AUTOMATIC DIFFERENTIATION FOR PDES – UNSATURATED FLOW CASE STUDY

George F. Corliss, Christian Bischof,
Andreas Griewank, and Steven J. Wright
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL 60439

Thomas Robey
SPECTRA Research Institute
1613 University Blvd. NE
Albuquerque, NM 87102–1710

Argonne Preprint ANL/MCS-P311-0692

Published in Advances in Computer Methods for Partial Differential Equations – VII, R. Vichnevetski, D. Knight, and G. Richter, Eds., IMACS, New Brunswick, pp. 150-156, 1992.

1 Introduction

The techniques of automatic differentiation [8, 10, 15] are applied to an example partial differential equation arising from the modeling of unsaturated flow. One common paradigm for the numerical solution to some classes of 2-, 3-, or higher-dimensional partial differential equations is:

1. Given a PDE and boundary conditions,
2. apply finite difference or finite element approximations on some appropriate (frequently nonuniform) grid, and
3. enforce an approximate solution by solving a nonlinear system $F(u) = 0$ for the residual by Newton's method.

The dimension of the nonlinear system $F(u) = 0$ is proportional to the number of grid points. In current algorithms, the Jacobian J required by Newton's method is computed by some combination of hand coding, divided differences, matrix coloring, and partial separability. We present a case study documenting the steps we took in analyzing a code for unsaturated flow in porous media for the purpose of computing J by automatic differentiation using ADOL–C [12], a tool for automatic differentiation using overloaded operators in C++. We conclude that

- ADOL–C can be successfully applied to large, complex, existing C codes,
- in this application, the fastest ADOL–C code takes twice as long as the best finite difference code,
- in this application, the reverse mode takes about twice as long as the forward mode, and
- significant efficiencies are possible in the linear system solver.

2 Conventional Methods for Finding J

In this section, we survey briefly some of the conventional methods of computing J.

Hand Coding

The best performance of algorithms is usually achieved when the analytic Jacobian J is hand coded. In general, this is a tedious, time consuming, and error-prone task. In many problems, hand coding is feasible because many of the dependencies of F on u are linear. If the linear dependencies can be separated from the nonlinear ones, hand coding of at least portions of J is made easier.

Divided Differences

The Jacobian J can be approximated by backward, centered, or forward divided differences. An appropriate choice of step size is difficult, especially when there may be differences of scale between different components. Hence, the accuracy of divided difference approximations is in doubt. A naive coding perturbing each component of u in turn is easy, but very expensive. Implementations usually attempt to exploit the sparsity structure which is known to be present.

Matrix Coloring

One technique for exploiting the sparsity structure of J is matrix coloring [3]. If two columns of J have nonzero elements only in disjoint sets of rows, then those two columns can be computed simultaniously, using either divided difference or automatic differentiation techniques. Suppose that J has the sparsity structure

$$Jf = \begin{pmatrix} \clubsuit & & \\ \clubsuit & & \heartsuit \\ & \spadesuit & \heartsuit \\ & \spadesuit \quad \diamondsuit & \\ & \spadesuit \quad \diamondsuit & \end{pmatrix}. \quad \text{Initialize } S = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{to yield } J{*}S = \begin{pmatrix} \clubsuit & \\ \clubsuit & \heartsuit \\ \spadesuit & \heartsuit \\ \spadesuit & \diamondsuit \\ \spadesuit & \diamondsuit \end{pmatrix}$$

at roughly half the cost and storage compared to computing J directly. The number of separate columns of $J * S$ which must be computed is usually of the

order of the number of points in the stencil used to construct the finite element or the finite difference approximation to the PDE. Hence, the chromatic number is often independent of the grid spacing, so the cost of computing J grows linearly with the number of grid points, instead of quadratically as suggested by the dimensions of J.

Partial Separability

A function $f : \mathbf{R}^n \to \mathbf{R}$ is called *partially separable* [13] if it can be expressed as a linear combination $f(u) = \sum f_i(u)$. Often, each f_i depends on only a few of the components of u. Partially separable functions arise frequently in optimization. In the context of PDEs, many residual functions F are partially separable because they are the sum of residuals from various components of the model. When the function F is partially separable, it is much faster to compute the smaller Jacobians of each function f_i separately, and then add then together. Further savings are often possible by coloring the smaller Jacobians with fewer colors than were required for J.

In many PDE applications, including the case studied here, some of functions f_i are linear. Their Jacobians are constant and may be coded analytically or computed once and re-used. Only the Jacobians of the nonlinear f_i's must be re-computed, whether hand coding, divided difference approximation, or automatic differentiation is used.

3 Automatic Differentiation

We illustrate automatic differentiation with an example. Assume that we have the sample program shown in Figure 1 for the computation of a function $f : \mathbf{R}^2 \mapsto \mathbf{R}^2$. Here, the vector `x` contains the independent variables, and the vector `y` contains the dependent variables. The function described by this program is defined except at $\mathtt{x(2)} = 0$ and is differentiable except at $\mathtt{x(1)} = 2$. We can transform the program in Figure 1 into one for computing derivatives by associating a derivative object $\nabla\mathtt{t}$ with every variable `t`. Assume that $\nabla\mathtt{t}$ contains the derivatives of `t` with respect to the independent variables `x`, $\nabla\mathtt{t} = \left(\frac{\partial \mathtt{t}}{\partial \mathtt{x(1)}}, \frac{\partial \mathtt{t}}{\partial \mathtt{x(2)}}\right)^T$. We propagate these derivatives by using elementary differentiation arithmetic based on the chain rule [8, 15] for computing the derivatives of `y(1)` and `y(2)`, as shown in Figure 2. In this example, each assignment to a derivative is actually a vector assignment of length 2.

```
if x(1) > 2 then
   a = x(1) + x(2)

else
   a = x(1) * x(2)

endif
do i = 1, 2

   a = a * x(i)

end do
y(1) = a / x(2)


y(2) = sin (x(2))
```

Figure 1. Sample function $f : \mathrm{x} \mapsto \mathrm{y}$

```
if x(1) > 2.0 then
   a = x(1) + x(2)
   ∇a = ∇x(1) + ∇x(2)
else
   a = x(1) * x(2)
   ∇a = x(2) * ∇x(1) + x(1) * ∇x(2)
endif
do i = 1, 2
   temp = a
   a = a * x(i)
   ∇a = x(i) * ∇a + temp * ∇x(i)
end do
y(1) = a / x(2)
∇y(1) = 1.0 / x(2) * ∇a
        - a / (x(2) * x(2)) * ∇x(2)
y(2) = sin (x(2))
∇y(2) = cos (x(2)) * ∇x(2)
```

Figure 2. Augmented with derivative code

This mode of automatic differentiation, where we maintain the derivatives with respect to the independent variables, is called the *forward mode* of automatic differentiation. The *reverse mode* of automatic differentiation maintains the derivative of the final result with respect to intermediate quantities, usually referred to as *adjoints,* which measure the sensitivity of the final result with respect to some intermediate quantity. The reverse mode requires fewer operations than the forward mode if the number of independent variables is larger than the number of dependent variables. This is exactly the case for computing

a gradient, which can be viewed as a Jacobian matrix with only one row. This issue is discussed in more detail in [8, 11, 12].

Wolfe observed [17], and Baur and Strassen confirmed [1], that if care is taken in handling quantities which are common to the (rational) function and its derivatives, then the cost of evaluating a gradient with n components is a small multiple of the cost of evaluating the underlying scalar function. Despite the advantages of the reverse mode from the viewpoint of complexity, the implementation for the general case is quite complicated. It requires the ability to access *in reverse order* the instructions performed for the computation of f and the values of their operands and results. Current tools (see [14]) achieve this by storing a record of every computation performed. An interpreter performs a backward pass on this "tape." The resulting overhead often dominates the complexity advantage of the reverse mode in an actual implementation (see [5]). We also note that even though we showed the computation only of first derivatives, the automatic differentiation approach can easily be generalized to the computation of univariate Taylor series or Hessians and multivariate higher-order derivatives [2, 9, 15].

This discussion is intended to demonstrate that the principles underlying automatic differentiation are not complicated: We just associate extra computations (which are entirely specified on a statement-by-statement basis) with the statements executed in the original code. As a result, a variety of implementations of automatic differentiation have been developed over the years (see [14] for a survey).

4 Unsaturated Flow Problem

We study a two-dimensional unsaturated flow in a porous medium. Steady state porous media flow involves an elliptic partial differential equation that contains a conductivity coeffecient [6, 16]. The coefficient is typically discontinuous across different materials and can vary greatly. For unsaturated flow, the conductivity is usually taken to be a function of pore pressure which introduces a nonlinearity to the problem. For materials such as tuff, the nonlinearity can become severe enough to dominate the problem as conductivity can change greatly for a small change in pressure. Our interest is in modeling flow in a region consisting of fractured tuff with conductivities that vary by ten or more orders of magnitude, often over very short distances. The code uses a mixed finite element approach with a quasi-Newton iteration to handle the very high nonlinearity. The nonlinear equations are contained in the C function `dual(x,f)`. Centered differences are used to calculate a very sparse 1989×1989 Jacobian J. The resulting linear equation is solved by a bi-conjugate gradient algorithm.

We approached the code hoping to demonstrate the superiority of the ADOL–C [12] implementation of automatic differentiation over the centered difference approximations used in Robey's original code. The high degree of nonlinearity

was felt to be a potential cause of inaccuracy using centered differences, and we hoped that automatic differentiation could improve accuracy and speed.

The test problem considered here is a 1-D test problem exhibiting a particularly simple structure. We hope to develop strategies that generalize to higher-dimensional problems of practical interest.

5 Exploitation of Structure

It is well known that J has a very regular sparse structure arising from the underlying discretization grid (see Figure 3).

$$J = \begin{pmatrix} B & \hat{D} & C_1^T & C_2^T \\ D & & & \\ C_1 & & 0 & \\ C_2 & & & \end{pmatrix}$$

Figure 3. Sparsity structure of J

The matrix $B = J(0..935, 0..935)$ is block diagonal. The diagonal blocks are 4×4 blocks of the form

$$\alpha \cdot \begin{pmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{pmatrix}.$$

The matrix $D = J(936..1286, 0..935)$ is built of 3×8 blocks along the slanted diagonal. The slanted diagonal has slope $3/8$. The matrix $\hat{D} = J(0..935, 936..1286)$ is equal to D^T in the limit as the nonlinear perturbation approaches zero. It is $\hat{D}$ that will be our focus in computing J. The matrix $C_1 = J(1287..1519, 0..935)$ has four slanted diagonals, each with slope $1/4$. The upper two diagonals have values -2, while the lower two diagonals have values $+2$. The matrix $C_2 = J(1519..1988, 0..935)$ has slanted diagonals with slopes $1/2$ and values ± 1.

Robey recognized that f depends only linearly on x, except for the dependence of $f_{0..935}$ on $x_{936..1286}$. That is, he coded most of the elements of J as linear functions of x. Only the elements of J that belong to $\hat{D}$ are more complicated to compute. In principle, the elements of $\hat{D}$ could be computed analytically since they involve only sums and products of components of x. This was not done because it is too hard to recognize and code the patterns of which components of x impact which components of f. The 351 rows of $\hat{D}$ can be computed in only six passes. The combination of partitioning J and coloring $\hat{D}$ reduced the time required to approximate J from about 31 minutes to about 5.78 seconds on a SPARC 1+. This is the code that formed the basis for the further explorations described below.

6 Conversion to ADOL–C

In this section, we describe the steps involved in converting the original code to generate J using ADOL–C. More details of the conversion can be found in [4].

6.1 Step 1: Convert to C++

The program `unsat` was first converted to run with the GNU G++ implementation of C++. The following modifications were necessary:

- Remove system-dependent graphics capabilities that had no significance to the mathematical problems of computing derivatives and solving a system of linear equations.
- Convert all function headers from their acceptable C form

```
int step(x,s)
double *x,*s;
```

  to a form acceptable to C++

```
int step (double *x, double *s)
```

In addition, some diagnostic print statements were removed, some were added, and system-dependent timing instrumentation was added. The resulting code required 5.78 seconds on a SPARC 1+ to evaluate the nonlinear part of the Jacobian $\hat{D}$.

Study of the structure of $\hat{D}$ suggested that it could be computed with three colors instead of the six colors used by Robey. Using three colors reduced the time required to compute $\hat{D}$ from 5.78 seconds to 2.87 seconds (see Table 1).

6.2 Step 2: Convert the Function `dual` to Use Type `adouble`

Now we were ready to explore the use of automatic differentiation. In the unsaturated flow code, the function to be differentiated is isolated in `int dual (double *xv, double *r)` which calls `double konduct (double *xv, int elem)`. In `dual`, `xv` contains the independent variables, and `r` contains the dependent variables. Both `dual` and `konduct` are called from several places in the code, so we needed to leave the original functions, while providing new ones called `adual` and `akonduct` to be called from `step` to compute the Jacobian. In ADOL–C, independent variables, dependent variables, and any other variables requiring derivative objects must be declared as type `adouble`. In each function, some of the `double` variables require derivatives, while others do not.

No changes of any kind were required to the body of either function. However, we removed from `adual` code that is required to compute the value of f but that is not required to compute the elements of $\hat{D}$, which require only `r[0..935]`.

6.3 Step 3: Record the "tape"

The next step was to modify the three-color finite difference code in `step3.c` to use automatic differentiation instead. We followed the instructions in [12] first for the forward mode of automatic differentiation.

We removed the finite difference code from `step3.c` as shown in Figure 4.

```
/* Nonlinear part */
  stepsize=1.0e-7;
  for (i=0;i<3;i++) {
      for (j=0;j<elements;j++) {
          deltax[j]=(fabs(xv[8*elements+3*j+i])>1.0)
               ? stepsize*fabs(xv[8*elements+3*j+i])
               : stepsize;
          xv[8*elements+3*j+i]+=deltax[j];
      }

      k=dual(xv,r);

      if (k<0) return(-2);

      for (j=0;j<8*elements;j++) df[j]=r[j];
      for (j=0;j<elements;j++)
        xv[8*elements+3*j+i]=x[8*elements+3*j+i]
                             -deltax[j];

      k=dual(xv,r);

      if (k<0) return(-2);

      /* Store Jacobian in a sparse structure. */

   }  /* end for (i  */
   for (i=0;i<dim;i++) xv[i]=x[i];
```

Figure 4. Jacobian by finite differences

We added include files:

```
#include "adouble.h"
#include "adutils.h"
```

We replaced the finite difference code with code to do the following:

1. Declare variables for ADOL–C.
2. Insert calls to `trace_on` and `trace_off` to mark the active section of the code.
3. Nominate independent variables.

4. Call `adual` within the active section to record the "tape". The function value is computed at this point.

5. Nominate dependent variables.

6. Make three passes in the forward mode:

1. Initialize independent and dependent derivative objects.

2. Call forward.

3. Extract derivatives.

The derivative values computed by ADOL–C as shown in Figure 5 were extracted from `Depend_Y` and stored in the original data structure for J. The resulting code required 5.53 seconds to evaluate $\hat{D}$, or twice as long as the three-color finite-difference code.

```
unsigned short Tape_Tag = 1;
int Keep     = 0;
int degree   = 1;
double **Indep_X  = new double*[dim];
double **Depend_Y = new double*[dim];
adouble ad_xv[dim];
adouble ad_r[dim];
int adual (adouble *, adouble *);

for (j = 0; j < dim; j ++) {
   Indep_X[j]  = new double[2];
   Depend_Y[j] = new double[2];
}
/* Compute right hand side vector f */
f=(double *) calloc(dim,sizeof(double));
if (f==NULL) return(-1);

trace_on (Tape_Tag, Keep);
for (i = 0; i < dim; i ++) {
   if ((i <= 935) || (1287 <= i)) {
      ad_xv[i] = x[i];
   }
   else {
      // Nominate independent variables
      ad_xv[i] <<= x[i];
   }
}

k = adual (ad_xv, ad_r);

if (k<0) return(-2);
```

```
  for (i = 0; i < dim; i ++) {
     if (i < 8*elements) {
        // Nominate dependent variables
        ad_r[i] >>= f[i];
     }
     else {
        f[i] = value (ad_r[i]);
     }
  }
  trace_off ();

/* Nonlinear part */
   for (i = 0; i < 3; i ++) {
      for (j = 0; j < 351; j ++) {
         Indep_X[j][0]  = x[8*elements+j];
         Indep_X[j][1]  = 0.0;
      }
      for (j = 0; j < 8*elements; j ++) {
         Depend_Y[j][0] = 0.0;
         Depend_Y[j][1] = 0.0;
      }
      for (j=0;j<elements;j++) {
           Indep_X[3*j+i][1] = 1.0;
      }

      forward (Tape_Tag, 8*elements, 351, degree,
               Keep, Indep_X, Depend_Y);

      /* Store Jacobian in a sparse structure. */

    }  // end for (i
```

Figure 5. Jacobian by 3-color, forward-mode ADOL–C

6.4 Reverse Mode

ADOL–C can also evaluate derivatives in the reverse mode. The reverse mode is usually faster than the forward mode when there are more independent variables than there are dependent variables. The entire Jacobian is square, but the block $\hat{D}$ that we are computing is composed of 3×8 blocks. This configuration implies that the forward mode (or finite differences) can be computed in three passes, while the reverse mode requires eight passes. We write three versions of `step` using the reverse mode. None of these versions was as fast as the three forward sweeps.

`step6.c`: Eight reverse sweeps. Similar to the three forward sweeps.

`step2.c`: Eight-vector reverse. The eight reverse sweeps are all performed at once.

`step7.c`: Eight-vector short reverse. The eight reverse sweeps are all performed at once, taking advantage (as in the three forward sweeps) of the fact that we do not need to differentiate with respect to all `x`, nor are we required to differentiate all dependent variables.

7 Results

Table 1 gives the timing comparisons of the various versions of `step` described above. These are the times in seconds on a SPARC 1+ required to compute $\hat{D}$, the nonlinear portion of the Jacobian J. The "tape" for the three-color forward mode evaluation was 1.5 mega-bytes long.

In general, the derivatives computed by automatic differentiation are more accurate than those computed by finite differences. In some applications, the improved accuracy enables Newton's method to converge in fewer iterations.

Table 1. CPU Times for Jacobian computation

Method	Seconds
Six-color finite differences	5.78
Three-color finite differences	2.87
Three-color forward mode	5.53
Eight sweeps of reverse mode	18.78
Eight-vector reverse mode	11.15
Eight-vector short reverse mode	11.12

8 Conclusions about Automatic Differentiation

- ADOL–C can be applied to existing C codes that are large and complicated enough to have real scientific interest.
- In this application, the fastest ADOL–C code takes twice as long as the best finite difference code.
- In this application, the reverse mode takes about twice as long as the forward mode, while it must perform nearly three times as many sweeps (8 vs. 3).
- In this application, the vector reverse is about 1/3 faster than the corresponding number of reverse sweeps performed separately.

- Recognizing that short vectors can be used for the independent and the dependent variables saves only an insignificant amount of time, but it is more complicated to code.

9 Linear Equation Solver

In truth, we have been looking at the wrong problem so far. It takes less than 3 seconds to compute the nonlinear part of J, but it takes up to 430 seconds to solve the system of linear equations. The original intent was to explore the application of ADOL–C, but we also pass along observations about linear equation solvers. The existing code stores J as a sparse matrix and solves the linear equation to find the Newton step using a biconjugate gradient iterative algorithm.

One alternative is to use the general direct sparse solver. Another alternative is to take better advantage of the structure of J, putting J into a banded form by suitable interchanges of rows and columns so that a banded solver can be used.

The problems we are really interested in solving are 2-D problems. The bands described in Section 5 do not generalize to 2-D problems. While both B and D are banded in the 2-D problem, C is not. The variation of condutivities is greater for 2-D problems than 1-D problems due to the increased dimension and flow paths.

The Jacobian of this problem is rank deficient due to the form of the flux boundary conditions. The rank defieciency is caused by not being able to specify the pressures at the flux boundaries. The problems of real interest are not necessarily rank deficient. However, the rapid changes in conductivities can cause poor conditioning of the Jacobian or possibly rank deficiency. One can handle rank deficiency by adding some constraints to uniquely define a solution. Alternatively, one should take into account the suggestions of Griewank [7] on the behavior of Newton's method and its variation for singular systems. Two different situations must be distinguished. In the first case, there is (locally) a smooth solution manifold of dimension p, and the rank of the Jacobian drops by exactly p at the solutions. In that case, Newton's method and variations have been observed to converge quite rapidly in terms of the residual norm, even though the iterates may wander up and down the solution manifold a bit. In the second case, when the rank drop of the Jacobian exceeds the dimension of the (largest) solution manifold, the situation is completely different. For any fixed point iteration of the form $x_{\text{new}} = G\left[x_{\text{old}}, f(x_{\text{old}})\right]$ with $f = 0$, the algebraic system being solved converges from almost all starting points sublinearly if G is differentiable with respect to the residual vector f. The only way to maintain at least linear convergence is to use Newton's method without bounding the inverse or to append the linear system by equations that enforce singularity. (R-sublinear convergence means that the k-th root of the k-th residual norm

tends to one in theory. In practice, that amounts to the iteration's stalling completely.)

Another approach to improving the performance of the linear equation solver is to apply a suitable preconditioner. Most simple preconditioners require either a positive definite matrix or diagonal dominance which do not apply to this problem. Work on implementing a more complicated preconditioner that takes advantage of the structure of the Jacobian is in progress.

References

[1] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[2] Bruce D. Christianson. Automatic Hessians by reverse accumulation. Technical Report NOC TR228, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K., April 1990.

[3] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187–209, 1984.

[4] George Corliss, Andreas Griewank, Tom Robey, and Steve Wright. Automatic differentiation applied to unsaturated flow — ADOL–C case study. Technical Memorandum ANL/MCS–TM–162, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., April 1992.

[5] Lawrence C. W. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 114–125. SIAM, Philadelphia, Penn., 1991.

[6] Richard Ewing and Mary Wheeler. Computational aspects of mixed finite element methods. In R. Stepleman et. al., editor, *Scientific Computing*. IMACS/North-Holland Publishing Comapany, 1983.

[7] Andreas Griewank. On solving nonlinear equations with simple singularities or nearly singular solutions. *SIAM Review*, 27(4):537–563, December 1985.

[8] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989. Also appeared as Preprint MCS–P10–1088, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., October 1988.

[9] Andreas Griewank. Automatic evaluation of first- and higher-derivative vectors. In R. Seydel, F. W. Schneider, T. Küpper, and H. Troger, editors, *Proceedings of the Conference at Würzburg, Aug. 1990, Bifurcation and Chaos: Analysis, Algorithms, Applications*, volume 97, pages 135–148. Birkhäuser Verlag, Basel, Switzerland, 1991.

[10] Andreas Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24, May & July 1991. No. 3, p. 20 & No. 4, p. 8.

[11] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, to appear. Also appeared as Preprint MCS–P228–0491, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991.

[12] Andreas Griewank, David Juedes, Jay Srinivasan, and Charles Tyner. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, to appear. Also appeared as Preprint MCS–P180–1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1990.

[13] Andreas Griewank and Ph. L. Toint. On the unconstrained opimization of partially separable functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, NATO Conference Series II: Systems Science, pages 301–321. Academic Press, New York, 1982.

[14] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329. SIAM, Philadelphia, Penn., 1991. Also appeared as Preprint MCS–P265–0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., October 1991.

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

[16] Mary Wheeler and Ruth Gonzalez. Mixed finite element methods for petroleum reservoir engineering problems. In R. Glowinski and J.-L. Lions, editors, *Computing Methods in Applied Sciences and Engineering, VI*. Elsevier, 1984.

[17] Philip Wolfe. Checking the calculation of gradients. *ACM Trans. Math. Softw.*, 6(4):337–343, 1982.