Developing a Derivative-Enhanced Object-Oriented Toolkit for Scientific Computations *

Paul Hovland† Boyana Norris† Lucas Roh† Barry Smith†

Abstract

We describe the development of a differentiated version of PETSc, an object-oriented toolkit for the parallel solution of scientific problems modeled by partial differential equations. Traditionally, automatic differentiation tools are applied to scientific applications to produce derivative-augmented code, which can then be used for sensitivity analysis, optimization, or parameter estimation. Scientific toolkits play an increasingly important role in developing large-scale scientific applications. By differentiating PETSc, we provide accurate derivative computations in applications implemented using the toolkit. In addition to using automatic differentiation to generate a derivative enhanced version of PETSc, we exploit the component-based organization of the toolkit, applying high-level mathematical insight to increase the accuracy and efficiency of derivative computations.

1 Introduction

In complex computational models of physical phenomena, it is often necessary or desirable to compute the derivatives of a function $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}^m$, where f is defined by a computer program with n inputs and m outputs. We call x the *independent variable* and y the *dependent variable* and denote the Jacobian matrix $f'(x)$ by J. There are many ways in which the desired derivatives can be obtained. After a short summary of some traditional approaches to computing derivatives, we discuss the differentiation of PETSc, an object-oriented toolkit for building scientific applications involving the solution of partial differential equations.

One standard approach is to use *divided differences* (DD) to approximate the Jacobian matrix. The i^{th} column of J is approximated by using first-order accurate forward differences,

$$\frac{f(x + h_i e_i) - f(x)}{h_i},$$

where the step size h_i is a suitably chosen parameter and e_i is the i^{th} unit vector. The DD approach has the advantage that the function is needed only as a black box. The accuracy

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave, Argonne, IL 60439, [hovland,norris,bsmith]@mcs.anl.gov.

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

of approximations depends on the step size h_i and may be difficult to assess. A small step size is needed to suitably approximate the derivatives, but this can lead to cancellation errors and loss of accuracy.

Another approach to obtaining the Jacobian matrix of a well-defined function is to use a symbolic manipulation package such as Mathematica. Because of excessive resource requirements, this approach is predominantly applicable to small problems (e.g., fewer than 50 lines of code).

Automatic differentiation is a powerful tool for augmenting arbitrary codes with accurate derivative computations. When application codes are written "from scratch," automatic differentiation tools can be applied directly by the application programmer. Scientific codes developed using numerical libraries or toolkits offer additional advantages: not only can automatic differentiation tools be applied to the library code, but the library programmers can use their own unique knowledge of the underlying algorithms to provide more accurate derivative information faster.

The remainder of this section introduces the concepts of automatic and computational differentiation. Section 2 addresses the consequences of differentiating approximate methods and describes our approach to improving the efficiency of derivative computations, and application areas in which AD can be successfully employed. Section 3 presents some experimental results and future directions. Section 4 summarizes our conclusions.

1.1 Automatic Differentiation

Automatic differentiation (AD) offers a black-box mechanism for accurately computing the derivatives of arbitrary functions. Virtually any computer program written in Fortran, C, or C++ can be automatically augmented to evaluate the derivative of f using AD, a chain-rule-based technique for evaluating the derivatives of functions defined by algorithms [4, 5, 6, 7, 10]. The code produced by AD tools computes both the function value y and the derivatives of some of the outputs y with respect to some of the inputs x.

The basic underlying principle of AD is that any computation, no matter how complex, can be viewed as a limited set of arithmetic operations and elementary function calls. By applying the chain rule to the composition of elementary operations, AD produces augmented code computing the derivatives of f exactly (to machine precision). AD tools can be used in a black-box fashion for differentiating large scientific applications. Derivatives computed using AD can be used by computational scientists for sensitivity analysis of computational models, that is, the sensitivity of a model's output to perturbations in its physical and computational parameters. AD can also be used to help generate derivatives needed in design optimization, parameter identification, and the solution of stiff differential and algebraic equations.

The ADIC (Automatic Differentiation in C) tool [6, 7] provides automatic differentiation of programs written in C. Given a collection of C subroutines and an indication of which program parameters correspond to independent and dependent variables, ADIC produces C code that allows the computation of derivatives of the dependent variables with respect to the independent variables.

1.2 Computational Differentiation

We use the term *computational differentiation* (CD) to designate the approach that couples AD technology with high-level knowledge about the code being differentiated. In general, AD tools operate on the level of simple arithmetic operations, applying the chain rule in

order to compute the derivatives of a given code. In some cases, we can reduce the memory requirements and increase the performance and accuracy of derivative computations by analytically deriving and hand-coding the derivatives of frequently used computational components (e.g., solving of a system of linear equations).

Using AD, one can easily produce derivative code that computes gradients with no round-off error. However, when approximate (e.g., iterative) methods are used, AD computes the derivatives of the algorithm implemented rather than those of the model function; thus, the resulting values may depend on the particular algorithm. This dependence may be an undesired side-effect of differentiating a *program* instead of the mathematical function. The use of CD in high-level computational components can minimize this effect. In the following subsection, we consider the computational differentiation of linear equation solvers.

1.3 PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is an object-oriented toolkit for the parallel solution of scientific problems modeled by partial differential equations [1, 2, 3]. PETSc includes a suite of parallel linear and nonlinear equation solvers and unconstrained minimization modules that may be used in application codes written in Fortran, C, and C++.

PETSc is organized hierarchically, allowing users to employ the level of abstraction that is most suitable for a particular problem. This decreases the development cycle and increases the software's maintainability. The actual implementations of most data structures and algorithms can be specified at run time, allowing for great flexibility in choosing the best combination. The object-oriented organization of PETSc makes applying an AD tool straightforward.

1.4 Combining AD Tools and Scientific Toolkits

AD tools provide a mechanism for the simplified generation of derivative-enhanced versions of scientific toolkits, which in turn simplify the differentiation of applications implemented using these toolkits. With minimum effort, one can perform sensitivity analysis of existing scientific codes or build extensions utilizing the derivatives in some outer computation (e.g., an optimization toolkit).

Augmenting PETSc with derivative code is a natural extension to the toolkit's functionality. Large scientific models that have been implemented using the toolkit can be analyzed and verified in a straightforward fashion. AD-enabled large-scale multivariate sensitivity analysis can help identify model deficiencies and possible improvements. A differentiated version of PETSc can also be useful in design optimization of complex systems, where the main goal is to select optimal values for critical model parameters in order to attain some specified design objectives.

PETSc provides standardized interfaces to each of its major components. This enables us to fully automate the process of generating a derivative-enhanced version of the toolkit, which is essential for keeping up with new releases.

2 Taking Advantage of Common Algorithmic Structures

Many computational methods for solving partial differential equations involve the solution of sparse linear systems of equations. The linear solver is usually inside the application being differentiated. The Scalable Linear Equation Solver (SLES) component of PETSc

provides a uniform interface to a variety of methods for solving large sparse linear systems in parallel. These methods find the solution of the system

$$A(s) \cdot x(s) = b(s), \tag{1}$$

where s is a m-dimensional parameter, $A(s)$ is a matrix, and $b(s)$ is a vector. This linear system of equations represents an implicit definition of a function $x(s)$. We wish to find the derivatives dx/ds (designated by ∇x) of the solution of (1). A detailed analysis of differentiating parametric linear systems can be found in [8]. For brevity, we will write equation (1) as $Ax = b$.

The combination of a Krylov subspace method and a preconditioner is at the core of most modern numerical codes for the iterative solution of linear systems. This is the approach implemented in the linear equation solver (SLES) component of PETSc. SLES defines a standard interface to solving a linear system using preconditioned iterative methods. At present, PETSc provides about ten different preconditioners and ten Krylov subspace methods.

The derivatives produced by augmenting iterative algorithms do not necessary converge at the same rate as the function being differentiated [8, 9, 11]. Thus, when the stopping criterion for the original iteration is satisfied, the derivative code may not have reached the same accuracy as the solution or may have converged to a satisfactory value in fewer iterations than the solution. In either case, the programmer has little control over the accuracy or performance of derivative computations.

To compute ∇x, we can differentiate (1), producing

$$\begin{aligned} A\nabla x + \nabla A\, x &= \nabla b \\ A\nabla x &= \nabla b - \nabla A\, x \qquad (2) \\ \nabla x &= A^{-1}(\nabla b - \nabla A\, x). \qquad (3) \end{aligned}$$

While logically the Jacobian is given by (3), in practice we obtain it by solving the linear system (2). The above derivation applies to the most general case, in which both A and b are dependent on the independent variable s.

In our implementation, we consider four cases depending on the relationship between A, b, and the user-specified independent variables s. The simplest case is when b is the independent variable, and A is not a function of b, that is, $\nabla b = I$ and $\nabla A = 0$. In this case, computing ∇x is reduced to solving the linear system $A\nabla x = I$. When $\nabla b \neq I$ and $\nabla A = 0$, we solve the system $A\nabla x = \nabla b$. Similarly, we provide implementations for the remaining two cases. In each of the four cases, the major part of computing ∇x involves solving a linear system with multiple right-hand sides. We use the SLES package, taking advantage of the run-time flexibility in selecting the preconditioner, solver, and desired accuracy at run time.

By considering each of these cases separately, we avoid overallocation of resources, and ensure that any potential parallelism can be exploited. For example, solving (2) requires that we first solve the system $Ax = b$ before we can compute ∇x by solving the linear system with multiple right-hand sides. When A and b are not parameterized by s, we can simultaneously solve for x and ∇x.

In addition to providing efficient implementation of the derivative computation, we provide a simple polymorphic interface for computing ∇x. The user need not keep track of the dependencies between A, b, and x in order to use our CD methods; the appropriate method is automatically selected at run time.

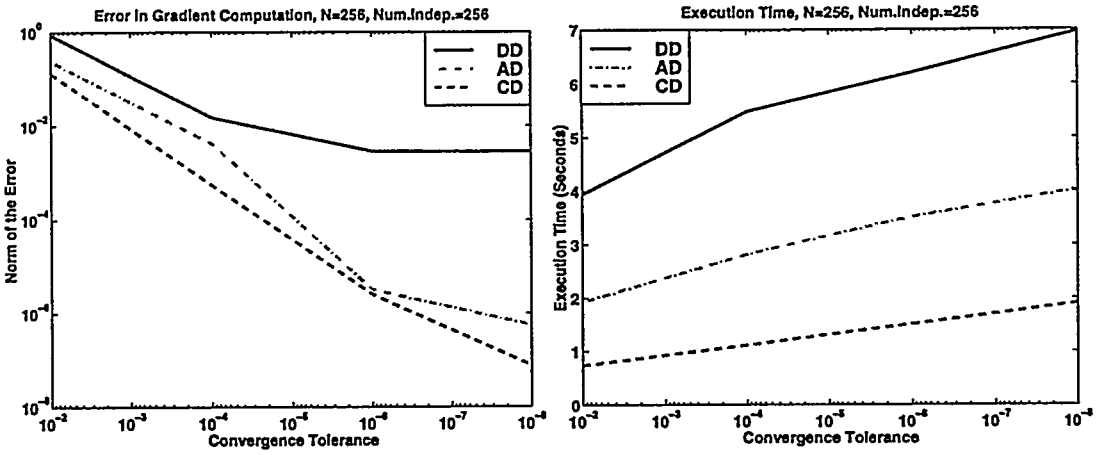

FIG. 1. : Gradient error and execution time with varying convergence tolerances.

3 Experimental Results

ADIC has been applied successfully to the uniprocessor version of PETSc, producing a derivative-augmented code for all components of the toolkit. In addition to the automatically generated code, we have implemented the CD methods discussed in Section 2.

Because of the modular design and implementation of PETSc, relatively few changes in the code were necessary to produce a working differentiated version of the toolkit. The dynamic polymorphism in PETSc enables the simultaneous use of the undifferentiated and derivative-enhanced versions in user applications. In general, AD can be applied to any modular software package, extending its functionality significantly at a reduced programming effort.

We have tested the differentiated version of PETSc, and in particular its SLES component, with an example involving the solution of a linear system of equations $A\,x = b$ where A is the 256×256 matrix corresponding to a five-point stencil discretization of a 16×16 computational domain. In all plots, DD designates divided difference approximation, AD designates black-box automatic differentiation, and CD stands for computational differentiation using hand-coded derivatives of common algorithmic structures. In all of the experiments, we have used a GMRES solver in combination with an incomplete LU factorization preconditioner.

Figure 1 contains the accuracy and performance results for various convergence tolerances. The termination condition of the Krylov subspace methods is based on the relative decrease of the l_2-norm of the residual and the convergence tolerance value, which is plotted along the x-axis. The y-axis of the accuracy plot is the l_2-norm of the matrix representing the difference between the derivatives produced by the various approaches and the actual solution, $\nabla x = A^{-1}b$, which we compute separately up to machine precision for verification purposes. For the DD and AD approaches the convergence tolerance refers to the convergence of x, while in the CD approach it refers to the convergence of ∇x. In this example, the CD approach exhibits significant performance improvement over DD and AD.

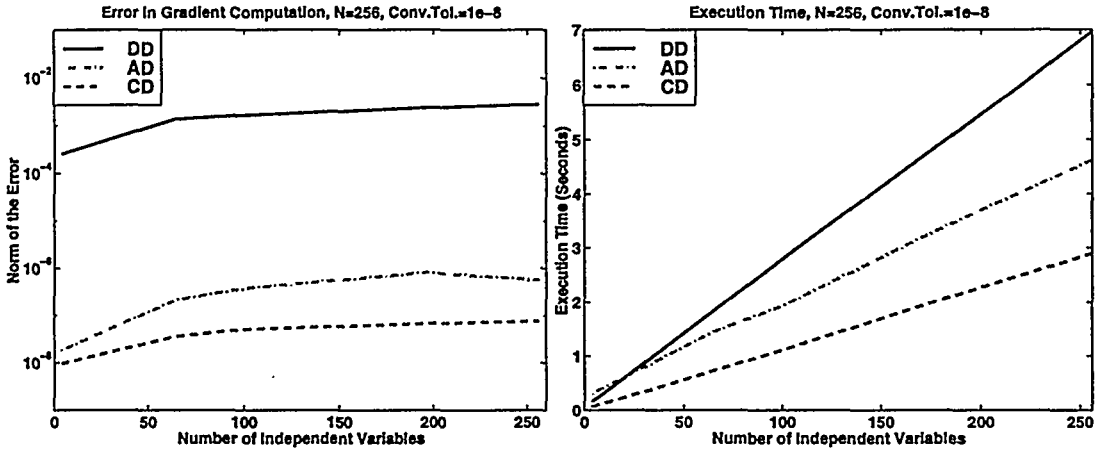

FIG. 2. : Gradient error and execution time with varying number of independent variables.

Figure 2 illustrates the accuracy and performance results for various numbers of independent variables and a fixed convergence tolerance of 10^{-8}. Again, the convergence tolerance refers to the computation of x in the case of DD and AD and to ∇x in the case of CD. In other words, we set the convergence criterion for the solution to 10^{-8} for all tests in this example, which does not imply that the derivative converges to the same accuracy. In fact, it has been shown that automatically generated derivatives of iterative solvers may often converge more slowly than the solution [11]. The accuracy of the resulting gradient is shown in the first plot. For larger numbers of independent variables, CD produces more accurate results than AD, and both AD and CD are several orders of magnitude more accurate than the DD approach.

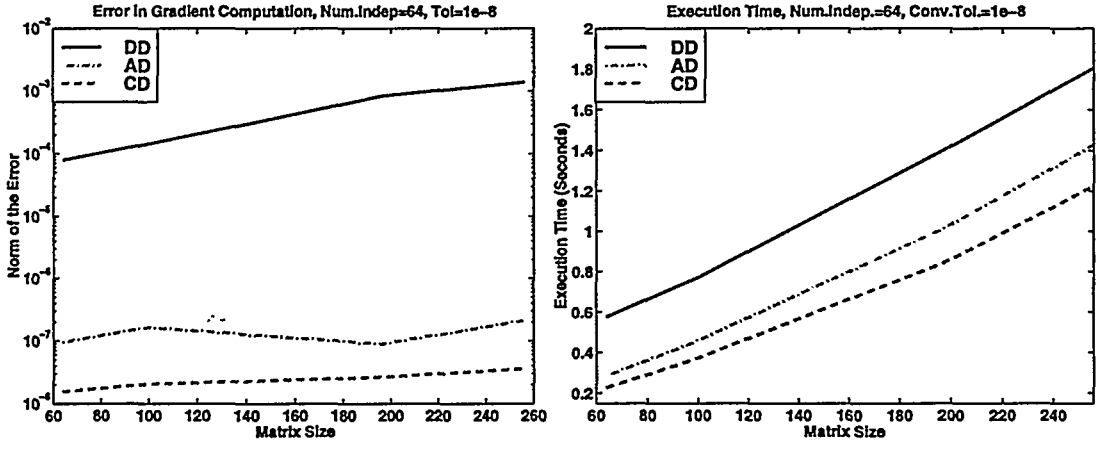

FIG. 3. : Gradient error and execution time with varying problem sizes.

Figure 3 shows the accuracy and timing results for varying problem sizes. The dimension of the square grid varies from 8 to 16, with corresponding matrix sizes indicated on the x axis. As in our previous experiments, we observe that the AD and CD methods for computing the derivatives result in more accurate results than the traditional DD approach. As described in Section 2, the CD approach involves the solution of a linear system with multiple right-hand sides. In our implementation we invoke an iterative solver for each right-hand side. An implementation that takes advantage of the multiple right-hand-sides would improve performance significantly.

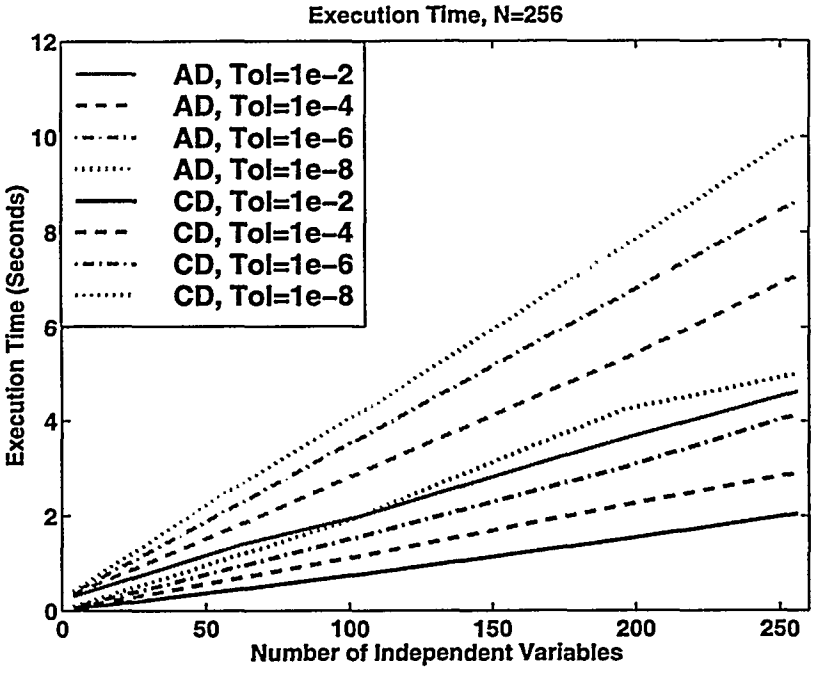

FIG. 4. : Performance comparison between AD and CD.

Figure 4 contains a summary of the execution times obtained for varying numbers of independent variables and convergence tolerances. As the convergence tolerance becomes tighter, the execution time for the AD approach increases more rapidly than CD as the number of independent variables grows. A large number of independent variables means more time spent performing a single iteration when using the AD approach, whereas for CD the time for a single iteration is the same, but the number of linear systems solved increases.

Figure 5 shows the performance of the three methods for computing ∇x to an accuracy of approximately 10^{-4}. Overall, we observe that for a fixed convergence tolerance with respect to ∇x, CD consistently outperforms AD and DD for various numbers of independent variables. For the problem sizes in our experiments, CD is clearly the best method for obtaining mathematically-meaningful derivatives of the solution to the linear system. Sometimes, it may be desirable to compute the derivatives of the *algorithm* itself, for example, when evaluating the sensitivity of the model to perturbations in its input parameters. In that case, CD-produced derivatives would be less meaningful than derivatives obtained with AD tools.

4 Conclusions

Augmenting PETSc with derivative computations using automatic differentiation greatly increases the functionality of the toolkit with minimum programming effort. While the derivatives produced using AD are often more accurate, the performance of the augmented code is somewhat worse than that of the original computation. Nevertheless, computing derivatives using AD is often faster than obtaining them with less accurate methods, such as divided differences.

We can draw several conclusions from our experiences:

- Toolkits providing good data and algorithm encapsulation allow computational

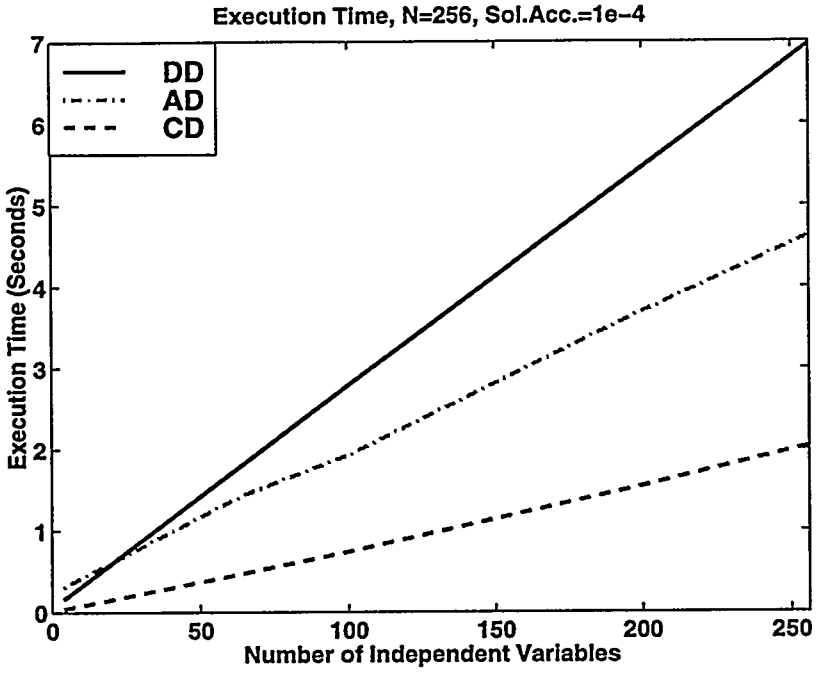

FIG. 5. : Execution time for achieving 10^{-4} accuracy in the gradient.

differentiation techniques to be incorporated under existing interfaces.

- Computational differentiation is usually faster and more accurate than both automatic differentiation and divided differences.
- Automatic differentiation can be used on an entire toolkit; subsequently, the computational differentiation approach can be applied to important common algorithmic structures.
- The simultaneous use of the toolkit and its derivative-enhanced version is possible.
- Toolkits maintain consistent interfaces for both non-derivative and derivative computations, while the underlying implementations may change. The applications using the toolkit need not be aware of such changes.

While high-level knowledge about the algorithms used in a toolkit can be used to utilize the faster and more mathematically meaningful CD approach to obtaining derivatives, sometimes we wish to analyze the algorithms themselves. In that case, AD-generated derivatives can be used. By augmenting PETSc with both types of computations, we allow the user to specify at run time the desired method for computing the derivatives. This strategy results in great flexibility, allowing the use of PETSc in a straightforward fashion for sensitivity analysis, design optimization, parameter identification, and other computations that need derivatives.

References

[1] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[2] ——, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[3] ——, *PETSc home page.* http://www.mcs.anl.gov/petsc, 1998.

[4] M. Berz, C. Bischof, G. Corliss, and A. Griewank, *Computational Differentiation: Techniques, Applications, and Tools*, SIAM, Philadelphia, 1996.

[5] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *ADIFOR 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Computational Science & Engineering, 3 (1996), pp. 18–32.

[6] C. Bischof and L. Roh, *ADIC user guide*, Technical Memorandum ANL/MCS-TM-225, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[7] C. Bischof, L. Roh, and A. Mauer, *ADIC — An extensible automatic differentiation tool for ANSI-C*, Software–Practice and Experience, 27 (1997), pp. 1427–1456.

[8] H. Fischer, *Automatic differentiation of the vector that solves a parametric linear system*, Journal of Computational and Applied Mathematics, 35 (1991), pp. 169–184.

[9] J.-C. Gilbert, *Automatic differentiation and iterative processes*, Optimization Methods and Software, 1 (1992), pp. 13–22.

[10] A. Griewank, *On automatic differentiation*, in Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108.

[11] A. Griewank, C. Bischof, G. Corliss, A. Carle, and K. Williamson, *Derivative convergence of iterative equation solvers*, Optimization Methods and Software, 2 (1993), pp. 321–355.