On Combining Computational Differentiation and Toolkits for Parallel Scientific Computing*

Christian H. Bischof[1], H. Martin Bücker[1], and Paul D. Hovland[2]

[1] Institute for Scientific Computing, Aachen University of Technology,
52056 Aachen, Germany,
{bischof,buecker}@sc.rwth-aachen.de,
WWW home page: http://www.sc.rwth-aachen.de
[2] Mathematics and Computer Science Division, Argonne National Laboratory,
9700 South Cass Ave, Argonne, IL 60439, USA,
hovland@mcs.anl.gov,
WWW home page: http://www.mcs.anl.gov

Abstract. Automatic differentiation is a powerful technique for evaluating derivatives of functions given in the form of a high-level programming language such as Fortran, C, or C++. The program is treated as a potentially very long sequence of elementary statements to which the chain rule of differential calculus is applied over and over again. Combining automatic differentiation and the organizational structure of toolkits for parallel scientific computing provides a mechanism for evaluating derivatives by exploiting mathematical insight on a higher level. In these toolkits, algorithmic structures such as BLAS-like operations, linear and nonlinear solvers, or integrators for ordinary differential equations can be identified by their standardized interfaces and recognized as high-level mathematical objects rather than as a sequence of elementary statements. In this note, the differentiation of a linear solver with respect to some parameter vector is taken as an example. Mathematical insight is used to reformulate this problem into the solution of multiple linear systems that share the same coefficient matrix but differ in their right-hand sides. The experiments reported here use ADIC, a tool for the automatic differentiation of C programs, and PETSc, an object-oriented toolkit for the parallel solution of scientific problems modeled by partial differential equations.

1 Numerical versus Automatic Differentiation

Gradient methods for optimization problems and Newton's method for the solution of nonlinear systems are only two examples showing that computational techniques require the evaluation of derivatives of some objective function. In large-scale scientific applications, the objective function $f : \mathbb{R}^n \to \mathbb{R}^m$ is typically not available in analytic form but is given by a computer code written in a

* This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

high-level programming language such as Fortran, C, or C++. Think of f as the function computed by, say, a (parallel) computational fluid dynamics code consisting of hundreds of thousands lines that simulates the flow around a complex three-dimensional geometry. Given such a representation of the objective function $f(\mathbf{x}) = \big(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})\big)^T$, computational methods often demand the evaluation of the Jacobian matrix

$$J_f(\mathbf{x}) := \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{m \times n} \tag{1}$$

at some point of interest $\mathbf{x} \in \mathbb{R}^n$.

Deriving an analytic expression for $J_f(\mathbf{x})$ is often inadequate. Moreover, implementing such an analytic expression by hand is challenging, error-prone, and time-consuming. Hence, other approaches are typically preferred.

A well-known and widely used approach for the approximation of the Jacobian matrix is divided differences (DD). For the sake of simplicity, we mention only first-order forward DD but stress that the following discussion applies to DD as a technique of numerical differentiation in general. Using first-order forward DD, one can approximate the ith column of the Jacobian matrix (1) by

$$\frac{f(\mathbf{x} + h_i \mathbf{e}_i) - f(\mathbf{x})}{h_i}, \tag{2}$$

where h_i is a suitably-chosen step size and $\mathbf{e}_i \in \mathbb{R}^n$ is the ith Cartesian unit vector. An advantage of the DD approach is that the function f need be evaluated only at some suitably chosen points. Roughly speaking, f is used as a black box evaluated at some points. The main disadvantage of DD is that the accuracy of the approximation depends crucially on a suitable choice of these points, specifically, of the step size h_i. There is always the dilemma that the step size should be small in order to decrease the truncation error of (2) and that, on the other hand, the step size should be large to avoid cancellation errors using finite-precision arithmetic when evaluating (2).

Analytic and numerical differentiation methods are often considered to be the only options for computing derivatives. Another option, however, is symbolic differentiation by computer algebra packages such as Macsyma or Mathematica. Unfortunately, because of the rapid growth of the underlying explicit expressions for the derivatives, traditional symbolic differentiation is currently inefficient [9].

Another technique for computing derivatives of an objective function is automatic differentiation (AD) [10, 16]. Given a computer code for the objective function in virtually any high-level programming language such as Fortran, C, or C++, automatic differentiation tools such as ADIFOR [4, 5], ADIC [6], or ADOL-C [13] can by applied in a black-box fashion. A survey of AD tools can be found at `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools`. These tools generate another computer program, called a derivative-enhanced program, that evaluates $f(\mathbf{x})$ and $J_f(\mathbf{x})$ simultaneously. The key concept behind AD is

the fact that every computation, no matter how complicated, is executed on a computer as a—potentially very long—sequence of a limited set of elementary arithmetic operations such as additions, multiplications, and intrinsic functions such as `sin()` and `cos()`. By applying the chain rule of differential calculus over and over again to the composition of those elementary operations, $f(\mathbf{x})$ and $J_f(\mathbf{x})$ can be evaluated up to machine precision. Besides the advantage of accuracy, AD requires minimal human effort and has been proven more efficient than DD under a wide range of circumstances [3, 5, 12].

2 Computational Differentiation in Scientific Toolkits

Given the fact that automatic differentiation tools need not know anything about the underlying problem whose code is being differentiated, the resulting efficiency of the automatically generated code is remarkable. However, it is possible not only to apply AD technology in a black-box fashion but also to couple the application of AD with high-level knowledge about the underlying code. We refer to this combination as *computational differentiation* (CD). In some cases, CD can reduce memory requirements, improve performance, and increase accuracy. For instance, a CD strategy identifying a major computational component, deriving its analytical expression, and coding the corresponding derivatives by hand is likely to perform better than the standard AD approach that can operate only on the level of simple arithmetic operations.

In toolkits for scientific computations, algorithmic structures can be automatically recognized when applying AD tools, provided standardized interfaces are available. Examples include standard (BLAS-like) linear algebra kernels, linear and nonlinear solvers, and integrators for ordinary differential equations. These algorithmic structures are the key to exploiting high-level knowledge when CD is used to differentiate applications written in toolkits such as the Portable, Extensible Toolkit for Scientific Computation (PETSc) [1, 2].

Consider the case of differentiating a code for the solution of sparse systems of linear equations. PETSc provides a uniform interface to a variety of methods for solving these systems in parallel. Rather than applying an AD tool in a black-box fashion to a particular method as a sequence of elementary arithmetic operations, the combination of CD and PETSc allows us to generate a single derivative-enhanced program for any linear solver. More precisely, assume that we are concerned with a code for the solution of

$$A \cdot \mathbf{x}(\mathbf{s}) = \mathbf{b}(\mathbf{s}) \tag{3}$$

where $A \in \mathbb{R}^{N \times N}$ is the coefficient matrix. For the sake of simplicity, it is assumed that only the solution $\mathbf{x}(\mathbf{s}) \in \mathbb{R}^N$ and the right-hand side $\mathbf{b}(\mathbf{s}) \in \mathbb{R}^N$, but not the coefficient matrix, depend on a free parameter vector $\mathbf{s} \in \mathbb{R}^r$. The code for the solution of (3) implicitly defines a function $\mathbf{x}(\mathbf{s})$. Now, suppose that one is interested in the Jacobian $J_{\mathbf{x}}(\mathbf{s}) \in \mathbb{R}^{N \times r}$ of the solution $\mathbf{x}$ with respect to the free parameter vector $\mathbf{s}$. Differentiating (3) with respect to $\mathbf{s}$ yields

$$A \cdot J_{\mathbf{x}}(\mathbf{s}) = J_{\mathbf{b}}(\mathbf{s}), \tag{4}$$

where $J_{\mathbf{b}}(\mathbf{s}) \in \mathbb{R}^{N \times r}$ denotes the Jacobian of the right-hand side $\mathbf{b}$.

In parallel high-performance computing, the coefficient matrix A is often large and sparse. For instance, numerical simulations based on partial differential equations typically lead to such systems. Krylov subspace methods [17] are currently considered to be among the most powerful techniques for the solution of sparse linear systems. These iterative methods generate a sequence of approximations to the exact solution $\mathbf{x}(\mathbf{s})$ of the system (3). Hence, an implementation of a Krylov subspace method does not compute the function $\mathbf{x}(\mathbf{s})$ but only an approximation to that function. Since, in this case, AD is applied to the approximation of a function rather than to the function itself, one may ask whether and how AD-produced derivatives are reasonable approximations to the desired derivatives of the function $\mathbf{x}(\mathbf{s})$. This sometimes undesired side-effect is discussed in more detail in [8, 11] and can be minimized by the following CD approach.

Recall that the standard AD approach would process the given code for a particular linear solver for (3), say an implementation of the biconjugate gradient method, as a sequence of binary additions, multiplications, and the like. In contrast, combining the CD approach with PETSc consists of the following steps:

1. Recognize from inspection of PETSc's interface that the code is meant to solve a linear system of type (3) regardless of which particular iterative method is used.
2. Exploit the knowledge that the Jacobian $J_{\mathbf{x}}(\mathbf{s})$ is given by the solution of the multiple linear systems (4) involving the same coefficient matrix, but r different right-hand sides.

The CD approach obviously eliminates the above mentioned problems with automatic differentiation of iterative schemes for the approximation of functions. There is also the advantage that the CD approach abstracts from the particular linear solver. Differentiation of codes involving any linear solver, not only those making use of the biconjugate gradient method, benefits from an efficient technique to solve (4).

3 Potential Gain of CD and Future Research Directions

A previous study [14] differentiating PETSc with ADIC has shown that, for iterative linear solvers, CD-produced derivatives are to be preferred to derivatives obtained from AD or DD. More precisely, the findings from that study with respect to differentiation of linear solvers are as follows. The derivatives produced by the CD and AD approaches are several orders of magnitude more accurate than those produced by DD. Compared with AD, the accuracy of CD is higher. In addition, the CD-produced derivatives are obtained in less execution time than those by AD, which in turn is faster than DD. The differences in execution time between these three approaches increase with increasing the dimension, r, of the free parameter vector $\mathbf{s}$.

While the CD approach turns out to be clearly the best of the three discussed approaches, its performance can be improved significantly. The linear systems (4)

involving the same coefficient matrix but r different right-hand sides are solved in [14] by running r times a typical Krylov subspace method for a linear system with a single right-hand side. In contrast to these successive runs, so-called block versions of Krylov subspace methods are suitable candidates for solving systems with multiple right-hand sides; see [7, 15] and the references given there. In each block iteration, block Krylov methods generate r iterates simultaneously, each of which is designed to be an approximation to the exact solutions of a single system. Note that direct methods such as Gaussian elimination can be trivially adapted to multiple linear systems because their computational work is dominated by the factorization of the coefficient matrix. Once the factorization is available, the solutions of multiple linear systems are given by a forward and back substitution per right-hand side. However, because of the excessive amount of fill-in, direct methods are often inappropriate for large sparse systems.

In this note, we extend the work reported in [14] by incorporating iterative block methods into the CD approach. Based on the given scenario of the combination of the ADIC tool and the PETSc package, we consider a parallel implementation of a block version of the biconjugate gradient method [15]. We focus here on some fundamental issues illustrating this approach; a rigorous numerical treatment will be presented elsewhere. To demonstrate the potential gain from using a block method in contrast to successive runs of a typical iterative method, we take the number of matrix-vector multiplications as a rough performance measure. This is a legitimate choice because, usually, the matrix-vector multiplications dominate the computational work of an iterative method for large sparse systems.

Figure 1 shows, on a log scale, the convergence behavior of the block biconjugate gradient method applied to a system arising from a discretization of a two-dimensional partial differential equation of order $N = 1,600$ with $r = 3$ right-hand sides. Throughout this note, we always consider the relative residual norm, that is, the Euclidean norm of the residual scaled by the Euclidean norm of the initial residual. In this example, the iterates for the $r = 3$ systems converge at the same step of the block iteration. In general, however, these iterates converge at different steps. Future work will therefore be concerned with how to detect and deflate converged systems. Such deflation techniques are crucial to block methods because the algorithm would break down in the next block iteration step; see the discussion in [7] for more details on deflation. We further assume that block iterates converge at the same step and that deflation is not necessary.

Next, we consider a finer discretization of the same equation leading to a larger system of order $N = 62,500$ with $r = 7$ right-hand sides to illustrate the potential gain of block methods. Figure 2 compares the convergence history of applying a block method to obtain block iterates for all $r = 7$ systems simultaneously and running a typical iterative method for a single right-hand side $r = 7$ times one after another. For all our experiments, we use the biconjugate gradient method provided by the linear equation solver (SLES) component of PETSc as a typical iterative method for a single right-hand side. For the plot

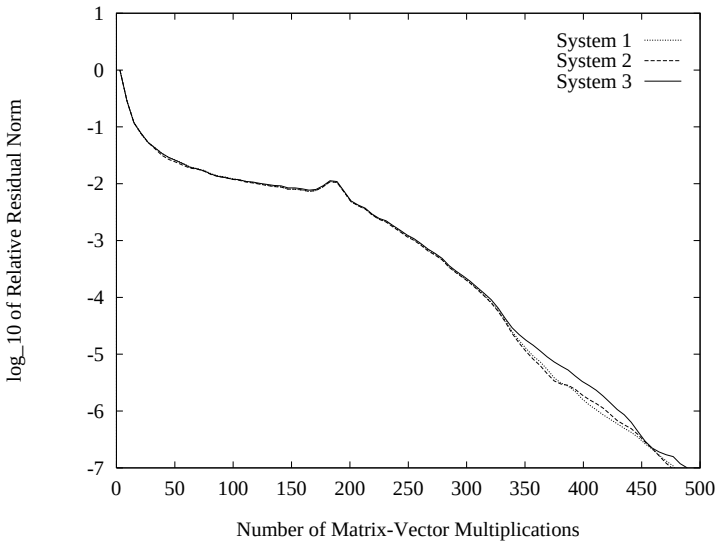

Fig. 1. Convergence history of the block method for the solution of $r = 3$ systems involving the same coefficient matrix of order $N = 1,600$. The residual norm is shown for each of the systems individually.

of the block method we use the largest relative residual norm of all systems. In this example, the biconjugate gradient method for a single right-hand side (dotted curve) needs $8,031$ matrix-vector multiplications to achieve a tolerance of 10^{-7} in the relative residual norm. The block method (solid curve), on the contrary, converges in only $5,089$ matrix-vector multiplications to achieve the same tolerance. Clearly, block methods offer a potential speedup in comparison with successive runs of methods for a single right-hand side.

The ratio of the number of matrix-vector multiplications of the method for a single right-hand side to the number of matrix-vector multiplications of the block method is 1.58 in the example above and is given in the corresponding column of Table 1. In addition to the case where the number of right-hand sides is $r = 7$, this table contains the results for the same coefficient matrix, but for varying numbers of right-hand sides. It is not surprising that the number of matrix-vector multiplications needed to converge increases with an increasing number of right-hand sides r. Note, however, that the ratio also increases with r. This behavior shows that the larger the number of right-hand sides the more attractive the use of block methods.

Many interesting aspects remain to be investigated. Besides the above mentioned deflation technique, there is the question of determining a suitable preconditioner. Here, we completely omitted preconditioning in order to make the comparison between the block method and its correspondence for a single right-hand side more visible. Nevertheless, preconditioning is an important ingredient in any iterative technique for the solution of sparse linear systems for both single

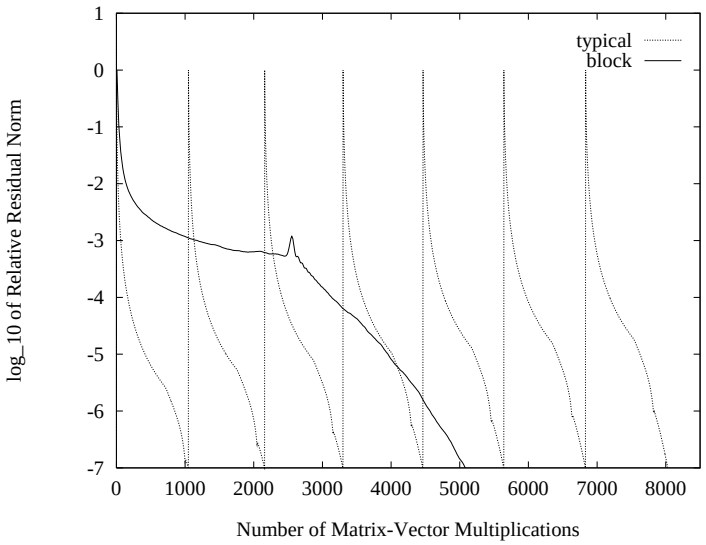

Fig. 2. Comparison of the block method for the solution of $r = 7$ systems involving the same coefficient matrix of order $N = 62,500$ and r successive runs of a typical iterative method for a single right-hand side.

and multiple right-hand sides. Notice that, in their method, Freund and Malhotra [7] report a dependence of the choice of an appropriate preconditioner on the parameter r.

Block methods are also of interest because they offer the potential for better performance. At the single-processor level, performing several matrix-vector products simultaneously provides increased temporal locality for the matrix, thus mitigating the effects of the memory bandwidth bottleneck. The availability of several vectors at the same time also provides opportunities for increased parallel performance, as increased data locality reduces the ratio of communi-

Table 1. Comparison of matrix-vector multiplications needed to require a decrease of seven orders of magnitude in the relative residual norm for different dimensions, r, of the free parameter vector. The rows show the number of matrix-vector multiplications for r successive runs of a typical iterative method for a single right-hand side, a corresponding block version, and their ratio, respectively. (The order of the matrix is $N = 62,500$.)

r	1	2	3	4	5	6	7	8	9	10
typical	1,047	2,157	3,299	4,463	5,641	6,831	8,031	9,237	10,451	11,669
block	971	1,770	2,361	3,060	3,815	4,554	5,089	5,624	6,219	6,550
ratio	1.08	1.22	1.40	1.46	1.48	1.50	1.58	1.64	1.68	1.78

cation to computation. Even for the single right-hand side case, block methods are attractive because of their potential for exploiting locality, a key issue in implementing techniques for high-performance computers.

4 Concluding Remarks

Automatic differentiation applied to toolkits for parallel scientific computing such as PETSc increases their functionality significantly. While automatic differentiation is more accurate and, under a wide range of circumstances, faster than approximating derivatives numerically, its performance can be improved even further by exploiting high-level mathematical knowledge. The organizational structure of toolkits provides this information in a natural way by relying on standardized interfaces for high-level algorithmic structures. The reason why improvements over the traditional form of automatic differentiation are possible is that, in the traditional approach, any program is treated as a sequence of elementary statements. Though powerful, automatic differentiation operates on the level of statements. In contrast, *computational differentiation*, the combination of mechanically applying techniques of automatic differentiation and human-guided mathematical insight, allows the analysis of objects on higher levels than on the level of elementary statements. These issues are demonstrated by taking the differentiation of an iterative solver for the solution of large sparse systems of linear equations as an example. Here, mathematical insight consists in reformulating the differentiation of a linear solver into a solution of multiple linear systems involving the same coefficient matrix, but whose right-hand sides differ. The reformulation enables the integration of appropriate techniques for the problem of solving multiple linear systems, leading to a significant performance improvement when differentiating code for any linear solver.

Acknowledgments

This work was completed while the second author was visiting the Mathematics and Computer Science Division, Argonne National Laboratory. He was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. Gail Pieper proofread a draft of this manuscript and gave several helpful comments.

References

[1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[3] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, 1996.

[4] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

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

[6] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[7] Roland W. Freund and Manish Malhotra. A block QMR algorithm for non-Hermitian linear systems with multiple right-hand sides. *Linear Algebra and Its Applications*, 254:119–157, 1997.

[8] Jean-Charles Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13–22, 1992.

[9] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[10] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, 2000.

[11] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence of iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[12] Andreas Griewank and George Corliss. *Automatic Differentiation of Algorithms.* SIAM, Philadelphia, 1991.

[13] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[14] Paul Hovland, Boyana Norris, Lucas Roh, and Barry Smith. Developing a derivative-enhanced object-oriented toolkit for scientific computations. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, pages 129–137, Philadelphia, 1999. SIAM.

[15] Dianne P. O'Leary. The block conjugated gradient algorithm and related methods. *Linear Algebra and Its Applications*, 29:293–322, 1980.

[16] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1996.