Challenges and Opportunities in Using Automatic Differentiation with Object-Oriented Toolkits for Scientific Computing*

Paul Hovland[1], Steven Lee[2], Lois McInnes[1], Boyana Norris[1], and Barry Smith[1]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439-4844.
[2] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Box 808, L-560, Livermore, CA 94551.

Abstract. The increased use of object-oriented toolkits in large-scale scientific simulation presents new opportunities and challenges for the use of automatic (or algorithmic) differentiation (AD) techniques, especially in the context of optimization. Because object-oriented toolkits use well-defined interfaces and data structures, there is potential for simplifying the AD process. Furthermore, derivative computation can be improved by exploiting high-level information about numerical and computational abstractions. However, challenges to the successful use of AD with these toolkits also exist. Among the greatest challenges is balancing the desire to limit the scope of the AD process with the desire to minimize the work required of a user. We discuss our experiences in integrating AD with the PETSc, PVODE, and TAO toolkits and our plans for future research and development in this area.

1 Introduction

The ever-increasing complexity of advanced computational science applications has led to an increase in the use of object-oriented software practices in the development of scientific applications and toolkits. A good object-oriented design increases productivity by allowing developers to focus on a small component of a complex system. Furthermore, increased code reuse provides justification for expending significant effort in the development of highly optimized object-oriented toolkits.

Many high-performance numerical toolkits include components designed to be combined with an application-specific nonlinear function. Examples include optimization components, nonlinear equation solvers, and differential algebraic equation solvers. Often the numerical methods implemented by these components also require first and possibly second derivatives of the

* The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

function. Frequently, the toolkit is able to approximate these derivatives by using finite differences; however, the convergence rate and robustness are often improved if the derivatives are computed analytically.

Developing correct parallel code for computing the analytic derivatives of a complicated nonlinear function can be an onerous task, especially when second derivatives are required. An automated alternative such as automatic differentiation (AD) [16,17] is therefore very attractive as a mechanism for providing these analytic derivatives. Furthermore, because object-oriented toolkits provide well-defined interfaces for nonlinear functions, the AD process can potentially be simplified.

We examine the use of AD in conjunction with object-oriented numerical toolkits. In particular, we describe the use of AD to provide first and, where appropriate, second derivatives in conjunction with the Portable, Extensible Toolkit for Scientific Computing (PETSc), the Toolkit for Advanced Optimization (TAO), and a parallel ODE solver for computing sensitivities (SensPVODE).

This paper is organized as follows. Section 2 provides a brief intoduction to automatic differentiation. Section 3 introduces the toolkits considered in this paper. Section 4 discusses the opportunities and challenges that arise in using AD with these toolkits. Section 5 provides a synopsis of experimental results. Section 6 summarizes our experiences and describes our expectations for using AD in the context of PDE-constrained optimization.

2 Automatic Differentiation

Automatic, or algorithmic, differentiation is a technique for augmenting arbitrarily complex computer subprograms with instructions for the computation of derivatives. The technique combines rules for analytically differentiating the finite number of elemental functions in a programming language with the chain rule of differential calculus.

The two principal approaches to implementing AD are operator overloading and compiler-based source transformation. Each method has its advantages and disadvantages [8]. There are also two basic modes of AD, the forward mode and the reverse mode. The forward mode is particularly appropriate when the number of independent variables is small or when a small number of directional derivatives are required. The reverse mode is attractive when the number of dependent variables is small or when a Jacobian-transpose-vector product is required. We focus on the source transformation approach using the forward mode, but most of the issues discussed in this paper are also applicable to the operator overloading approach and the reverse mode. More details of how AD works and various methods for exploiting chain rule associativity can be found in [17].

Figure 1 shows a simple example of the code generated by the ADIFOR [7] source transformation tool for Fortran 77. While this example is not indicative of the power of automatic differtiation, which is equally applicable to applications spanning thousands or millions of lines of code and utilizing complex control structures, it does illustrate some important concepts.

```
y(1) = 10.0 * (x2-x1*x1)         d2_b = dble(10.0)
y(2) = 1.0 - x1                  d5_b = (-d2_b) * x1 + (-d2_b) * x1
                                 do g_i_ = 1, g_p_
                                   g_y(g_i_, 1) = d5_b * g_x1(g_i_) +
                                +                 d2_b * g_x2(g_i_)
                                 enddo
                                 y(1) = dble(10.0) * (x2 - x1 * x1)

                                 do g_i_ = 1, g_p_
                                   g_y(g_i_, 2) = -g_x1(g_i_)
                                 enddo
                                 y(2) = 1.0d0 - x1
```

Fig. 1. A simple example of automatic differentiation. The code on the right was generated by the ADIFOR tool from the code fragment on the left.

Each scalar variable in the original program has associated with it a derivative vector (sometimes generically referred to as a derivative object). In the case of ADIFOR-generated code, this association is by name. For example, the derivative vector associated with `x1` is `g_x1`. In the case of an AD tool for C, such as ADIC [10], association by name is not possible because of aliasing. Instead, association must be by address. ADIC accomplishes this association by converting all floating-point variables to a new datatype, `DERIV_TYPE`.

```
typedef struct {
        double value;
        double  grad[ad_GRAD_MAX];
} DERIV_TYPE;
```

The `value` field carries the original floating-point value, while the `grad` field contains the associated derivative vector.

The derivative vectors associated with independent variables are collectively referred to as the seed matrix. The seed matrix can be initialized such that upon completion of the computation the derivative vectors associated with the dependent variables contain the Jacobian, a Jacobian-vector product, or an arbitrary Jacobian-matrix product. For example, if in our simple example we initialize `g_x1` to $[1.0, 0.0]$ and `g_x2` to $[0.0, 1.0]$, then `g_y(i,j)` will contain $\partial y_j / \partial xi$.

3 Toolkits

We have investigated the use of AD in conjunction with three object-oriented toolkits for scientific computing: PETSc, TAO, and SensPVODE. All of these toolkits employ an object-oriented design but are implemented in C. In the next sections, we briefly describe these toolkits and the role derivatives play.

3.1 Portable, Extensible Toolkit for Scientific Computing

PETSc [2,3] is a suite of data structures and routines for the scalable solution of scientific applications modeled by partial differential equations. The software integrates a hierarchy of components that range from low-level distributed data structures for vectors and matrices through high-level linear, nonlinear, and timestepping solvers. The algorithmic source code is written in high-level abstractions so that it can be easily understood and modified. This approach promotes code reuse and flexibility and, in many cases, helps to decouple issues of parallelism from algorithm choices.

PETSc provides a suite of nonlinear solvers that couple a Newton-based method, offering the advantage of rapid convergence when an iterate is near to a problem's solution, with a line search, trust region, and pseudo-transient continuation strategy to extend the radius of convergence of the Newton techniques. The linearized systems are typically solved inexactly with preconditioned Krylov methods. The basic Newton method requires the Jacobian matrix, $J = F'(u)$, of a nonlinear function $F(u)$. Matrix-free Newton-Krylov methods require Jacobian-vector products, $F'(u)v$, and may require an approximate Jacobian for preconditioning.

PETSc also provides components for managing computations on structured grids, including hierarchies of grids for use in multigrid methods. Among the functions provided by these components are generalized gather-scatter operations for communicating ghost values, colorings for use in finite difference (and AD) Jacobian computations, and simplified facilities for mapping between local and global indices.

3.2 Toolkit for Advanced Optimization

TAO [4,5] focuses on scalable optimization software, including nonlinear least squares, unconstrained minimization, bound-constrained optimization, and general nonlinear optimization. The TAO optimization algorithms use high-level abstractions for matrices and vectors and emphasize the reuse of external tools where appropriate, including support for using the linear algebra components provided by PETSc and related tools.

Many of the algorithms employed by TAO require first and sometimes second derivatives. For example, unconstrained minimization solvers that require the gradient, $f'(u)$, of an objective function, $f(u)$, include a limited-memory variable metric method and a conjugate gradient method, while solvers that require both the gradient, $f'(u)$, and Hessian, $f''(u)$, (or Hessian-vector products) include line search and trust region variants of Newton methods. In addition, algorithms for nonlinear least squares and constrained optimization often require the Jacobian of the constraint functions.

3.3 SensPVODE

PVODE [11] is a high-performance ordinary differential equation solver for the types of initial value problems (IVPs) that arise in large-scale computational simulations.

$$y'(t) = f(t, y, p), \quad y(t_0) = y_0(p), \quad y \in \mathbf{R}^N, \quad p \in \mathbf{R}^m. \tag{1}$$

Often, one wants to compute sensitivities with respect to certain parameters p_i in the IVP, $s_i = \partial y / \partial p_i$. SensPVODE [21] is a variant of PVODE that simultaneously solves the original ODE and the sensitivity ODEs.

$$s_i'(t) = \frac{\partial f}{\partial y} s_i(t) + \frac{\partial f}{\partial p_i}, \; s_i(t_0) = \frac{\partial y_0(p)}{\partial p_i}, \; i = 1, \cdots, m. \tag{2}$$

Thus, SensPVODE requires the derivatives $\frac{\partial f}{\partial y} s_i(t) + \frac{\partial f}{\partial p_i}$.

4 Using Automatic Differentiation with Object-Oriented Toolkits

Automatic differentiation can be used in conjunction with object-oriented toolkits at many different levels. At the highest level, AD can be applied to a toolkit to facilitate sensitivity analysis and optimization of models constructed with the toolkit. Another option is to use AD to provide the derivatives required by the toolkit. To provide these derivatives, AD can be applied directly to the parallel nonlinear function. Alternatively, AD can be applied to the building blocks of the nonlinear function, such as the nonlinear function on a local subdomain or an element or vertex function. In the following sections we consider some of the opportunities and the challenges in applying AD at these various levels.

4.1 Toolkit Level

Applying AD to an object-oriented toolkit for scientific computing offers the opportunity to compute derivatives (also called sensitivities) of scientific applications that use the toolkit. These derivatives can be used for sensitivity analysis, to understand the sensitivity of the simulation results to uncertainties in model parameters, or for optimization. Applying AD to a toolkit also provides an opportunity to employ so-called computational differentiation techniques exploiting high-level mathematical and algorithmic features.

Using computational differentiation techniques can also circumvent some of the challenges that arise in applying AD to a toolkit, including the fact that functions and derivatives may not converge at the same rate in an iterative method and that certain numerical methods, including ODE solvers with adaptive stepsize control, introduce feedback into a computation that may seriously influence the derivatives.

One simple example of a computational differentiation technique arises in the differentiation of a linear solver. Rather than differentiate through a preconditioner and iterative Krylov solver, one can solve directly a linear system with multiple right-hand sides, perhaps using block Krylov methods [6,24] or a projection method [13]. We have conducted preliminary research in this area in developing a differentiated version of PETSc [19]. A more sophisticated example of computational differentiation, applicable to other types of iterative solvers, is described elsewhere in this volume [?]. We also note that the development of SensPVODE may be interpreted as applying computational differentiation techniques to the PVODE toolkit.

4.2 Parallel Nonlinear Function Level

Automatic differentiation is often not quite "automatic." Often, the user of an AD tool needs to specify the independent and dependent variables and possibly initialize the seed matrix to indicate exactly which derivatives are to be computed. The use of AD in conjunction with a numerical toolkit promises to make full automation possible because of the use of well-defined interfaces. For example, the nonlinear solver in PETSc requires a nonlinear function adhering to the interface

```
int Function(SNES, Vec, Vec, void *);
```

while SensPVODE requires a function adhering to the interface

```
void function(integer, Real, N_Vector, N_Vector, void *);
```

In both cases, all of the required information regarding the derivatives to be computed is known in advance, making automation of the AD process possible. Other work [12,14,23] has also demonstrated the benefits of well-defined interfaces for automating the AD process.

However, applying AD directly to the parallel nonlinear function required by the toolkit is not without challenges. The function may include many calls to toolkit support functions. Thus, differentiated versions of these functions must be developed using automatic or computational differentiation techniques. In many cases, these support functions are used for communicating ghost values or performing similar problem setup or data movement functions. Consequently, the differentiated versions of the functions may end up performing unnecessary work. Parallelism is also an issue. The nonlinear function may use OpenMP or make calls to MPI functions, so the AD tool must support the parallel programming paradigm being used. Furthermore, the parameters to the function are likely parallel objects, as is the case with the `Vec` and `N_Vector` objects in the examples above. Therefore, care must be taken in the way seed matrices are initialized; additional communication may also be required.

An important issue in the differentiation of application functions arises from the benign-looking `void *` that appears as the final argument in both of the examples above. This argument is used to pass a pointer to an application-specific data structure that contains various data objects of use to the application. Figure 2 contains several examples of these data structures.

As discussed earlier, in order to provide association by address between variables and their derivative vectors, ADIC changes the datatypes of all floating-point variables. The consequence for these application-specific data structures is that if AD is naively applied, the type of several fields within the data structure will be changed. Therefore, all application functions that access the data structure, not just the nonlinear function, must be modified to use the new datatypes.

An alternative is to use two data structures, one with the original datatypes and one with the derivative datatypes, and to copy data between them. However, as the final example in Figure 2 illustrates, some care must be taken with this approach. If the data structure includes workspace, such as the `uext` field in this example, it may be inefficient to copy such unneeded data. Furthermore, it is usually not possible to generate code automatically for copying from one data structure to another. Therefore, at this point in our work coupling ADIC with SensPVODE, we require the user to provide the routine for copying data between the application-specific data structures and their differentiated counterparts.

4.3 Local Subdomain Level

As alluded to in the preceding section, many parallel nonlinear functions follow the pattern gather/scatter ghost values, compute function on local subdomain, and assemble/map function from local to global indices. Therefore, an approach that avoids many of the complications associated with applying AD to the full parallel nonlinear function is to apply AD only to the local subdomain function, with manual modifications to the setup and assembly phases. We have examined this approach in conjunction with PETSc and TAO by manually extracting the local subdomain function, as illustrated in Figure 3 and described in [1,9].

Recent work has examined how we can use the structured grid components to simplify the process. The setup and assembly phases are essentially the same for most nonlinear functions on a structured grid. Therefore, instead of extracting the subdomain computation from the parallel nonlinear function, we ask the user to provide only the subdomain function. Instead of providing a nonlinear function with the interface

```
typedef struct {
   double      param;          /* test problem parameter */
   int         mx,my;          /* discretization in x, y directions */
   Vec         localX,localF;  /* ghosted local vectors */
   DA          da;             /* distributed array data structure */
   int         rank;           /* processor rank */
} AppCtx;

typedef struct {
   double     lidvelocity,prandtl,grashof;
   PetscTruth draw_contours;
} AppCtx;

typedef struct {
  real om, dx, dy, q4;
  real uext[NVARS*(MXSUB+2)*(MYSUB+2)];
  integer my_pe, isubx, isuby, nvmxsub, nvmxsub2;
  real *p;
  real *pbar;
  MPI_Comm comm;
} *UserData;
```

Fig. 2. Examples of application-specific data structures used in PETSc, TAO, and SensPVODE applications.

```
int Function(SNES, Vec, Vec, void *);
```

the user provides, for example, a local subdomain function with the interface

```
int LocalFormFunction2d(Field  **, Field  **, Coords, void *);
```

where `Coords` contains information about the corners of the subdomain and `Field` is a structure with a number of scalar fields corresponding to the number of degrees of freedom at each vertex. The setup and assembly are handled by the structured grid component. Given a differentiated version of `LocalFormFunction2d`, the structured grid component can also perform the necessary setup, including seed matrix intialization, and assembly of the Jacobian. The setup and assembly phases use coloring [15] to reduce the length of the derivative vectors to the stencil size times the number of degrees of freedom. Future research will extend this work to Hessian computations and unstructured meshes.

4.4 Element or Vertex Function Level

A final option is to differentiate the element or vertex function and then assemble the full Jacobian from the element (or vertex) Jacobians. This is common practice in finite element computations, especially when the element

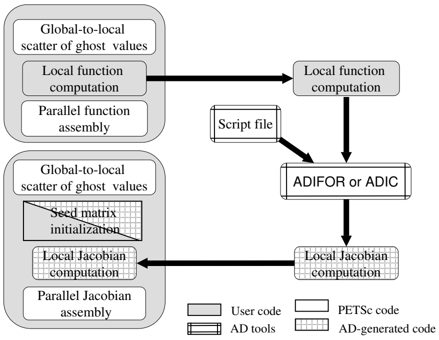


Fig. 3. Schematic diagram of the use of automatic differentiation tools to generate the Jacobian routine for a nonlinear PDE computation.

Jacobian is simple to derive by hand. For complex element or vertex functions, or for higher-order derivatives, AD can be employed. This approach eliminates the need for a matrix coloring, reduces the memory requirements, and is easily extended to second and higher-order derivatives.

The principal impediment to this approach is that not all functions are easily decomposed to this level. It may be computationally more efficient to precompute fluxes for all of the edges/faces in the subdomain, then use these fluxes in computing the element functions. Furthermore, boundary conditions may introduce many special cases. It is computationally more efficient to handle these special cases separately, then loop over the remaining elements, than to test every element to determine whether a special case applies.

5 Experimental Results

We provide a brief synopsis of experimental results from the use of AD in conjunction with the PETSc and SensPVODE toolkits. More detailed descriptions of these and related experiments can be found in [1,19,20,22].

5.1 Toolkit Level

In [19] we describe research into the application of automatic and computational differentiation to PETSc. Here we briefly discuss some of the relevant experimental results.

We have tested the differentiated version of PETSc, and in particular its linear solver component, with an example involving the solution of a linear system of equations $A\,x = b$ where A is the 256×256 matrix whose sparsity pattern corresponds to a five-point stencil discretization of a 16×16 computational domain. In all of the experiments, we have used a GMRES solver in combination with an incomplete LU factorization preconditioner.

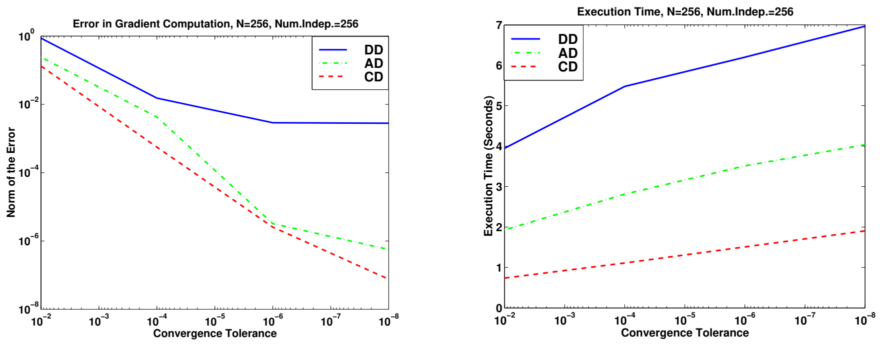


Fig. 4. Gradient error and execution time with varying convergence tolerances.

Figure 4 shows the accuracy and performance results for various convergence tolerances. DD (for “divided differences”) designates finite difference approximation, AD designates black-box automatic differentiation, and CD stands for computational differentiation using successive linear solves on the multiple right-hand sides. 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 finite difference and AD approaches the convergence tolerance refers to the convergence of x, whereas in the computational differentiation approach it refers to the convergence of ∇x. In this example, the computational differentiation approach exhibits significant performance improvement over finite differences and AD.

5.2 Parallel Nonlinear Function Level

We applied SensPVODE to a simple test case, a two-species diurnal kinetics advection-diffusion system in two space dimensions. The PDEs can be written as

$$\frac{\partial c_i}{\partial t} = K_h \frac{\partial^2 c_i}{\partial x^2} + V \frac{\partial c_i}{\partial x} + \frac{\partial}{\partial y}\left(K_v(y)\frac{\partial c_i}{\partial y}\right) + R_i(c_1, c_2, t) \quad (i = 1, 2),$$

where the subscripts i are used to distinguish the chemical species. The reaction terms are given by

$$R_1(c_1, c_2, t) = -q_1 c_1 c_3 - q_2 c_1 c_2 + 2q_3(t) c_3 + q_4(t) c_2 \text{ and}$$
$$R_2(c_1, c_2, t) = q_1 c_1 c_3 - q_2 c_1 c_2 - q_4(t) c_2,$$

and $K_v(y) = K_0 e^{(y/5)}$. The scalar constants for this problem are $K_h = 4.0 \times 10^{-6}$, $V = 10^{-3}$, $K_0 = 10^{-8}$, $q_1 = 1.63 \times 10^{-16}$, $q_2 = 4.66 \times 10^{-16}$, and $c_3 = 3.7 \times 10^{16}$. The diurnal rate constants are

$$\begin{aligned} q_i(t) &= e^{[-a_i/\sin \omega t]} && \text{for } \sin \omega t > 0, \\ q_i(t) &= 0 && \text{for } \sin \omega t \leq 0, \end{aligned}$$

where $i = 3$ and 4, $\omega = \pi/43200$, $a_3 = 22.62$, and $a_4 = 7.601$. The time interval of integration is $[0, 86400]$, representing 24 hours measured in seconds.

The problem is posed on the square $0 \leq x \leq 20$, $30 \leq y \leq 50$ (all in km), with homogeneous Neumann boundary conditions. The PDE system is treated by central differences on a uniform 100×100 grid, with simple polynomial initial profiles. See [21] for more details. For the purpose of sensitivity analysis, we identify the following 8 parameters associated with this problem: $p_1 = q_1$, $p_2 = q_2$, $p_3 = c_3$, $p_4 = a_3$, $p_5 = a_4$, $p_6 = K_h$, $p_7 = V$, and $p_8 = K_0$.

We varied the number of sensitivities from 1 to 8 and compared the use of AD with three finite difference methods to compute the derivatives $\frac{\partial f}{\partial y} s_i(t) + \frac{\partial f}{\partial p_i}$ required by SensPVODE. In the combined central difference method, p_i and y are perturbed simultaneously to obtain both terms in the derivative expression. In the other finite difference methods, the terms are approximated separately. See [21] for a complete description of the finite difference strategies.

Figures 5 and 6 summarize our results on 16 nodes of a Linux cluster. Each node in the cluster has two 550 MHz Pentium III processors (only one processor per node was used) and Myrinet interconnect. AD shows a significant performance advantage over the finite difference methods. However, as Figure 6 illustrates, the performance improvements are due to a reduction in the number of timesteps required; the runtime per timestep actually increases.

5.3 Local Subdomain Level

We used AD to provide the directional derivatives required by PETSc to solve the steady-state, three-dimensional compressible Euler equations on mapped, structured meshes using a second-order, Roe-type, finite-volume discretization. We solved in parallel a nonlinear system, using matrix-free Newton-Krylov-Schwarz algorithms with pseudo-transient continuation to model transonic flow over an ONERA M6 airplane wing. See [18] for details about the problem formulation and algorithmic approach. The linearized

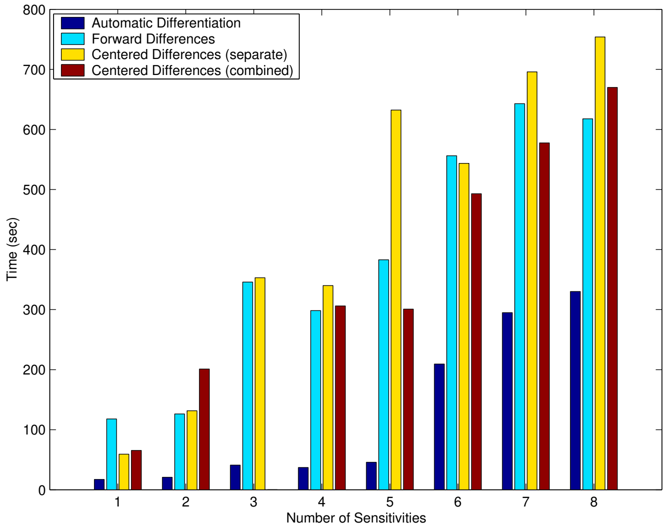


Fig. 5. Comparison of SensPVODE performance (total time to solution) for various derivative computation strategies. Results are the average of three runs on 16 processors of a Linux cluster.

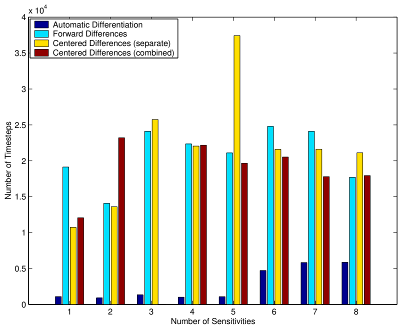
Newton correction equations were solved by using restarted GMRES preconditioned with the restricted additive Schwarz method with one degree of overlap.

As discussed in depth in [20] and summarized in Figure 7, our results indicate that, for matrix-free Newton-Krylov methods, AD offers significantly greater robustness and provides better algorithmic performance than do finite difference approximations (FD). However, because the directional derivatives required by a matrix-free method can be computed less expensively by FD than by AD, AD does not always provide a performance advantage in terms of runtime [20]. We are therefore investigating hybrid AD-FD strategies, which combine the robustness of AD with the reduced cost of FD.

6 Conclusions and Expectations

The combination of AD and object-oriented toolkits has proven to be an effective instrument for scientific computing. The analytic derivatives of AD enhance robustness and accelerate the convergence of Newton methods. The well-defined interfaces and data encapsulation of object-oriented toolkits simplify the AD process. There are many options as to the level at which AD can be applied. Each level has its advantages and disadvantages.

AD offers great promise as a useful tool in PDE-constrained optimization. Analytic derivatives are often necessary to ensure robust and efficient



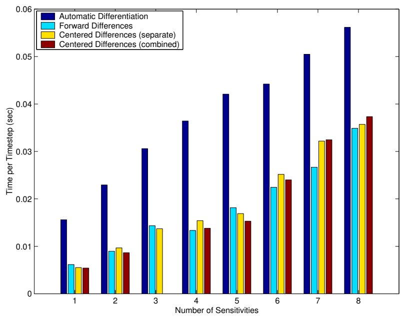


Fig. 6. Number of timesteps and time per timestep for various derivative computation strategies in SensPVODE. Results are the average of three runs on 16 processors of a Linux cluster.

convergence to the true minimum. For complex PDE-based simulations, however, developing analytic derivatives, especially second derivatives, by hand is often intractible. Judicious use of AD can overcome these obstacles. Furthermore, the reverse mode of AD enables gradients, Jacobian-transpose-vector, and Hessian-vector products to be computed at significantly lower cost than is possible with finite difference approximations. The ability to compute accurately and efficiently a full suite of first and second derivative matrices and directional derivatives should facilitate the algorithmic experimentation necessary for the advancement of PDE-constrained optimization. Thus, AD can play an important role in advancing both the science and the practice of PDE-contrained optimization.

Acknowledgments

The work of S. Lee was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. The other authors were 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.

We thank Peter Brown, Alan Hindmarsh, David Keyes, Matt Knepley, Jorge Moré, and Linda Petzold for informative discussions and Gail Pieper for proofreading an early draft of this manuscript.

References

1. J. Abate, S. Benson, L. Grignon, P. Hovland, L. McInnes, and B. Norris. Integrating automatic differentiation with object-oriented toolkits for high-performance scientific computing. Technical Report ANL/MCS-P820-0500,

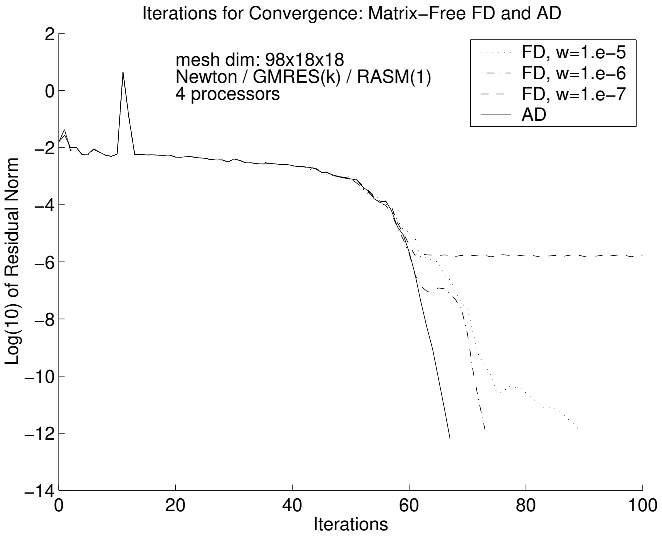


Fig. 7. Algorithmic performance of automatic differentiation derivatives versus finite difference approximations

Mathematics and Computer Science Division, Argonne National Laboratory, 2000.

2. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.
3. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.28, Argonne National Laboratory, March 2000. See `http://www.mcs.anl.gov/petsc`.
4. S. Benson, L. C. McInnes, and J. Moré. GPCG: A case study in the performance and scalability of optimization algorithms. Technical Report ANL/MCS-P768-0799, Mathematics and Computer Science Division, Argonne National Laboratory, 1999.
5. S. Benson, L. C. McInnes, and J. Moré. TAO users manual. Technical Report ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory, 2000. See `http://www.mcs.anl.gov/tao`.
6. C. Bischof, M. Bücker, and P. Hovland. On combining computational differentiation and toolkits for parallel scientific computing. Technical Report ANL/MCS-P797-0200, Mathematics and Computer Science Division, Argonne National Laboratory, 2000. To appear in Proceedings of EuroPar 2000.
7. C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
8. C. Bischof and A. Griewank. Tools for the automatic differentiation of computer programs. In *ICIAM/GAMM 95: Issue 1: Numerical Analysis, Scientific*

Computing, Computer Science, pages 267–272, 1996. Special Issue of Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM).
9. C. Bischof, P. Hovland, and P.-T. Wu. Using ADIFOR and ADIC to provide a Jacobian for the SNES component of PETSc. Technical Memorandum ANL/MCS-TM-233, Argonne National Laboratory, 1997.
10. C. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.
11. G. D. Byrne and A. C. Hindmarsh. PVODE, an ODE solver for parallel computers. *Int. J. High Perf. Comput. Appl.*, 13:354–365, 1999.
12. M. C. Ferris, M. Mesnier, and J. J. Moré. NEOS and Condor: Solving optimization problems over the Internet. Preprint ANL/MCS-P708-0398, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1998.
13. P. Fischer. Projection techniques for iterative solution of $Ax = b$ with successive right-hand sides. *Comput. Methods Appl. Mech. Engng.*, 163:193–204, 1998.
14. M. Gertz, 2000. Personal communication.
15. D. Goldfarb and P. L. Toint. Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, 43:69–88, 1984.
16. A. Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.
17. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2000.
18. W. D. Gropp, D. E. Keyes, L. C. McInnes, and M. D. Tidriri. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. Technical Report 98-24, ICASE, Aug. 1998. To appear in *Int. Journal on Supercomputing Applications*.
19. P. Hovland, B. Norris, L. Roh, and B. Smith. Developing a derivative-enhanced object-oriented toolkit for scientific computations. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 129–137. SIAM, October 1998.
20. P. D. Hovland and L. C. McInnes. Parallel simulation of compressible flow using automatic differentiation and PETSc. Technical Report ANL/MCS-P796-0200, Mathematics and Computer Science Division, Argonne National Laboratory, 2000. To appear in a special issue of *Parallel Computing* on "Parallel Computing in Aerospace".
21. S. L. Lee, A. C. Hindmarsh, and P. N. Brown. User documentation for SensPVODE, a variant of PVODE for sensitivity analysis. Technical Report UCRL-MA-140211, Lawrence Livermore National Laboratory, 2000.
22. S. L. Lee and P. D. Hovland. Sensitivity analysis using parallel ODE solvers and automatic differentiation in C: SensPVODE and ADIC. Technical Report ANL/MCS-P818-0500, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.
23. S. Li and L. Petzold. Design of new DASPK for sensitivity analysis. Technical report, University of California at Santa Barbara, 1999.
24. D. P. O'Leary. The block conjugated gradient algorithm and related methods. *Linear Algebra Appl.*, 29:293–322, 1980.