derivative seed matrix associated with inputs corresponding to the surface grid (sfg). As a result, the complexity of CSCMDO.AD, the ADIC-generated derivative code for CSCMDO, depends on the number of geometric design variables (gdv), *not* on the number of surface grid points (sfg).

Varying the number of geometric design variables from 1 to 7, we compare the runtime performance of CSCMDO.AD with that of a one-sided FD approximation on a Sun Sparcstation 5 and IBM RS/6000 platform. Employing version 2.5.8 of the GNU C compiler with the "-02 -ffast-math" compiler flags, we obtain the performance shown in Tables 1 and 2.

To generate a grid, CSCMDO can be thought of as using a two-step approach. First, it generates a grid, secondly, it validates the quality of the generated grid. In the line labeled "CSCMDO.AD (1)," we differentiated through the entire CSCMDO code, including the validation part. The line labeled "CSCMDO.AD (2)" refers to a somewhat more judicious use of CSCMDO, where derivatives are only propagated through the grid-generation part and not through the validation part. Note, however, that because ADIC intersperses derivatives and original program variables (see Ref. 9 for details), the validation part also had to be modified by ADIC to have data structures compatible with the sensitivity-enhanced grid generation part, but it did not include code for propagating derivatives. The line labeled "Best-case FD" lists the time required for a one-sided finite difference (FD) approximation of the derivatives, under the assumption that the correct step size was

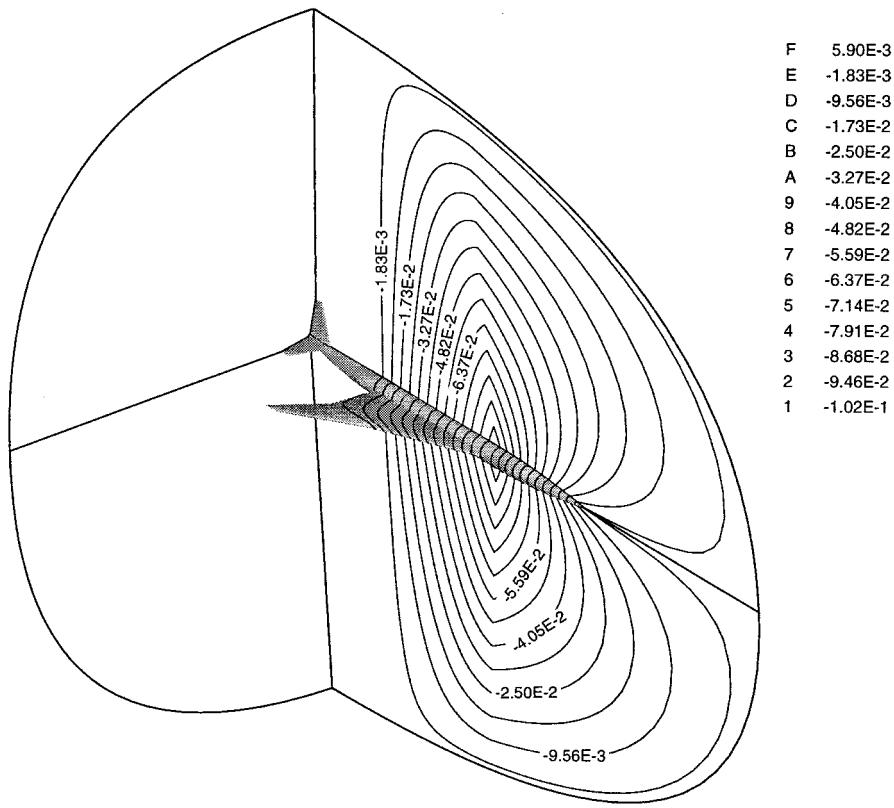

Fig. 5 Sensitivity in the field of CFD grid X coordinate with respect to wing root chord.

chosen. This is an optimistic assumption, because, typically, several perturbation sizes had to be tried before a good match of the derivative approximations with FD along with the values obtained by CSCMDO.AD was obtained.

We see that the code generated by ADIC when applied to all of CSCMDO is considerably slower than a ''best-case'' FD approximation of derivatives. On the other hand, if we do avoid the (useless) derivative computation in the verification stage, CSCMDO.AD exhibits runtimes that are close to that of a best-case FD approximation. Note that we cannot avoid the verification stage in FD approximations, as the new volume grid arising from a parameter perturbation must be verified. We also mention that ADIC is still in the prototype stage and further improvements will narrow this gap. In either case, though, these runtimes are dwarfed by the runtime of a CFD flow code such as TLNS3D. However, the accuracy of the grid sensitivities is essential for the accuracy of the overall design sensitivities.

When examining our results, considering the sensitivity of the CFD volume grid X coordinate with respect to a change in wing root chord, we obtain the results shown in Figs. 3–5. Note that our RAPID output assumes that the wing trailing edge remains fixed with respect to the fuselage. Therefore, a change in the wing root chord should only be propagated forward of the wing trailing-edge/fuselage intersection.

The volume grid shown consists of two blocks and over 525,000 grid points. Contour lines represent the sensitivity of the volume grid X component with respect to a change in the root chord. The expected symmetry can be noted in Figs. 4 and 5. The grid X component can be seen to be the most sensitive to changes in the wing root chord around the leading edge of the wing/fuselage intersection. As mentioned earlier, the surface grid points (sfg) and (∂sfg/∂gdv) were obtained from the sensitivity-enhanced version of RAPID. These data were used by CSCMDO.AD to produce the volume grid points (vlg) and the sensitivities (∂vlg/∂gdv) shown in Figs. 3–5.

V. Conclusions

In this paper we reported on the application of the ADIC automatic differentiation tool for ANSI-C programs on the CSCMDO multiblock three-dimensional volume grid generator. CSCMDO is a general-purpose grid generator tailored specifically for MDO applications. ADIC is a prototype tool for the automatic sensitivity enhancement of codes written in ANSI-C. In our experience, ADIC allowed us to obtain accurate derivatives with a minimum of human effort. Without ADIC we would have been forced to spend considerable effort developing a sensitivity-enhanced version by hand. Instead, this effort was more profitably spent improving the features of CSCMDO. Together with the ADIFOR tool for Fortran 77 programs, ADIC provides efficient support for multilanguage, multidisciplinary design optimization where codes written in Fortran and C are embedded in the design loop.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Aerospace Agency under Purchase Order L25935D. We thank Larry Green of the Multidisciplinary Optimization Branch of NASA Langley Research Center for providing us with sensitivities from the RAPID code, and Brad Homann of the Mathematics and Computer Science Division of Argonne National Laboratory for performing the CSCMDO benchmark runs. We are also indebted to Tom Zang of the Multidisciplinary Optimization Branch of NASA Langley Research Center for initiating this collaboration, and to Dennis Gannon and his group at Indiana University for their support with Sage++.

References

[1]Moré, J. J., and Wright, S. J., *Optimization Software Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1993.

[2]Griewank, A., ''On Automatic Differentiation,'' *Mathematical Programming: Recent Developments and Applications*, Kluwer, Dordrecht, Amsterdam, 1989, pp. 83–108.

[3]Rall, L. B., ''Automatic Differentiation: Techniques and Applications,'' *Lecture Notes in Computer Science*, Vol. 120, Springer–Verlag, Berlin, 1981.

[4]Griewank, A., and Reese, S., ''On the Calculation of Jacobian Matrices by the Markowitz Rule,'' *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, edited by Andreas Griewank and George F. Corliss, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991, pp. 126–135.

[5]Jones, W. T., and Samareh-Abolhassani, J., ''A Grid Generation System for Multidisciplinary Design Optimization,'' *Proceedings of the AIAA 12th Computational Fluid Dynamics Conference*, AIAA, Washington, DC, 1995, pp. 474–482.

[6]Bischof, C., Carle, A., Khademi, P., and Mauer, A., ''ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs.'' *IEEE Computational Science and Engineering*, Vol. 3, No. 3, 1996, pp. 18–32.

[7]Griewank, A., Juedes, D., and Utke, J., ''ADOL-C, a Package for the Automatic Differentiation of Algorithms Written in C/C++,'' *ACM Transactions on Mathematical Software*, Vol. 22, No. 2, 1996, pp. 131–167.

[8]Bodin, F., Beckman, P., Gannon, D., Goutwals, J., Narayana, S., Srinivas, S., and Winnicka, B., ''Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools,'' *Proceedings of the 2nd Annual Object-Oriented Numerics Conference*. Inst. of Electrical and Electronics Engineering, New York, 1994.

[9]Bischof, C., Roh, L., and Mauer, A., ''ADIC—An Extensible Automatic Differentiation Tool for ANSI-C,'' *Software—Practice and Experience*, Vol. 27, No. 12, 1997, pp. 1427–1456.

[10]Smith, R. E., Bloor, M. G. I., Wilson, M., and Thomas, A. M., ''Rapid Airplane Parametric Input Design (RAPID),'' *Proceedings of the AIAA 12th Computational Fluid Dynamics Conference*, AIAA, Washington, DC, 1995.

[11]Vatsa, V. N., and Wedan, B. W., ''Development of a Multigrid Code for 3-D Navier-Stokes Equations and Its Application to a Grid-Refinement Study,'' *Computers and Fluids Journal*, Vol. 18, No. 4, 1990, pp. 391–403.

[12]Vatsa, V. N., Sanetrik, M. D., and Parlette, E. B., ''Development of a Flexible and Efficient Multigrid-Based Multiblock Flow Solver,'' *Proceedings of the AIAA 31st Aerospace Sciences Meeting*, AIAA, Washington, DC, 1993.

[13]Bischof, C., Carle, A., Corliss, G., Griewank, A., and Hovland, P., ''ADIFOR: Generating Derivative Codes from Fortran Programs,'' *Scientific Programming*, Vol. 1, No. 1, 1992, pp. 11–29.

[14]Carle, A., Green, L., Bischof, C., and Newman, P., ''Applications of Automatic Differentiation in CFD,'' *Proceedings of the 25th AIAA Fluid Dynamics Conference*, AIAA Paper 94-2197, Washington, DC, 1994.