Making Automatic Differentiation Truly Automatic: Coupling PETSc with ADIC

Paul Hovland, Boyana Norris, and Barry Smith

Mathematics and Computer Science Division,
Argonne National Laboratory,
9700 S. Cass Avenue,
Argonne, IL 60439
{hovland, norris, bsmith}@mcs.anl.gov
http://www.mcs.anl.gov/

Abstract. Despite its name, automatic differentiation (AD) is often far from an automatic process. Often one must specify independent and dependent variables, indicate the derivative quantities to be computed, and perhaps even provide information about the structure of the Jacobians or Hessians being computed. However, when AD is used in conjunction with a toolkit with well-defined interfaces, many of these issues do not arise. We describe recent research into coupling the ADIC automatic differentiation tool with PETSc, a toolkit for the parallel numerical solution of PDEs. This research leverages the interfaces and objects of PETSc to make the AD process very nearly transparent.

1 Introduction

Many varieties of scientific computation, including the numerical solution of nonlinear partial differential equations (PDEs), require derivatives. For complicated functions, it can be a difficult task to implement derivative computations by hand. In contrast, finite difference approximations are simple to implement, but they suffer from both roundoff and truncation error. Furthermore, finding a stepsize that balances these sources of error (thus minimizing the total error) can be difficult. Automatic differentiation (AD) [13, 16] offers an alternative that minimizes human effort and eliminates truncation error. For this reason, automatic differentiation has become a popular tool for scientific computing (see, for example [4, 8]).

One obstacle to widespread adoption of automatic differentiation is that the process is often far from automatic. To achieve acceptable levels of performance, the user may need to specify independent and dependent variables, indicate the derivatives to be computed, and provide information about the structure of the Jacobians or Hessians being computed. Previous work [10, 11, 15], however, has demonstrated that when AD is used in conjunction with a toolkit with well-defined interfaces, many of these issues do not arise. This paper describes research into coupling the ADIC [7] automatic differentiation tool with PETSc, a toolkit for the parallel numerical solution of PDEs [3]. This research extends

earlier results by directly exploiting the sparsity structure of the Jacobians to be computed. It also provides a strategy for computing Jacobians in parallel, without requiring an AD capability for MPI. Most important, unlike previous work done in coupling AD with numerical toolkits, the use of ADIC within the PETSc environment is fully automated. Thus, application developers can take full advantage of the increased accuracy and potentially better performance of AD-generated derivatives with no extra effort.

The organization of this paper is as follows. Sections 2 and 3 provide brief introductions to ADIC and PETSc, respectively. Section 4 describes how the two tools have been coupled to provide an automatic differentiation process that is nearly transparent to PETSc users. Section 5 illustrates the potential for increased performance and robustness provided by automatic differentiation. Section 6 summarizes our results and describes opportunities for future work.

2 ADIC

ADIC is a tool for the automatic differentiation of ANSI C. Given a set of functions that compute a mathematical function F, ADIC generates a new set of functions that compute F and its Jacobian, $J = F'$. ADIC differentiates statements by using the so-called reverse mode of automatic differentiation and propagates these partial derivatives from independent to dependent variables by using the so-called forward mode. See [7] for more details on ADIC and its implementation.

The behavior of ADIC can be configured with a large number of user-specified options via one or more control files. A control file contains a set of bindings, expressed as key-value pairs, organized in several sections. Some aspects of the coupling of PETSc and ADIC were handled in control scripts, for example, specifying inactive variables and types, and generating different prefixes for multiple versions of the differentiated code that coexist in the final executable.

A number of enhancements to ADIC were inspired by the need to make the AD process fully automated within PETSc. Some of these include the ability to process multiple control scripts, options for renaming generated header files, a facility for deactivating entire structures, and specialized run-time libraries for scalar-valued gradient accumulations (which arise in the matrix-free case).

3 PETSc

PETSc is an object-oriented toolkit for the parallel numerical solution of PDEs. PETSc provides implementations of basic objects, such as matrices and vectors, facilities for managing data associated with both structured and unstructured meshes (distributed arrays and index sets), linear solvers (primarily Krylov methods with a variety of preconditioners), and nonlinear solvers (primarily Newton-type methods).

3.1 Nonlinear Solvers

PETSc provides a collection of Newton-based nonlinear solvers (SNES). These solvers require a nonlinear function (the discretized PDE) whose input and output are a vector. The prototype for this function is

```
int FormFunction(SNES snes,Vec X,Vec F,void *ptr);
```

The solvers also require a Jacobian, or at least the action of the Jacobian on a vector, but PETSc is able to automatically compute an approximation to the Jacobian (or its action) using finite differences. In cases where the inaccuracy of finite differences leads to a degradation in convergence (see, for example, [14]), it is desirable to use analytic derivatives. In such instances, the user can provide a routine for computing the Jacobian or, using the approach described in Section 4, PETSc and ADIC can generate code for computing the Jacobian automatically.

3.2 Distributed Arrays and Multigrid Algorithms

PETSc provides several objects to assist in the management of data associated with structured and unstructured meshes. One such object is the distributed array (DA), which provides facilities for managing the field data associated with a single structured grid. The DMMG object manages a hierarchical collection of such objects for use in multigrid algorithms. PETSc provides functions (methods) for exchanging data associated with ghost vertices (generalized gather-scatter operations) and for obtaining a coloring suitable for computing a Jacobian using either finite difference approximations or automatic differentiation.

The coloring provided by PETSc is of the Curtis-Powell-Reed (CPR) variety [9]–two columns of the same color contain no row in which both have a nonzero. Thus, the Jacobian can be approximated by perturbing all columns of the same color simultaneously. Alternatively, the seed matrix for automatic differentiation can be initialized by applying the coloring to an identity matrix; all columns (unit vectors) of the same color are combined into a single column of the seed matrix (see [6] for more details). We note that obtaining an optimal or nearly optimal CPR coloring for Jacobians arising from structured grids is simple [12] while efficient parallel algorithms for coloring Jacobians from unstructured meshes remains an open research topic.

4 Coupling PETSc and ADIC

To produce a routine for computing a Jacobian, one might be inclined to apply ADIC directly to the user's `FormFunction` routine (see Section 3.1). A similar strategy has been effective in other contexts [10, 15]. In the context of PETSc, however, this approach is less appealing. One reason is that the user's `FormFunction` routine usually contains a number of calls to PETSc utilities, such as the generalized gather-scatter routines for ghost data communication

described in Section 3.2. Thus, applying ADIC to `FormFunction` would lead to automatic differentiation of these utility methods. However, because many of these utilities deal with problem setup and data movement and often use MPI, applying ADIC directly would likely lead to unnecessary work plus the added complication of differentiating MPI (and including the appropriate runtime support library). For these reasons, we have followed the example of previous semi-automated approaches to coupling ADIC and PETSc [1, 17] and provide an automated solution based on a domain decomposition approach.

4.1 A Domain Decomposition-Based Strategy

Figure 1 provides an example of a simple nonlinear function.[1] This example illustrates a structure common to most nonlinear functions that use PETSc's DA or DMMG objects. A setup and communication phase precedes a computation phase in which the function is evaluated over the local subdomain. A final phase assigns local values to parallel objects. The first and final phase are essentially problem independent and depend only on the structure of the DA or DMMG grid object being used. Therefore, it is possible to isolate the local computation in a separate routine and have PETSc manage the rest of the nonlinear function evaluation. This approach reduces the amount of user code and simplifies the automatic differentiation process. The user provides a function adhering to the following prototype, an example of which appears in Figure 2.

```
int FormFunctionLocal(DALocalInfo *info,Field **x,
                      Field **f,void *ptr);
```

4.2 The User's Experience

As intended, the user interface to the AD-enabled PETSc functionality is virtually the same as the interface used for computing derivatives by means of finite differences. In both cases, the user must provide the nonlinear subdomain function evaluation and initialize the nonlinear solver context. If the user's applications uses DMMG objects, the changes are all minor. To use a nonlinear subdomain function, the user replaces a call such as

```
ierr = DMMGSetSNES(dmmg,FormFunction,0);
```

with a call such as

```
ierr = DMMGSetSNESLocal(dmmg,FormFunctionLocal,0,
           ad_FormFunctionLocal, admf_FormFunctionLocal);
```

To indicate that `FormFunctionLocal` should be differentiated using ADIC, the user adds a comment of the form

[1] This function comes from PETSc SNES example 5, a solid fuel ignition problem derived from the Bratu problem in the MINPACK-2 test problem collection [2].

```
int FormFunction(SNES snes,Vec X,Vec F,void *ptr)
{
  AppCtx       *user = (AppCtx*)ptr;
  int          err,i,j,Mx,My,xs,ys,xm,ym;
  PetscReal    two = 2.0,lambda,hx,hy,hxdhy,hydhx,sc;
  PetscScalar  u,uxx,uyy,**x,**f;
  Vec          localX;

  PetscFunctionBegin;
  err = DAGetLocalVector(user->da,&localX);CHKERRQ(err);
  err = DAGetInfo(user->da,PETSC_IGNORE,&Mx,&My, ... );

  hx     = 1.0/(PetscReal)(Mx-1); hy    = 1.0/(PetscReal)(My-1);
  lambda = user->param;           sc    = hx*hy*lambda;
  hxdhy  = hx/hy;                 hydhx = hy/hx;

  /* Scatter ghost points to local vector, using a 2-step process */
  err = DAGlobalToLocalBegin(user->da,X,INSERT_VALUES,localX);
  CHKERRQ(err);
  err = DAGlobalToLocalEnd(user->da,X,INSERT_VALUES,localX);
  CHKERRQ(err);
  /* Get pointers to vector data */
  err = DAVecGetArray(user->da,localX,(void**)&x);CHKERRQ(err);
  err = DAVecGetArray(user->da,F,(void**)&f);CHKERRQ(err);
  /* Get local grid boundaries */
  err = DAGetCorners(user->da,&xs,&ys,0,&xm,&ym,0);CHKERRQ(err);

  /* Compute function over the locally owned part of the grid */
  for (j=ys; j<ys+ym; j++) {
    for (i=xs; i<xs+xm; i++) {
      if (i == 0 || j == 0 || i == Mx-1 || j == My-1) {
        f[j][i] = x[j][i];
      } else {
        u       = x[j][i];
        uxx     = (two*u - x[j][i-1] - x[j][i+1])*hydhx;
        uyy     = (two*u - x[j-1][i] - x[j+1][i])*hxdhy;
        f[j][i] = uxx + uyy - sc*PetscExpScalar(u);
      }
    }
  }
  /* Restore vectors */
  err = DAVecRestoreArray(user->da,localX,(void**)&x);CHKERRQ(err);
  err = DAVecRestoreArray(user->da,F,(void**)&f);CHKERRQ(err);
  err = DARestoreLocalVector(user->da,&localX);CHKERRQ(err);
  err = PetscLogFlops(11*ym*xm);CHKERRQ(err);
  PetscFunctionReturn(0);
}
```

Fig. 1. Example of a nonlinear function.

```
int FormFunctionLocal(DALocalInfo *info, PetscScalar **x,
                      PetscScalar **f,AppCtx *user)
{
  int         ierr,i,j;
  PetscReal   two = 2.0,lambda,hx,hy,hxdhy,hydhx,sc;
  PetscScalar u,uxx,uyy;

  PetscFunctionBegin;

  lambda = user->param;
  hx     = 1.0/(PetscReal)(info->mx-1);
  hy     = 1.0/(PetscReal)(info->my-1);
  sc     = hx*hy*lambda;
  hxdhy  = hx/hy;
  hydhx  = hy/hx;

  /* Compute function over the locally owned part of the grid */
  for (j=info->ys; j<info->ys+info->ym; j++) {
    for (i=info->xs; i<info->xs+info->xm; i++) {
      if (i == 0 || j == 0 || i == info->mx-1 || j == info->my-1) {
        f[j][i] = x[j][i];
      } else {
        u       = x[j][i];
        uxx     = (two*u - x[j][i-1] - x[j][i+1])*hydhx;
        uyy     = (two*u - x[j-1][i] - x[j+1][i])*hxdhy;
        f[j][i] = uxx + uyy - sc*PetscExpScalar(u);
      }
    }
  }

  ierr = PetscLogFlops(11*info->ym*info->xm);CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```

Fig. 2. Example of a nonlinear subdomain function.

```
/* Process adiC: FormFunctionLocal */
```

With these changes in place, the user can switch between using AD and finite differences using a single runtime option, for example `-dmmg_jacobian_mf_ad_operator` versus `-dmmg_jacobian_mf_fd_operator`. When DA objects are used, the interface for initializing the SNES context in the user's driver routine contains a few minor differences for the AD case, mainly in the methods used for obtaining the coloring for the Jacobian. These differences will eventually disappear as the interfaces continue to evolve.

4.3 Behind the Scenes

While the user's experience is largely unchanged with the new subdomain interface and automatic differentiation capability, several additions to PETSc and ADIC were necessary to make nearly total automation possible.

PETSc was augmented to automatically allocate storage for several derivative objects, principally those associated with the input and output vectors (arrays at the subdomain level). PETSc was also modified to initialize the derivatives associated with the input vector, or so-called seed matrix. The initialization uses the CPR coloring discussed in Section 3.2. The cost of computing the resulting "compressed" Jacobian is proportional to the number of colors. For a structured grid, this is usually the product of the stencil size and the number of data fields at each grid point.

Once the compressed Jacobian has been computed, the values must be assembled into a PETSc sparse matrix object. Support for extracting a row of the compressed Jacobian was added to ADIC, and PETSc was enhanced to be able to assemble the sparse matrix directly from these compressed rows.

To further simplify the task of using automatic differentiation with PETSc, we extended the PETSc build process to include processing of source files with a python script. This script searches for the phrase "Process adiC," extracts the indicated functions into a separate file, processes them with ADIC using the control scripts described in Section 2, and compiles and links the resulting derivative code to provide Jacobian computations in a manner that is virtually transparent to the user.

5 Experimental Results

The potential benefits of using automatic differentiation for derivative computations are illustrated in a simple example from the PETSc distribution, a two-dimensional driven cavity problem using a velocity-vorticity formulation (see `ex19.c` in the SNES tutorials examples directory for more details). We solved a 50×50 version of this problem with derivatives computed using either automatic differentiation or finite difference approximations, the default nonlinear solver, the default finite-difference noise estimate, ILU preconditioning, and several linear solvers (GMRES-k, TFQMR, and BiCGStab). The choice between

automatic differentiation and finite differences was made with the runtime options `-dmmg_jacobian_mf_ad_operator` and `-dmmg_jacobian_mf_fd_operator`. For example, to solve using BiCGStab and automatic differentiation derivatives, we used the following command.

```
ex19 -da_grid_x 50 -da_grid_y 50 -ksp_type bcgs -pc_type ilu \
     -ksp_max_it 500 -dmmg_jacobian_mf_ad_operator
```

Figure 3 illustrates the convergence speed by plotting the nonlinear residual norm at each iteration. The X-axis represents the cumulative elapsed time. The increased accuracy of automatic differentiation results in increased robustness (TFQMR with finite differences fails to converge) and faster convergence.

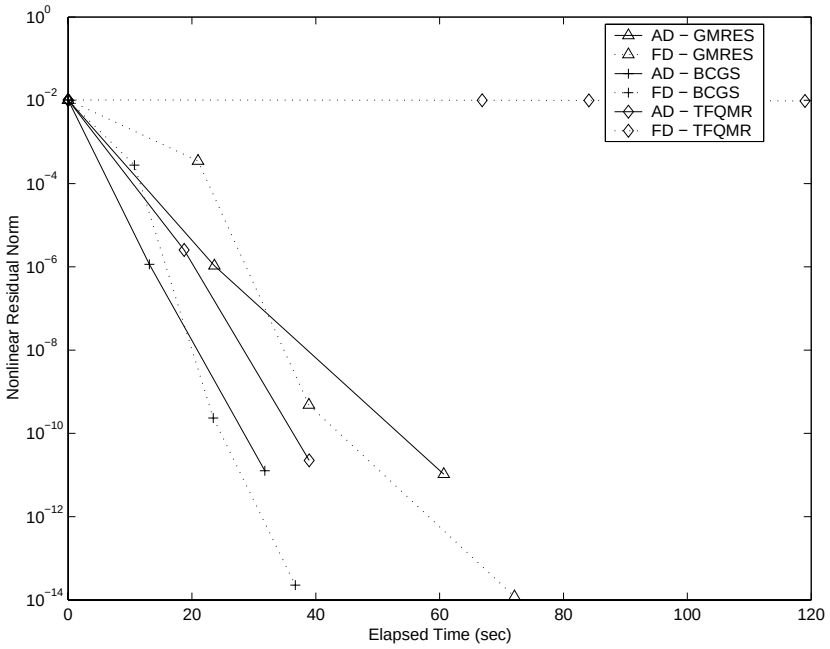

Fig. 3. Performance of AD versus fin for various linear solvers.

6 Conclusions and Future Work

We have presented an integration of automatic differentiation into PETSc, using high-level interfaces to automate fully the use of ADIC to generate the code for computing the Jacobian of a local subdomain function. We described some of the implementation details of this coupling, and we presented a simple application that takes advantage of it. Our experimental results show that frequently the

superior accuracy of AD-generated code leads to convergence in fewer nonlinear and linear iterations than with finite differences.

Future tasks include extending this work to the optimization regime, providing the capability to compute Hessians and gradients of partially separable functions using automatic differentiation. In addition, basic block reverse mode and cross-country preaccumulation strategies could significantly reduce the cost of typical Jacobian computations. The work must also be extended to the case of unstructured meshes. In the case of finite element computations, applying automatic differentiation at the level of element function, and then assembling the full Jacobian from the small, dense element Jacobians may be an effective strategy.

Types, variables, or entire functions that have been specified as inactive in a control file are not augmented with derivative computations. Inactive objects are designated by name, however. In the case of PETSc applications, the user-defined nonlinear function name is not known at the time the ADIC control scripts are created. As shown in Section 4, the user specifies the name of the nonlinear function in a special comment. By extending ADIC to allow *active* functions to be designated via a prototype, and generating derivatives only for those functions (and any other objects designated as active), we could eliminate the need for special pre- and postprocessing of the application and differentiated code.

Acknowledgments

This work 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. Po-Ting Wu performed many of the original experiments in coupling ADIC and PETSc. We thank Gail Pieper for proofreading a draft 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. Preprint ANL/MCS-P818-0500, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2000. Accepted for publication in Proceedings of AD2000.
2. B. M. Averick, R. G. Carter, J. J. Moré, and G.-L. Xue. The MINPACK-2 test problem collection. Preprint MCS-P153-0694, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1992.
3. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 2001.
4. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Philadephia, 1996. SIAM.

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. C. Bischof and P. Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS-TM-158, Argonne National Laboratory, 1991.
7. 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.
8. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation: From Simulation to Optimization*. Computer and Information Science. Springer, New York, 2001.
9. A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.
10. 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.
11. E. M. Gertz, P. E. Gill, and J. Muetherig. User's guide for SnadiOpt: A package adding automatic differentiation to Snopt. Report NA 01–1, Department of Mathematics, University of California, San Diego, 2000.
12. 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.
13. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, 2000.
14. P. D. Hovland and L. C. McInnes. Parallel simulation of compressible flow using automatic differentiation and PETSc. Preprint 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".
15. S. Li and L. Petzold. Design of new DASPK for sensitivity analysis. Technical report, University of California at Santa Barbara, 1999.
16. R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.
17. P. Wu, C. Bischof, and P. D. Hovland. Using ADIFOR and ADIC to provide Jacobians for the SNES component of PETSc. Technical Report ANL/MCS-TM-233, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.