Automatic Differentiation Applied to Unsaturated Flow — ADOL–C Case Study

by

George Corliss, Andreas Griewank, Thomas Robey, and Steve Wright

Abstract

We have experimented with many variants of the code `dual.c` for two-dimensional unsaturated flow in a porous medium. The goal has been to speed up the evaluation of derivatives required for a Newton iteration. We have primarily investigated the use of ADOL–C, a C++ tool for automatic differentiation and have come to the following conclusions:

- Three colors suffice for computing the nonlinear portion of the Jacobian. That speeds up the Jacobian evaluation in the original code by a factor of two.
- The use of ADOL–C for automatic differentiation does not speed up the code. The best result we have achieved for automatic differentiation takes twice as long as the original centered difference approximation.
- The derivative values computed by ADOL–C are more accurate than the centered difference approximations.
- We can realize *big* savings in the linear equation solver.

1 Purpose

The purpose of this report is to document the steps we took in analyzing a code for unsaturated flow in porous media for the purpose of applying automatic differentiation and of speeding up the execution of the code.

2 Unsaturated Flow Problem

The problem is a two-dimensional unsaturated flow in a porous medium. The intended application is 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 is written in C and uses a mixed finite element approach with a quasi-Newton iteration to handle the very high nonlinearity. The nonlinear equations are contained in the subroutine `adual(x,f)`. Centered differences were used to calculate a very sparse 1989×1989 Jacobian J. The resulting linear equation was solved by a bi-conjugate gradient algorithm.

We approached the code hoping to demonstrate the superiority of the ADOL–C [3] 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.

3 First Attempts

The initial experiments with the first version of the `dual.c` code underscored the regularity of the structure of the Jacobian matrix, a fact that eventually led to major performance improvements.

ADOL–C supports both the forward and the reverse modes (see [2]) of automatic differentiation. It is not clear which method should be preferred for computing the square Jacobian required for this problem.

4 Exploitation of Structure

It is well known that J has a very regular sparse structure arising from the underlying discretization grid (see Figure 1).

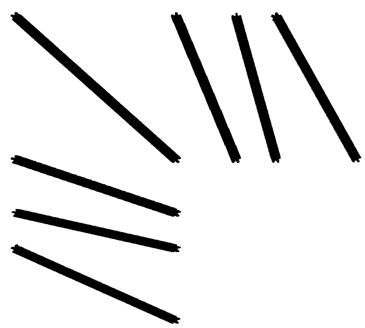

Figure 1. Sparsity structure of J

The Jacobian has the form

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

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.

Robey also recognized that the 351 rows of $\hat{D}$ could 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. This code is available by anonymous ftp from `boris.mscs.mu.edu` (134.48.4.4) in subdirectory `pub/corliss/Robey/Feb_04`.

5 Conversion to ADOL–C

Griewank and Corliss were interested in `dual.c` as an example to demonstrate ADOL–C [3]. ADOL–C is an automatic differentiation tool using overloaded operators in C++. In this section, we describe the steps involved in converting the original `unsat.c` code to generate J using ADOL–C.

5.1 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, and some were added, system-dependent timing instrumentation was added.

The resulting code ran, appeared to give correct answers, and required 5.78 seconds on a SPARC 1+ to evaluate the nonlinear part of the Jacobian $\hat{D}$. This version of the code is in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_19`.

Study of the structure of $\hat{D}$ suggested that it could be computed with three colors instead of the six colors used by Robey. In function `step`, we eliminated the `for (l=0; ... )` loops and the `if (redblack[j] ... )` tests. This is file `step3.c` in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_19`. Using three colors reduced the time required to compute $\hat{D}$ from 5.78 seconds to 2.87 seconds.

5.2 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)
{
  int i,j,k;
  double kinv,xdelta,ydelta,norm,t,p0,p1,s[3];
  JENTRY *cptr;
```

which calls

```
extern double *yc,sm,sf;

double konduct (double *xv, int elem)
{
  int i,j,m,gp;
  double p,rho,s,t,gamma,y,kc,kminv,kfinv;
```

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, all variables requiring derivative objects, including all independent variables and all dependent variables, must be declared as type `adouble`. In each function, some of the `double` variables require derivatives, while others do not. Also in `konduct`, the global variables `sm` and `sf` must be of type `adouble`, but they do not really need to be global. Hence, the new headers are as follows:

```
#include "adouble.h"

int adual (adouble *xv, adouble *r)
{
  int     i,j,k;
  double  xdelta,ydelta,norm,t,p0,p1,s[3];
  adouble kinv;
  JENTRY *cptr;
  adouble akonduct (adouble *xv, int elem);
```

and

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

extern double *yc;

adouble akonduct (adouble *xv, int elem)
{
  int     i, j, m, gp;
  double  gamma, y;
  adouble p, rho, s, t, kc, kminv, kfinv, sm, sf;
```

No changes of any kind were required to the body of either function. We did, however, remove 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]`. The resulting code is in files `adual.c` and `akonduct.c` in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_14`.

5.3 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 [3] first for the forward mode of automatic differentiation.

We removed the finite difference code from `step3.c`:

```
/* 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);
      for (j=0;j<elements;j++) {
          for (k=0;k<8;k++) {
            df[8*j+k]=(df[8*j+k]-r[8*j+k])/(2.0*deltax[j]);
            if (df[8*j+k] != 0.0) {
              jnptr = (JENTRY *) calloc(1,sizeof(JENTRY));
              if (jnptr == NULL)
                return(-1);
              jnptr->col   = 8*elements+3*j+i;
              jnptr->value = -df[8*j+k];
              jnptr->next  = NULL;
              if (row[8*j+k] == NULL)
                row[8*j+k] = jnptr;
              else {
                jptr = row[8*j+k];
                while (jptr->next != NULL)
                  jptr = jptr->next;
                jptr->next = jnptr;
              }  /* end if (row  */
            }  /* end if (df  */
          }  /* end for (k  */
      }  /* end for (j  */
  }  /* end for (i  */
  for (i=0;i<dim;i++)
      xv[i]=x[i];
```

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:
   (a) Initialize independent and dependent derivative objects.
   (b) Call forward.
   (c) Extract derivatives.

The derivative values computed by ADOL–C were extracted from `Depend_Y` and stored into the original data structure for J.

```
{  // (Should be unnecessary) open ADOL-C block

  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 {
        ad_xv[i] <<= x[i];  // Nominate ADOL-C independent variables
     }
  }

  k = adual (ad_xv, ad_r);
```

```
  if (k<0)
    return(-2);
  for (i = 0; i < dim; i ++) {
     if (i < 8*elements) {
        ad_r[i] >>= f[i];  // Nominate ADOL-C dependent variables
     }
     else {
        f[i] = value (ad_r[i]);
     }
  }
  trace_off ();

   times(buffer);
   Time_Begin_Step = buffer->tms_utime;

/* 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);

      for (j=0;j<elements;j++) {
          for (k=0;k<8;k++) {
            df = Depend_Y[8*j+k][1];
            if (df != 0.0) {
              jnptr = (JENTRY *) calloc(1,sizeof(JENTRY));
              if (jnptr == NULL)
                return(-1);
              jnptr->col   = 8*elements+3*j+i;
              jnptr->value = -df;
              jnptr->next  = NULL;
              if (row[8*j+k] == NULL)
                row[8*j+k] = jnptr;
              else {
                jptr = row[8*j+k];
                while (jptr->next != NULL)
                  jptr = jptr->next;
                jptr->next = jnptr;
              }  // end if (row
            }  // end if (df
          }  // end for (k
      }  // end for (j
    }  // end for (i
}  // (Should be unnecessary) close ADOL-C block
```

Warning: The declarations and magic numbers are explicitly tailored for the input file `flux.in`. For other data, the structure and size of the Jacobian must be re-examined.

The resulting code is in file `step4.c` in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_14`. It required 5.53 seconds to evaluate $\hat{D}$, or twice as long as the three-color finite difference code.

5.4 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:

`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.

The code may be found in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_14`. Since none of these versions was as fast as the three forward sweeps, we omit the code here, but we include the relevant portions of the code as appendixes to serve as examples for programming the reverse mode.

6 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.

Table 1. CPU Times for Jacobian computation

Method	File	Seconds
6-color finite differences	unsat	5.78
3-color finite differences	unsat3	2.87
3-color forward mode	unsat4	5.53
8 sweeps of reverse mode	unsat6	18.78
8-vector reverse mode	unsat2	11.15
8-vector short reverse mode	unsat7	11.12

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.

7 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.

8 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 written in C available from `netlib`. (Send a message "send index from sparse" to `netlib@ornl.gov`.) Another alternative is to take better advantage of the structure of J.

Wright observed that a matrix with the regularity of the structure of J can be put into a banded form by suitable interchanges of rows and columns. The numbers of rows of B, D, C_1, and C_2 are 936, 351, 232, and 470, respectively. These are almost exactly in the ratios $8:3:2:4$. (That is why we cited the slopes of the slanting diagonals in Section 4.) The ratios would be exactly $8:3:2:4$ if the number of rows were 936, 351, 234, and 468, respectively.

To transform J into a banded matrix, we take 8 rows from B, then 3 rows from D, then 2 rows from C_1, then 4 rows from C_2, and then repeat. The columns are reordered in the same way. To make things come out right, we take only one row of C_1 and 5 rows of C_2 on the first and last passes. The program given in Appendix D (in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_19/mapindex.c`) sets up an array to map the old indices to the new ones, reads from standard input J in an i, j, $J_{i,j}$ format, and writes to standard output J in a similar format, except that rows and columns have been interchanged to make it banded. The half-band width is 28.

In principle, one could modify `step` to immediately store J in a banded form.

Once one has J in a banded form, there are at least two alternatives:

1. Continue to use `bicon`. It should converge much faster when applied to a banded matrix.
2. Use a banded solver.

We strongly recommend the second alternative. We anticipate that a direct banded solver will be *much* faster than a general iterative solver like `bicon`. One such banded solver can be found in `bandtest.f` (test file) and `tom_band.f` (reads the output of `mapindex` and solves) in `boris.mscs.mu.edu:pub/corliss/Robey/Feb_19`.

The problems we are really interested in solving are 2-D problems. The bands described in Section 4 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 [1] 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 1 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.

Appendix A. `Step6.c` – Eight Reverse Sweeps

This appendix lists the portion of the code in `step` which computes D using eight reverse sweeps.

```
{  // (Should be unnecessary) open ADOL-C block

  unsigned short Tape_Tag = 1;
  int Keep     = 1;
  int degree   = 0;
  double *Weight_U  = new double[dim];   // Weight matrix
  double **Depend_Y = new double*[dim];  // Result adjoints
  adouble ad_xv[dim];
  adouble ad_r[dim];
  int adual (adouble *, adouble *);

  for (i = 0; i < dim; i ++) {
     Depend_Y[i] = new double[degree+1];
  }
  /* 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 ++) {
     ad_xv[i] <<= xv[i];  // Nominate ADOL-C independent variables
  }
  k = adual (ad_xv, ad_r);
  if (k<0)
    return(-2);
  for (i = 0; i < dim; i ++) {
      ad_r[i] >>= f[i];  // Nominate ADOL-C dependent variables
  }
  trace_off ();

   times(buffer);
   Time_Begin_Step = buffer->tms_utime;

/* Nonlinear part */
  for (k = 0; k < 8; k ++) {
     for (i = 0; i < dim; i ++) {
        Weight_U[i] = 0.0;
        Depend_Y[i][0] = 0.0;
     }

     for (j = 0; j < elements; j ++) {
        Weight_U[8*j+k] = 1.0;
     }

     reverse (Tape_Tag, dim, dim, degree, Weight_U, Depend_Y);

       for (i=0;i<3;i++) {
         for (j=0;j<elements;j++) {
            df = Depend_Y[8*elements+3*j+i][0];
            if (df!=0.0) {
```

```
              jnptr=(JENTRY *) calloc(1,sizeof(JENTRY));
              if (jnptr==NULL)
                return(-1);
              jnptr->col=8*elements+3*j+i;
              jnptr->value=-df;
              jnptr->next=NULL;
              if (row[8*j+k]==NULL)
                row[8*j+k]=jnptr;
              else {
                jptr=row[8*j+k];
                while (jptr->next!=NULL)
                  jptr=jptr->next;
                jptr->next=jnptr;
              }  // end if (row
            }  // end if (df
         }  // end for (j
       }  // end for (i
    }  // end for (k
}  // (Should be unnecessary) close ADOL-C block
```

Appendix B. `Step2.c` – Eight-Vector Reverse Mode

This appendix lists the portion of the code in **step** which computes D using one reverse sweeps consisting of eight vectors.

```
{  // (Should be unnecessary) open ADOL-C block

  unsigned short Tape_Tag = 1;
  int Keep     = 1;
  int degree   = 0;
  double **Weight_U  = new double*[8];   // Weight matrix
  double ***Depend_Y = (double ***) new double**[8];   // Result adjoints
  adouble ad_xv[dim];
  adouble ad_r[dim];
  int adual (adouble *, adouble *);

  for (k = 0; k < 8; k ++) {
     Weight_U[k] = new double[dim];
     Depend_Y[k] = new double*[dim];
     for (i = 0; i < dim; i ++) {
        Weight_U[k][i] = 0.0;
        Depend_Y[k][i] = new double[degree+1];
        Depend_Y[k][i][0] = 0.0;
     }
  }
  for (j = 0; j < elements; j ++) {
     for (k = 0; k < 8; k ++)
        Weight_U[k][8*j+k] = 1.0;
  }  // end for (j
  /* 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 ++)
  {
     ad_xv[i] <<= xv[i];  // Nominate ADOL-C independent variables
  }
  k = adual (ad_xv, ad_r);
  if (k<0)
    return(-2);
  for (i = 0; i < dim; i ++)
  {
      ad_r[i] >>= f[i];  // Nominate ADOL-C dependent variables
  }
  trace_off ();

   times(buffer);
   Time_Begin_Step = buffer->tms_utime;

/* Nonlinear part */
  reverse (Tape_Tag, dim, dim, 8, degree, Weight_U, Depend_Y);

    for (i=0;i<3;i++) {
```

```
      for (j=0;j<elements;j++) {
          for (k=0;k<8;k++) {
             df = Depend_Y[k][8*elements+3*j+i][0];
             if (df!=0.0) {
              jnptr=(JENTRY *) calloc(1,sizeof(JENTRY));
              if (jnptr==NULL)
                return(-1);
              jnptr->col=8*elements+3*j+i;
              jnptr->value=-df;
              jnptr->next=NULL;
              if (row[8*j+k]==NULL)
                row[8*j+k]=jnptr;
              else {
                jptr=row[8*j+k];
                while (jptr->next!=NULL)
                  jptr=jptr->next;
                jptr->next=jnptr;
              }  // end if (row
            }  // end if (df
          }  // end for (k
      }  // end for (j
    }  // end for (i
}  // (Should be unnecessary) close ADOL-C block
```

Appendix C. `Step7.c` – Eight-Vector Short Reverse Mode

This appendix lists the portion of the code in **step** which computes D using one reverse sweeps consisting of eight vectors shortened to take advantage of the fact that we are only computing a portion of J.

```
{  // (Should be unnecessary) open ADOL-C block

  unsigned short Tape_Tag = 1;
  int Keep      = 1;
  int degree    = 0;
  double **Weight_U  = new double*[8];   // Weight matrix
  double ***Depend_Y = (double ***) new double**[8];   // Result adjoints
  adouble ad_xv[dim];
  adouble ad_r[dim];
  int adual (adouble *, adouble *);

  for (k = 0; k < 8; k ++) {
     Weight_U[k] = new double[dim];
     Depend_Y[k] = new double*[dim];
     for (i = 0; i < dim; i ++) {
        Weight_U[k][i] = 0.0;
        Depend_Y[k][i] = new double[degree+1];
        Depend_Y[k][i][0] = 0.0;
     }
  }
  for (j = 0; j < elements; j ++) {
     for (k = 0; k < 8; k ++)
        Weight_U[k][8*j+k] = 1.0;
  } // end for (j
  /* 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) || (1290 <= i)) {
        ad_xv[i] = xv[i];
     }
     else {
        ad_xv[i] <<= xv[i];  // Nominate ADOL-C independent variables
     }
  }

  k = adual (ad_xv, ad_r);

  if (k<0)
    return(-2);
  for (i = 0; i < dim; i ++) {
     if (i < 8*elements) {
        ad_r[i] >>= f[i];  // Nominate ADOL-C dependent variables
     }
     else {
        f[i] = value (ad_r[i]);
```

```
    }
  }
  trace_off ();

   times(buffer);
   Time_Begin_Step = buffer->tms_utime;

/* Nonlinear part */
  reverse (Tape_Tag, 8*elements, 354, 8, degree, Weight_U, Depend_Y);

    for (i = 0; i < 3; i ++) {
      for (j = 0; j < elements; j ++) {
          for (k = 0; k < 8; k ++) {
            df = Depend_Y[k][3*j+i][0];
            if (df != 0.0) {
              jnptr = (JENTRY *) calloc(1,sizeof(JENTRY));
              if (jnptr == NULL)
                return(-1);
              jnptr->col   = 8*elements+3*j+i;
              jnptr->value = -df;
              jnptr->next  = NULL;
              if (row[8*j+k] == NULL)
                row[8*j+k] = jnptr;
              else {
                jptr = row[8*j+k];
                while (jptr->next != NULL)
                   jptr = jptr->next;
                jptr->next = jnptr;
              }  // end if (row
            }  // end if (df
          }  // end for (k
      }  // end for (j
    }  // end for (i
}  // (Should be unnecessary) close ADOL-C block
```

Appendix D. `Mapindex.c` – Row and Column Interchanges

```
/*  File:  MAPINDEX.c              20-FEB-1992

/*  Purpose:  Map index to transform Robey's Jacobian into a
/*            banded structure.
/*  Author:   George Corliss, Argonne National Labs, 19-FEB-1992.
*/

/*  Map the original Jacobian into a banded structure for faster solution.
/*  There are 4 blocks of columns:  0..935, 936..1286, 1287..1518, and
/*  1519..1988.  We take columns in proportions: 8, 3, 2, 4, except that
/*  the block 1287..1518 is 2 columns short (hence initial and tail
/*  stages take only 1 column each), and the block 1287..1518 is 2 columns
/*  to long (hence initial and tail stages take 5 columns each).  There
/*  are 3 stages in defining the mapping: initial, body, and tail.

       Original   New
    Block 1:
          0         0
         ..        ..
          7         7
      j = 1
          8        17
         ..        ..
         15        25
      ...
      j = 115
        920      1955
         ..        ..
        927      1962

        928      1972
         ..        ..
        935      1979

     Block 2:
        936         8
        937         9
        938        10
      j = 1
        939        25
        940        26
        941        27
      . . .
      j = 115
       1281      1963
       1282      1964
       1283      1965

       1284      1980
       1285      1981
       1286      1982

     Block 3:
```

```
      ==========

       1287        11
      j = 1
       1288        28
       1289        29
      . . .
      j = 115
       1516      1966
       1517      1967

       1518      1983

     Block 4:
      ==========

       1519        12
       1520        13
       1521        14
       1522        15
       1523        16
      j = 1
       1524        30
        ..         ..
       1527        33
      . . .
      j = 115
       1980      1968
        ..         ..
       1983      1971

       1984      1984
       1985      1985
       1986      1986
       1987      1987
       1988      1988
*/

#include <stdio.h>

main ()
{
   int i, j, k, map_index[1989];
   int column, row, band_width, element_width;
   float value;

   /*  Initial phase:  */
   for (k = 0; k <= 7; k ++) map_index[k]      = k;
   for (k = 0; k <= 2; k ++) map_index[k+936]  = k+8;
                             map_index[1287]   = 11;
   for (k = 0; k <= 4; k ++) map_index[k+1519] = k+12;

   /*  Body phase:  */
   for (j = 1; j <= 115; j ++) {
      for (k = 0; k <= 7; k ++) map_index[k+8*j]      = k+17*j;
```

```
      for (k = 0; k <= 2; k ++) map_index[k+3*j+936]  = k+17*j+8;
      for (k = 0; k <= 1; k ++) map_index[k+2*j+1286] = k+17*j+11;
      for (k = 0; k <= 3; k ++) map_index[k+4*j+1520] = k+17*j+13;
   }

   /*  Tail phase: */
   for (k = 0; k <= 7; k ++) map_index[k+928]  = k+1972;
   for (k = 0; k <= 2; k ++) map_index[k+1284] = k+1980;
                             map_index[1518]   = 1983;
   for (k = 0; k <= 4; k ++) map_index[k+1984] = k+1984;
/*
   for (k = 0; k < 1989; k ++)
      printf ("old, new: %5d %5d\n", k, map_index[k]);
*/

   band_width = 0;
   while ((scanf ("%d %d %f", &row, &column, &value)) != EOF) {
      element_width = map_index[row] - map_index[column];
      if (band_width < element_width)
         band_width = element_width;
      printf ("%5d %5d %lg\n", map_index[row], map_index[column], value);
  }

   printf ("Band width = %5d\n", band_width);

}  /*  end main  */
```

References

[1] A. Griewank, *On solving nonlinear equations with simple singularities or nearly singular solutions*, SIAM Review, 27/4 (1985), 537–563.

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

[3] A. Griewank, D. Juedes, J. Srinivasan, and C. 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.

Some information for Judy's title page:

Located in `/home/corliss/Research/Robey/Papers/case3.tex`

DRAFT – April 16, 1992

Automatic Differentiation Applied to Unsaturated Flow — ADOL–C Case Study[1]

by

George Corliss[2], Andreas Griewank[2], Tom Robey[3], and Steve Wright[2]

Technical Memorandum MCS–TM–162

[1]This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 and through NSF Cooperative Agreement No. CCR-8809615.

[2]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439.

[3]SPECTRA Research Institute, 1613 University Blvd. NE, Albuquerque, NM 87102–1710