ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-233

Using ADIFOR and ADIC to Provide Jacobians for the SNES Component of PETSc

by

Po-Ting Wu, * *Christian H. Bischof, and Paul D. Hovland*

e-mail: {psu,bischof,hovland}@mcs.anl.gov

Mathematics and Computer Science Division

Technical Memorandum No. 233

November 1997

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

*Current address: M/S MPK16-102, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, (415) 786-9507, e-mail: po-ting.wu@Eng.Sun.COM.

Contents

Using ADIFOR and ADIC to Provide Jacobians
for the SNES Component of PETSc

by

Po-Ting Wu, Christian H. Bischof, and Paul D. Hovland

Abstract

The solution of large-scale nonlinear problems is important to many areas of computational science. The SNES component of PETSc provides a robust and flexible suite of numerical routines for the solving such problems. These routines generally utilize the Jacobian matrix. We present a strategy for using ADIFOR or ADIC to assist in the development of a subroutine for computing this matrix. We illustrate this strategy using one of the PETSc example programs and four different approaches to computing the Jacobian via automatic differentiation.

1 Introduction

The solution of large-scale nonlinear problems pervades many facets of computational science and demands robust and flexible solution strategies. The SNES component of PETSc [2] provides a powerful suite of data-structure-neutral numerical routines for such problems. These routines generally require some derivative matrix, such as the Jacobian or, when the nonlinear system arises from optimization, a Hessian. These derivatives can be approximated with finite differences. However, the program will typically be more efficient and accurate if the programmer supplies a subroutine to compute the derivatives. Unfortunately, such a subroutine is often very complicated and difficult to program correctly by hand. We describe how automatic differentiation (AD) tools such as ADIFOR [3] and ADIC [6] can be utilized within the PETSc framework to automatically generate the subroutine for computing the derivatives. To describe this procedure, we use a simple problem from the PETSc SNES examples.

2 Outline of Procedure

We refer to the user-defined routines utilized by PETSc nonlinear solvers as `FormFunction` and `FormJacobian`, which compute the nonlinear function $F(X)$ and the Jacobian matrix $dF(X, dX)$, respectively. The routine `FormFunction` often includes several calls to PETSc routines before and after the actual function computation. These calls take care of data structure setup and communication, enabling a completely local function computation. It is this local function that we will differentiate using AD, and for that reason we will refer to it as `ADFunction`. After differentiating this routine using AD to form `g_ADFunction`, we build the `FormJacobian` routine. This procedure is illustrated in Figure 1 and outlined below.

1. Within the `FormFunction` routine, separate the PETSc pre- and postdeclaration from the local function evaluation. Extract the function computation into routine `ADFunction`.

2. Construct a composition file that lists the names of all of the source files that constitute the function evaluation program.

3. Construct a script file that tells ADIFOR/ADIC to differentiate the procedure named `ADFunction` with the independent variable X and the dependent variable F, namely, to generate code to compute the derivative $\frac{dF}{dX}$, where F is computed from X by procedure `ADFunction`.

4. Execute the AD tools command, for example,

```
Adifor2.0 AD_SCRIPT=ad_petsc.adf
```

or

```
adiC -xv -d gradient -i ad_petsc.init -u
```

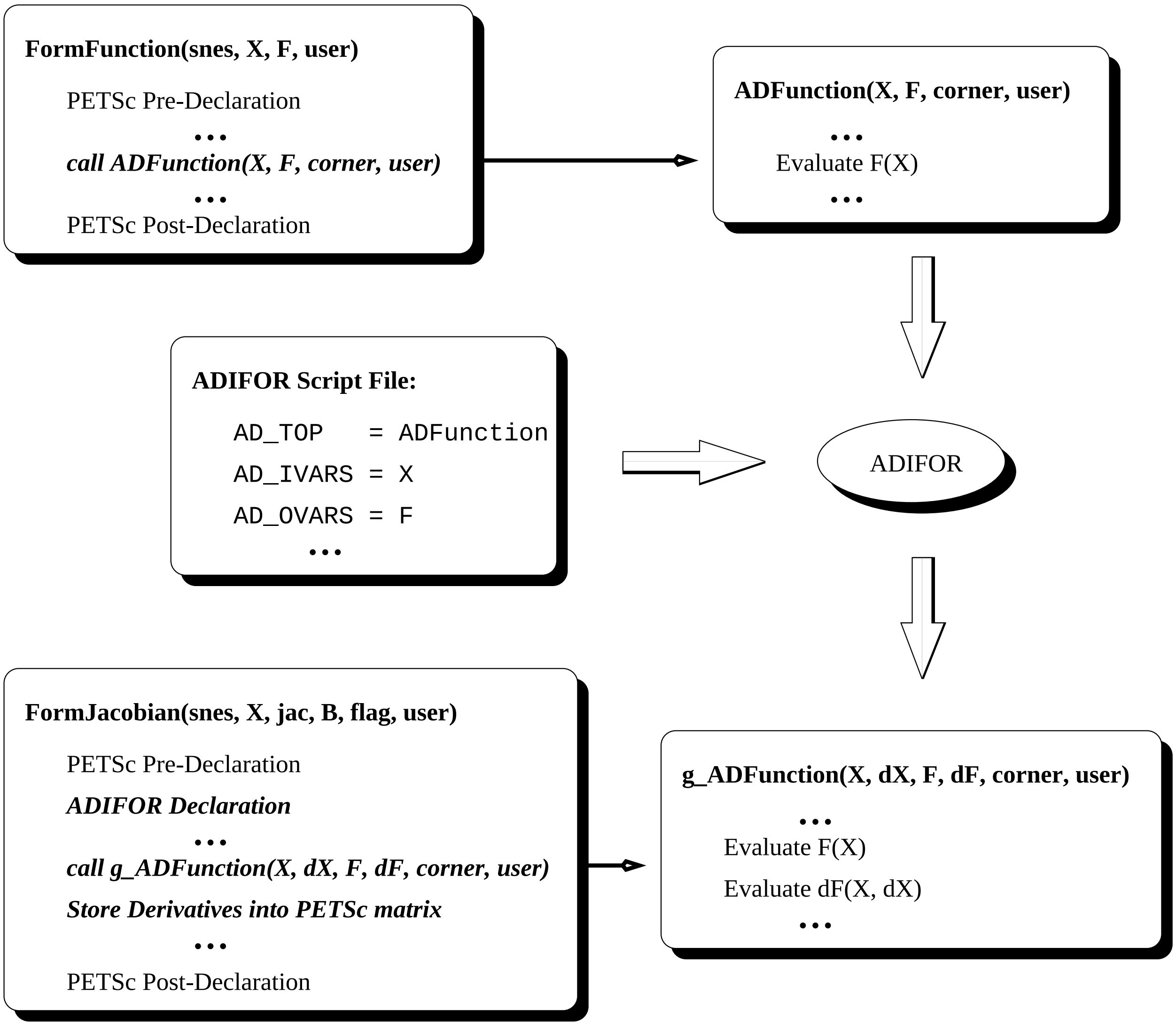

Figure 1: Integration of PETSc and ADIFOR

where ad_petsc.adf (for ADIFOR) and ad_petsc.init (for ADIC) are the names of the script files defined in the previous step. This process creates a file that contains the derivative routine `g_ADFunction` (for ADIFOR) or `ad_ADFunction` (for ADIC).

5. Create the `FormJacobian` routine:
   (a) Incorporate the PETSc predeclaration.
   (b) Set up the derivative computation.
   (c) Invoke `g_ADFunction` (for ADIFOR) or `ad_ADFunction` (for ADIC) with a user-specified value of X to compute the value of F and $\frac{dF}{dX}$.
   (d) Copy derivatives into PETSc data structures.
   (e) Incorporate the PETSc postdeclaration.

3 Isolating ADFunction

Consider the version of `FormFunction` in Figure 2. This is the Fortran version of SNES example 5 that is part of the standard PETSc distribution ($PETSC_DIR/src/snes/examples/tutorials/ex5f.F). We have inserted comments to denote the beginning and end of the local function computation. This code can be extracted into the routine `ADFunction`, as shown in Figure 3. The calls to PETSc routines surrounding this code make up the PETSC pre- and postdeclarations, as shown in Figures 4 and 5. A similar procedure can be employed for the C version. We note that newer versions of PETSc include a version of this example where most of the local function computation is encapsulated in the routine `ApplicationFunction`. However, the older version presented here seemed more appropriate for didactic purposes.

4 Running ADIFOR/ADIC

After the local function has been extracted into `ADFunction`, we can differentiate this function using ADIFOR or ADIC. To use ADIFOR, we first create a composition file specifying the file(s) containing `ADFunction` as well as a driver that calls `ADFunction`. Figure 6 gives an example of a composition file. Next we construct the ADIFOR script file shown in Figure 7. We use the `AD_TOP` keyword to specify that the function to be differentiated is defined by `ADFunction`. The independent variable (`AD_IVARS`) is `lx_a` and the dependent variable (`AD_DVARS`) is `lf_a`. We do not expect the length of `lx_a` to exceed 100, and so we use this value for `AD_PMAX`. For more information on ADIFOR script options, see the ADIFOR users manual [4]. If the filename for the ADIFOR script is `ex5f.adf`, we can invoke ADIFOR using the following command.

```
Adifor2.0 AD_SCRIPT=ex5f.adf
```

ADIFOR normally generates code that uses arrays to store dense derivative objects, but it also capable of generating code that interfaces with the SparsLinC library [4]. This library provides support for creating, updating, and extracting dynamically allocated sparse derivative objects. Figure 8 shows an ADIFOR script for generating code using sparse derivative objects. To make it easier to distinguish between the two types of code generated by ADIFOR, we use the `AD_SEP` option to specify a prefix of `g$` for variables and functions in the dense version and a prefix of `g_` for those in the SparsLinC version. The `g$ADFunction` and `g_ADFunction` subroutines produced by ADIFOR are provided in Appendixes A and B, respectively.

ADIC can be invoked by the following command, where `ex5.init` is the initialization file in Figure 9. See [5] for more information on ADIC options. The resultant code is provided in Appendix C.

```
adiC -xv -d gradient -i ex5.init -u
```

```
      subroutine FormFunction(snes,X,F,dummy)
C Declarations [omitted]

C  Scatter ghost points to local vector
      call DAGlobalToLocalBegin(da,X,INSERT_VALUES,localX,ierr)
      call DAGlobalToLocalEnd(da,X,INSERT_VALUES,localX,ierr)

C  Get pointers to vector data.
      call VecGetArray(localX,lx_v,lx_i,ierr)
      call VecGetArray(localF,lf_v,lf_i,ierr)

C  Get local grid boundaries
      call DAGetCorners(da,xs,ys,PETSC_NULL,xm,ym,PETSC_NULL,ierr)
      call DAGetGhostCorners(da,gxs,gys,PETSC_NULL,gxm,gym,
     &                                        PETSC_NULL,ierr)

C  BEGIN local computation
      ye = ys+ym-1
      xe = xs+xm-1

C  Compute function over the locally owned part of the grid
      do 20 j=ys,ye
         row = (j - gys)*gxm + xs - gxs - 1
         do 10 i=xs,xe
            row = row + 1
            if (i .eq. 0 .or. j .eq. 0
     &             .or. i .eq. mx-1 .or. j .eq. my-1) then
               lf_a(row) = lx_a(row)
            else
               u = lx_a(row)
               uxx = hydhx * (two*u
     &               - lx_a(row-1)
     &               - lx_a(row+1))
               uyy = hxdhy * (two*u
     &               - lx_a(row-gxm)
     &               - lx_a(row+gxm))
               lf_a(row) = uxx + uyy - sc*exp(u)
            endif
 10      continue
 20   continue

C END local computation

C  Restore vectors
      call VecRestoreArray(localX,lx_v,lx_i,ierr)
      call VecRestoreArray(localF,lf_v,lf_i,ierr)

C  Insert values into global vector
      call DALocalToGlobal(da,localF,INSERT_VALUES,F,ierr)
      call PLogFlops(11*ym*xm,ierr)

      return
```

Figure 2: FormFunction

```
C ---------------------------------------------------------------------
C
C  ADFunction - Evaluates nonlinear function, F(x).
C
C  Input Parameters:
C  lx_a - input vector
C
C  Output Parameter:
C  lf_a - function vector
C
      subroutine ADFunction(lx_a, lf_a, xs, ys, xm, ym,
     +                       gxs, gys, gxm, gym, user)

C  Declarations
C
C  [Omitted]

C  Get local grid boundaries

      ye = ys+ym-1
      xe = xs+xm-1

C  Compute function over the locally owned part of the grid

      do 20 j=ys,ye
         row = (j - gys)*gxm + xs - gxs - 1
         do 10 i=xs,xe
            row = row + 1
            if (i .eq. 0 .or. j .eq. 0
     &             .or. i .eq. mx-1 .or. j .eq. my-1) then
               lf_a(row) = lx_a(row)
            else
               u = lx_a(row)
               uxx = hydhx * (two*u
     &              - lx_a(row-1)
     &              - lx_a(row+1))
               uyy = hxdhy * (two*u
     &              - lx_a(row-gxm)
     &              - lx_a(row+gxm))
               lf_a(row) = uxx + uyy - sc*exp(u)
            endif
 10      continue
 20   continue

      return
      end
```

Figure 3: ADFunction

```
C  Scatter ghost points to local vector
      call DAGlobalToLocalBegin(da,X,INSERT_VALUES,localX,ierr)
      call DAGlobalToLocalEnd(da,X,INSERT_VALUES,localX,ierr)

C  Get pointers to vector data.
      call VecGetArray(localX,lx_v,lx_i,ierr)
      call VecGetArray(localF,lf_v,lf_i,ierr)

C  Get local grid boundaries
      call DAGetCorners(da,xs,ys,PETSC_NULL,xm,ym,PETSC_NULL,ierr)
      call DAGetGhostCorners(da,gxs,gys,PETSC_NULL,gxm,gym,
     &                                         PETSC_NULL,ierr)
```

Figure 4: PETSc Predeclaration

```
C  Restore vectors
      call VecRestoreArray(localX,lx_v,lx_i,ierr)
      call VecRestoreArray(localF,lf_v,lf_i,ierr)

C  Insert values into global vector
      call DALocalToGlobal(da,localF,INSERT_VALUES,F,ierr)
      call PLogFlops(11*ym*xm,ierr)
```

Figure 5: PETSc Postdeclaration

```
ex5f.f
```

Figure 6: ADIFOR composition file

```
AD_TOP                 = ADFunction
AD_PMAX                = 100
AD_IVARS               = lx_a
AD_DVARS               = lf_a
AD_PROG                = ex5f.cmp
AD_SEP                 = $
AD_OUTPUT_DIR          = .
```

Figure 7: ADIFOR script

```
AD_TOP                    = ADFunction
AD_PMAX                   = 100
AD_IVARS                  = lx_a
AD_DVARS                  = lf_a
AD_PROG                   = sprse5.cmp
AD_FLAVOR                 = sparse
AD_SEP                    = _
AD_OUTPUT_DIR             = .
```

Figure 8: ADIFOR script specifying use of SPARSLinC sparse linear algebra routines

```
[SOURCE_FILES]
        func5.c
```

Figure 9: ADIC initialization file

5 Constructing FormJacobian

We constructed four versions of `FormJacobian`. The first version uses the ADIFOR-generated routine `g$ADFunction` to compute the Jacobian. This version is most appropriate when the Jacobian is dense. The second version uses the ADIFOR-generated routine `g_ADFunction` to compute a sparse representation of the Jacobian. This version is most appropriate when the Jacobian is sparse but the sparsity pattern is unknown. The third version employs `g$ADFunction` to compute a compressed representation of the Jacobian using the strategy discussed in [1]. This method uses a coloring of the Jacobian to store structurally othogonal columns of the Jacobian in a single column of the compressed representation. The structure of the Jacobian can be supplied by the user or determined automatically. The final version uses the ADIC-generated routine `ad_ADFunction` to compute the Jacobian.

As enumerated earlier, the `FormJacobian` routine consists of five parts: PETSc predeclarations, AD setup, calling of the AD-generated function, reformatting of the Jacobian, and PETSc postdeclarations. The first and fifth parts are based on the `FormFunction` routine and are the same for all versions. In the following sections we describe the other three phases for each version. See Appendixes D–G to see how the five phases are assembled into a complete subroutine.

5.1 Jacobian

In this version, we evaluate the Jacobian matrix over the locally owned part of the grid by calling the dense version of ADIFOR-generated code. Here we present the three phases of AD setup, calling the AD-generated routine, and transferring data to PETSc data structures. The complete `FormJacobian` routine can be found in Appendix D.

5.1.1 AD Setup

The following variables are declared for use in AD.

```
double precision  g$lx_a(g$pmax$, max_x), g$lf_a(g$pmax$, max_f)
double precision  der(0:max_x)
integer           ind(0:max_x)
integer           nsize_x, nsize_f
```

The parameter g$pmax$ is the maximum number of columns in the Jacobian. Typically this will be the same as max_x, the length of lx_a, and should be consistent with the value used for AD_PMAX in the ADIFOR script. The variables g$lx_a and g$lf_a are used to store the seed matrix (see [4] for more information on seed matrices) and Jacobian, respectively. The variables der and ind are used in transferring the Jacobian to a PETSc matrix. The variables nsize_x and nsize_f are used to store the size of the local portions of X and F, respectively.

To compute the Jacobian, we initialize the seed matrix g$lx_a to the identity matrix. In order to do so, we need to know the size of the local portion of vector X, which can be determined by using the following PETSc call.

```
      call VecGetLocalSize(localX,nsize_x,ierr)
```

Once the size of localX is known, we perform the following initialization.

```
      do i = 1, nsize_x
          do j = 1, nsize_x
              g$lx_a(j, i) = 0.0
          enddo
          g$lx_a(i, i) = 1.0
      enddo
```

5.1.2 Call g$ADFunction

We invoke the AD-generated routine as follows.

```
      call g$ADFunction(nsize_x, X, g$lx_a, g$pmax$, F, g$lf_a, g$pmax$,
     +                  xs, ys, xm, ym, gxs, gys, gxm, gym, user)
```

5.1.3 Transfer AD-generated Jacobian into PETSc Data Structures

To assemble global stiffness matrices, one needs to be able to determine the global node number of each local node including the ghost nodes. These values can be determined by using the PETSc routine DAGetGlobalIndices.

```
      call DAGetGlobalIndices(da,n,ltog_v,ierr)
```

The output argument n contains the number of local nodes, including ghost nodes, while ltog_v contains a list of the global indices that correspond to the local nodes.

We determine the size of the local portion of F using the same PETSc routine as we did for X.

```
      call VecGetLocalSize(localF,nsize_f,ierr)
```

We store derivatives in the PETSc Jacobian matrix row by row as follows. The expressions ltog_v(j - 1) and ltog_v(i - 1) arise because indices in PETSc start at 0, as in C, while the variables we have used in AD follow the Fortran convention of starting at 1.

```
      do i = 1, nsize_f
          outlen = 0
          do j = 1, nsize_x
              if (g$lf_a(j, i) .ne. 0.0) then
                  der(outlen) = g$lf_a(j, i)
                  ind(outlen) = ltog_v(j - 1)
                  outlen = outlen + 1
              endif
          enddo
          if (outlen .ne. 0) then
              call MatSetValues(jac,
     +                  1, ltog_v(i - 1),
```

```
+                   outlen, ind,
+                   der, INSERT_VALUES, ierr)
    endif
enddo
```

5.2 Sparse Jacobian

In the second version, we evaluate the Jacobian matrix over the locally owned part of the grid by calling the SparsLinC version of ADIFOR-generated code. Here we present the three phases of AD setup, calling the AD-generated routine, and transferring data to PETSc data structures. The complete `FormJacobian` routine can be found in Appendix E.

5.2.1 AD Setup

In order to use SparsLinC, we must initialize it using the following call.

```
call xspini
```

After we have finished using SparsLinC, we should clean up with the following call.

```
call xspfra
```

The calls can be performed within the `FormJacobian` routine, or at a higher level, such as the main program. If `FormJacobian` is to be used more than once, we can reduce the overhead of calling these routines by calling them from the main program.

We declare the ADIFOR derivative variables as follows.

```
integer           g_lx_a(max_x), g_lf_a(max_f)
double precision  der(max_x)
integer           ind(max_x)
integer           nsize_x, nsize_f
```

These variables play the same roles as they did in the dense version. Note, however, that `g$lx_a` and `g$lf_a` do not have a leading dimension of `g$pmax$` as in the dense case. This is because these variables are used as pointers to dynamically allocated SparsLinC data structures.

Again we initialize the seed matrix to the identity matrix. However, since sparse derivative vectors are being used, each seed vector `g_lx_a(i)` contains only a single nonzero entry, with index `i` and value `one`.

```
one = 1.0
do i = 1, nsize_x
    call dspsd(g_lx_a(i), i, one, 1)
enddo
```

5.2.2 Call `g_ADFunction`

We invoke the AD-generated routine as follows.

```
call g_ADFunction(X, g_lx_a, F, g_lf_a,
+                   xs, ys, xm, ym, gxs, gys, gxm, gym, user)
```

5.2.3 Transfer AD-generated Jacobian into PETSc Data Structures

We store derivatives in the PETSc Jacobian matrix row by row as follows. The `dspxsq` routine extracts derivative values from the SparsLinC data structure `g_lf_a(i)` and stores the indices and values in `ind` and `der`, respectively.

```
      do i = 1, nsize_f
          call dspxsq(ind, der, max_x, g_lf_a(i), outlen, info)
          if (outlen .ne. 0) then
              do j = 1, outlen
                  ind(j) = ltog_v(ind(j) - 1)
              enddo
              call MatSetValues(jac,
     +                   1, ltog_v(i - 1),
     +                   outlen, ind,
     +                   der, INSERT_VALUES, ierr)
          endif
      enddo
```

5.3 Compressed Jacobian

In the third version, we compute a compressed Jacobian matrix over the locally owned part of the grid by calling the ADIFOR-generated code. Our basic strategy is to compute a sparse Jacobian using SparsLinC the first time the Jacobian is computed, then use the sparsity pattern of this Jacobian to identify structurally orthogonal columns of the Jacobian. Thereafter, we compute a compressed Jacobian and test to verify that the sparsity pattern has not changed. If the pattern has changed, we recompute the Jacobian using SparsLinC. Because this scheme is complex, we encapsulate it in `d_ADFunction`, a routine called by `FormJacobian`. The complete `FormJacobian` and `d_ADFunction` routines can be found in Appendix F.

5.3.1 AD Setup

We declare the following derivatives for use in AD.

```
      double precision  g$lx_a(g$pmax$, max_x), g$lf_a(g$pmax$, max_f)
      integer           g_lx_a(max_x), g_lf_a(max_f)
      integer           ind(max_x)
      integer           nsize_x, nsize_f
      logical           vcheck
      integer           npairs, indrow(max_nnz), indcol(max_nnz)
```

The variable `g$lx_a` is used to store a seed matrix based on the sparsity pattern of the Jacobian, while `g$lf_a` is used to store the compressed Jacobian. The parameter `g$pmax$` should be at least the number of columns in the compressed Jacobian. A conservative value is the number of columns in the Jacobian. The variables `g_lx_a` and `g_lf_a` are pointers to sparse representations of the seed matrix and Jacobian, respectively. The flag `vcheck` indicates whether we should check whether the sparsity structure of the Jacobian has changed. The variables `npairs`, `indrow`, and `indcol` are used to store the sparsity structure. The variable `ind` is used in transferring the Jacobian to the PETSc matrix.

Because we use SparsLinC for the first Jacobian evaluation, we must perform the same initialization and cleanup as in the sparse version. In the event that the sparsity pattern is already known, the user can supply it using the variables `npairs`, `indrow`, and `indcol` and the first evaluation can be performed by using a compressed Jacobian rather than SparsLinC.

In computing the compressed Jacobian, we call the `dsm` coloring routine to identify sets of structurally orthogonal columns. The "colors" of the columns are stored in `ngrp`, and the number of colors is stored in `maxgrp`. We compress the Jacobian by storing all columns of the same color in the same column of `g$lf_a`. We accomplish this by initializing the seed matrix `g$lx_a` as follows.

```
          do i = 1, nnx
              do j = 1, g$pmax$
                  g$lx_a(j, i) = 0.0
              enddo
              g$lx_a(ngrp(i), i) = 1.0
          enddo
```

If we wish to verify that the sparsity structure has not changed, we add a random column to the seed matrix. The compressed Jacobian will contain a Jacobian-vector product in this column, which can be used to check whether the sparsity pattern has changed.

```
      if (vcheck) then
          ncols = maxgrp + 1
          do i = 1, nnx
              call get_random(g$lx_a(maxgrp1, i))
          enddo
      else
          ncols = maxgrp
      endif
```

5.3.2 Call `g$ADFunction`

We invoke the AD-generated routine as follows.

```
      call g$ADFunction(ncols, lx_a, g$lx_a, g$pmax$, lf_a, g$lf_a,
     *                  g$pmax$, xs, ys, xm, ym, gxs, gys, gxm, gym,
     *                  user)
```

5.3.3 Determine Whether Sparsity Pattern Has Changed

If `vcheck` is true, we compare the Jacobian-vector product computed in the last column of `g$lf_a` with the product of the Jacobian stored in the rest of `g$lf_a` (in compressed form) with the random vector in the last column `g$lx_a`. If they are the same, we believe that the current sparsity pattern is still valid, and we accept the derivatives. If not, we reevaluate the Jacobian using SparsLinC as described above.

After recomputing the Jacobian, we compare the sparsity patterns. If the difference between these sparsity patterns is "small", then we merge (by union) the two index sets. Otherwise, we discard the old one and replace it with the new sparsity pattern. See Appendix F for the actual implementation of this strategy.

5.3.4 Transfer AD-generated Jacobian into PETSc Data Structures

Because columns of the same color are structurally orthogonal, we can recover each nonzero element of the Jacobian by using the row index and color of the column index for that element. Thus, using a row-pointer array `ptr_F`, a column-index array `id_F`, and an array the coloring information, we can construct a column-value array `d_F`, such that the nonzero values for the $i-$ th row of the Jacobian are stored in `d_F(ptr_F(i))`–`d_F(ptr_F(i+1)-1)`. Derivatives are then transferred into a PETSc Jacobian matrix row by row as follows.

```
     do i = 1, nsize_f
         outlen = ptr_F(i+1) - ptr_F(i)
         if (outlen .ne. 0) then
             do j = 1, outlen
                 ind(j) = ltog_v(id_F(ptr_F(i) + j - 1) - 1)
             enddo
             call MatSetValues(jac,
    +                      1, ltog_v(i - 1),
    +                      outlen, ind,
    +                      d_F(ptr_F(i)), INSERT_VALUES, ierr)
         endif
     enddo
```

5.4 ADIC Jacobian

In the fourth version, we evaluate the Jacobian matrix over the locally owned part of the grid by calling the ADIC generated code. Again we present the three phases of AD setup, calling the AD-generated routine, and transferring data to PETSc data structures. The complete `FormJacobian` routine can be found in Appendix G.

5.4.1 AD Setup

We include two header files that define data structures and macros for AD.

```
#include "ad_deriv.h"
#include "adintrinsics.h"
```

We also declare the following variables for use in AD.

```
DERIV_TYPE    ad_x[max_x], ad_f[max_f] ;
double        der[max_x] ;
int           ind[max_x] ;
int           nsize_x, nsize_f ;
ad_AppCtx     ad_user ;
```

The variable `ad_x` is an array of data structures of type `DERIV_TYPE` that contain fields for both the value of `x` and the derivative vector associated with `x`. Thus, this variable plays the role of the Fortran variables `lx_a` and `g$lx_a`, combined. Similarly, the variable `ad_f` plays the same role as `lf_a` and `g$lf_a`. The variables `der`, `ind`, `nsize_x`, and `nsize_f` play exactly the same role as the do in the Fortran programs.

Seed matrix initalization is performed by using the following calls.

```
ad_INIT_VAR_ARRAY(ad_x, x, nsize_x) ;
ad_INIT_INDEPENDENTS() ;
ad_SET_INDEPENDENT_ARRAY(ad_x, nsize_x) ;
```

5.4.2 Call ad_ADFunction

We invoke the ADIC-generated routine as follows.

```
ad_ADFunction(ad_x, ad_f, xs, ys, xm, ym, gxs, gys, gxm, gym, &ad_user) ;
```

5.4.3 Transfer AD-generated Jacobian into PETSc Data Structures

To assemble global stiffness matrices, one needs to be able to determine the global node number of each local node, including the ghost nodes. These values can be determined by using the PETSc routine `DAGetGlobalIndices`.

```
ierr = DAGetGlobalIndices(user->da,&nloc,&ltog)
```

The output argument `nloc` contains the number of local nodes, including ghost nodes, while `ltog` contains a list of the global indices that correspond to the local nodes.

Derivatives are extracted by using the macro `ad_GET_DERIV` and stored in the PETSc Jacobian matrix row by row as follows. Note the similarity between this code and the code presented for dense ADIFOR in Section 5.1.3. The principal differences arise because of the differences in matrix indexing in Fortran and C.

```
for (i = 0 ; i < nsize_f ; i++) {
    outlen = 0 ;
    for (j = 0 ; j < nsize_x ; j++) {
        ad_GET_DERIV(grad, ad_f[i], j) ;
        if (grad != 0.0) {
            der[outlen] = grad ;
```

```
            ind[outlen] = ltog[j] ;
            ++outlen ;
        }
    }
    if (outlen != 0) {
        ierr = MatSetValues(jac,1,&ltog[i],outlen,ind,der,INSERT_VALUES) ;
        CHKERRQ(ierr) ;
    }
}
```

6 Summary

We presented a general strategy for using ADIFOR or ADIC to assist in developing a subroutine for computing the Jacobian of a function. This subroutine can be used by the SNES component of PETSc. The basic approach is to isolate a local function evaluation from PETSc data distribution instructions, apply ADIFOR or ADIC to this local function, then encapsulate the code generated using AD in a subroutine that performs the appropriate distribution of data. We demonstrated how this strategy could be applied to one of the PETSc example programs using each of four different approaches to computing the Jacobian via AD.

Appendix

A g$ADFunction

```
C                          DISCLAIMER
C
C   This file was generated on 11/14/97 by the version of
C   ADIFOR compiled on Oct 30 1997.
C
C   ADIFOR was prepared as an account of work sponsored by an
C   agency of the United States Government, Rice University, and
C   the University of Chicago.  NEITHER THE AUTHOR(S), THE UNITED
C   STATES GOVERNMENT NOR ANY AGENCY THEREOF, NOR RICE UNIVERSITY,
C   NOR THE UNIVERSITY OF CHICAGO, INCLUDING ANY OF THEIR EMPLOYEES
C   OR OFFICERS, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES
C   ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETE-
C   NESS, OR USEFULNESS OF ANY INFORMATION OR PROCESS DISCLOSED, OR
C   REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.
C
C --------------------------------------------------------------------
C
C  ADFunction - Evaluates nonlinear function, F(x).
C
C  Input Parameters:
C  lx_a - input vector
C  user - optional user-defined context, as set by SNESSetFunction()
C
C  Output Parameter:
C  lf_a - function vector
C
      subroutine g$adfunction(g$p$, lx_a, g$lx_a, ldg$lx_a, lf_a, g$lf_a
     *, ldg$lf_a, xs, ys, xm, ym, gxs, gys, gxm, gym, user)
        implicit none
C
        double precision two, one, lambda, hx, hy, hxdhy, hydhx, sc
        double precision u, uxx, uyy
        integer i, j, row, mx, my, xs, ys, xm, ym, xe, ye
        integer user(*), gxs, gys, gxm, gym
C
C  Declarations for use with local arrays
C
        double precision lx_a(0:*), lf_a(0:*)
C
        common /params/ lambda
C
        integer g$pmax$
        parameter (g$pmax$ = 100)
        integer g$i$, g$p$, ldg$lf_a, ldg$lx_a
        double precision d1$p, d7$b, d5$v, d5$b, g$lf_a(ldg$lf_a, 0:*),
     *g$lx_a(ldg$lx_a, 0:*), g$u(g$pmax$), g$uxx(g$pmax$), g$uyy(g$pmax$
     *)
        integer g_ehfid
        save g$u, g$uxx, g$uyy
        data g_ehfid /0/
C
        call ehsfid(g_ehfid, 'adfunction','g_func14.f')
C
```

```
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p$ is greater than g$pmax$'
          stop
        endif
        one = 1.0
        two = 2.0
        mx = user(1)
        my = user(2)
        hx = one / dble(mx - 1)
        hy = one / dble(my - 1)
        sc = hx * hy * lambda
        hxdhy = hx / hy
        hydhx = hy / hx
C
C  Get local grid boundaries
C
        ye = ys + ym - 1
        xe = xs + xm - 1
C
C  Compute function over the locally owned part of the grid
C
        do 99998 j = ys, ye
          row = (j - gys) * gxm + xs - gxs - 1
          do 99999 i = xs, xe
            row = row + 1
            if (i .eq. 0 .or. j .eq. 0 .or. i .eq. mx - 1 .or. j .eq. my
     * - 1) then
              do g$i$ = 1, g$p$
                g$lf_a(g$i$, row) = g$lx_a(g$i$, row)
              enddo
              lf_a(row) = lx_a(row)
C--------
            else
              do g$i$ = 1, g$p$
                g$u(g$i$) = g$lx_a(g$i$, row)
              enddo
              u = lx_a(row)
C--------
              d7$b = hydhx * two
              do g$i$ = 1, g$p$
                g$uxx(g$i$) = (-hydhx) * g$lx_a(g$i$, row + 1) + (-hydhx
     *) * g$lx_a(g$i$, row - 1) + d7$b * g$u(g$i$)
              enddo
              uxx = hydhx * (two * u - lx_a(row - 1) - lx_a(row + 1))
C--------
              d7$b = hxdhy * two
              do g$i$ = 1, g$p$
                g$uyy(g$i$) = (-hxdhy) * g$lx_a(g$i$, row + gxm) + (-hxd
     *hy) * g$lx_a(g$i$, row - gxm) + d7$b * g$u(g$i$)
              enddo
              uyy = hxdhy * (two * u - lx_a(row - gxm) - lx_a(row + gxm)
     *)
C--------
              d5$v = exp(u)
              d1$p =  d5$v
              d5$b = (-sc) * d1$p
              do g$i$ = 1, g$p$
                g$lf_a(g$i$, row) = d5$b * g$u(g$i$) + g$uyy(g$i$) + g$u
```

```
     *xx(g$i$)
              enddo
              lf_a(row) = uxx + uyy - sc * d5$v
C--------
            endif
10          continue
99999     continue
20        continue
99998   continue
C
        return
      end
```

B g_ADFunction

```
C                           DISCLAIMER
C
C   This file was generated on 11/14/97 by the version of
C   ADIFOR compiled on Oct 30 1997.
C
C   ADIFOR was prepared as an account of work sponsored by an
C   agency of the United States Government, Rice University, and
C   the University of Chicago.  NEITHER THE AUTHOR(S), THE UNITED
C   STATES GOVERNMENT NOR ANY AGENCY THEREOF, NOR RICE UNIVERSITY,
C   NOR THE UNIVERSITY OF CHICAGO, INCLUDING ANY OF THEIR EMPLOYEES
C   OR OFFICERS, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES
C   ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETE-
C   NESS, OR USEFULNESS OF ANY INFORMATION OR PROCESS DISCLOSED, OR
C   REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.
C
C --------------------------------------------------------------------
C
C  ADFunction - Evaluates nonlinear function, F(x).
C
C  Input Parameters:
C  lx_a - input vector
C  user - optional user-defined context, as set by SNESSetFunction()
C
C  Output Parameter:
C  lf_a - function vector
C
      subroutine g_adfunction(lx_a, g_lx_a, lf_a, g_lf_a, xs, ys, xm, ym
     *, gxs, gys, gxm, gym, user)
        implicit none
C
        double precision two, one, lambda, hx, hy, hxdhy, hydhx, sc
        double precision u, uxx, uyy
        integer i, j, row, mx, my, xs, ys, xm, ym, xe, ye
        integer user(*), gxs, gys, gxm, gym
C
C  Declarations for use with local arrays
C
        double precision lx_a(0:1), lf_a(0:1)
C
        common /params/ lambda
C
        integer g_lf_a(0:1), g_lx_a(0:1), g_u, g_uxx, g_uyy
```

```
        double precision d1_p, d5_v
        integer g_ehfid
        save g_u, g_uxx, g_uyy
        data g_ehfid /0/
C
        call ehsfid(g_ehfid, 'adfunction','g_func14_1.f')
C
        one = 1.0
        two = 2.0
        mx = user(1)
        my = user(2)
        hx = one / dble(mx - 1)
        hy = one / dble(my - 1)
        sc = hx * hy * lambda
        hxdhy = hx / hy
        hydhx = hy / hx
C
C  Get local grid boundaries
C
        ye = ys + ym - 1
        xe = xs + xm - 1
C
C  Compute function over the locally owned part of the grid
C
        do 99998 j = ys, ye
          row = (j - gys) * gxm + xs - gxs - 1
          do 99999 i = xs, xe
            row = row + 1
            if (i .eq. 0 .or. j .eq. 0 .or. i .eq. mx - 1 .or. j .eq. my
     * - 1) then
              call dspcpq(g_lf_a(row), g_lx_a(row))
              lf_a(row) = lx_a(row)
C--------
            else
              call dspcpq(g_u, g_lx_a(row))
              u = lx_a(row)
C--------
              call dspg3q(g_uxx, -hydhx, g_lx_a(row + 1), -hydhx, g_lx_a
     +(row - 1), hydhx * two, g_u)
              uxx = hydhx * (two * u - lx_a(row - 1) - lx_a(row + 1))
C--------
              call dspg3q(g_uyy, -hxdhy, g_lx_a(row + gxm), -hxdhy, g_lx
     +_a(row - gxm), hxdhy * two, g_u)
              uyy = hxdhy * (two * u - lx_a(row - gxm) - lx_a(row + gxm)
     *)
C--------
              d5_v = exp(u)
              d1_p =  d5_v
              call dspg3q(g_lf_a(row), (-sc) * d1_p, g_u, 1.0d0, g_uyy,
     +1.0d0, g_uxx)
              lf_a(row) = uxx + uyy - sc * d5_v
C--------
            endif
10          continue
99999     continue
20        continue
99998   continue
C
```

```
        return
      end
```

C ad_ADFunction

```
#include "ad_deriv.h"
#include <math.h>
#include "adintrinsics.h"
typedef struct {
  DERIV_TYPE param;
  int mx, my;
  void   *localX,  *localF;
  void   *da;
  int rank;
} AppCtx;
int ad_ADFunction(DERIV_TYPE x[], DERIV_TYPE f[], int xs, int ys, int xm,
    int ym, int gxs, int gys, int gxm, int gym, AppCtx  *user) {
  double  ad_loc_0,ad_loc_1,ad_loc_2,ad_loc_3;
  double  ad_adj_0,ad_adj_1,ad_adj_2,ad_adj_3;
  double ad_adji_0;
  DERIV_TYPE ad_var_0;
  DERIV_TYPE ad_var_1;
  DERIV_TYPE ad_var_2;
  int i, j, row, mx, my;
  DERIV_TYPE two, one, lambda, hx, hy, hxdhy, hydhx, sc;
  DERIV_TYPE u, uxx, uyy;

        static int g_filenum = 0;
        if (g_filenum == 0) {
            adintr_ehsfid(&g_filenum, __FILE__, "ADFunction");
        }
          {
    ad_grad_axpy_0(DERIV_grad(two));
    DERIV_VAL(two) = 2.0;
  }
  {
    ad_grad_axpy_0(DERIV_grad(one));
    DERIV_VAL(one) = 1.0;
  }
  mx = user->mx;
  my = user->my;
  {
    ad_grad_axpy_1(DERIV_grad(lambda), 1.000000000000000e+00,
        DERIV_grad(user->param));
    DERIV_VAL(lambda) = DERIV_VAL(user->param);
  }
  {
    ad_grad_axpy_0(DERIV_grad(ad_var_0));
    DERIV_VAL(ad_var_0) = (double )(mx - 1);
  }
  {
    ad_loc_0 = DERIV_VAL(one) / DERIV_VAL(ad_var_0);
    ad_adj_0 =  - ad_loc_0 / DERIV_VAL(ad_var_0);
    ad_adj_1 = 1.000000000000000e+00 / DERIV_VAL(ad_var_0);
    ad_grad_axpy_2(DERIV_grad(hx), ad_adj_1, DERIV_grad(one), ad_adj_0,
        DERIV_grad(ad_var_0));
    DERIV_VAL(hx) = ad_loc_0;
```

```
}
{
  ad_grad_axpy_0(DERIV_grad(ad_var_1));
  DERIV_VAL(ad_var_1) = (double )(my - 1);
}
{
  ad_loc_0 = DERIV_VAL(one) / DERIV_VAL(ad_var_1);
  ad_adj_0 =  - ad_loc_0 / DERIV_VAL(ad_var_1);
  ad_adj_1 = 1.000000000000000e+00 / DERIV_VAL(ad_var_1);
  ad_grad_axpy_2(DERIV_grad(hy), ad_adj_1, DERIV_grad(one), ad_adj_0,
      DERIV_grad(ad_var_1));
  DERIV_VAL(hy) = ad_loc_0;
}
{
  ad_loc_0 = DERIV_VAL(hx) * DERIV_VAL(hy);
  ad_loc_1 = ad_loc_0 * DERIV_VAL(lambda);
  ad_adj_0 = DERIV_VAL(hx) * DERIV_VAL(lambda);
  ad_adj_1 = DERIV_VAL(hy) * DERIV_VAL(lambda);
  ad_grad_axpy_3(DERIV_grad(sc), ad_adj_1, DERIV_grad(hx), ad_adj_0,
      DERIV_grad(hy), ad_loc_0, DERIV_grad(lambda));
  DERIV_VAL(sc) = ad_loc_1;
}
{
  ad_loc_0 = DERIV_VAL(hx) / DERIV_VAL(hy);
  ad_adj_0 =  - ad_loc_0 / DERIV_VAL(hy);
  ad_adj_1 = 1.000000000000000e+00 / DERIV_VAL(hy);
  ad_grad_axpy_2(DERIV_grad(hxdhy), ad_adj_1, DERIV_grad(hx), ad_adj_0,
      DERIV_grad(hy));
  DERIV_VAL(hxdhy) = ad_loc_0;
}
{
  ad_loc_0 = DERIV_VAL(hy) / DERIV_VAL(hx);
  ad_adj_0 =  - ad_loc_0 / DERIV_VAL(hx);
  ad_adj_1 = 1.000000000000000e+00 / DERIV_VAL(hx);
  ad_grad_axpy_2(DERIV_grad(hydhx), ad_adj_1, DERIV_grad(hy), ad_adj_0,
      DERIV_grad(hx));
  DERIV_VAL(hydhx) = ad_loc_0;
}
for (j = ys; j < ys + ym; j++) {
  row = (j - gys) * gxm + xs - gxs - 1;
  for (i = xs; i < xs + xm; i++) {
    (row)++;
    if (i == 0 || j == 0 || i == mx - 1 || j == my - 1) {
      {
        ad_grad_axpy_1(DERIV_grad(f[row]), 1.000000000000000e+00,
            DERIV_grad(x[row]));
        DERIV_VAL(f[row]) = DERIV_VAL(x[row]);
      }
      continue;
           }
    {
      ad_grad_axpy_1(DERIV_grad(u), 1.000000000000000e+00,
          DERIV_grad(x[row]));
      DERIV_VAL(u) = DERIV_VAL(x[row]);
    }
    {
      ad_loc_0 = DERIV_VAL(two) * DERIV_VAL(u);
      ad_loc_1 = ad_loc_0 - DERIV_VAL(x[row - 1]);
```

```
        ad_loc_2 = ad_loc_1 - DERIV_VAL(x[row + 1]);
        ad_loc_3 = ad_loc_2 * DERIV_VAL(hydhx);
        ad_adj_0 =  - DERIV_VAL(hydhx);
        ad_adj_1 =  - DERIV_VAL(hydhx);
        ad_adj_2 = DERIV_VAL(two) * DERIV_VAL(hydhx);
        ad_adj_3 = DERIV_VAL(u) * DERIV_VAL(hydhx);
        ad_grad_axpy_5(DERIV_grad(uxx), ad_adj_3, DERIV_grad(two), ad_adj_2,
            DERIV_grad(u), ad_adj_1, DERIV_grad(x[row - 1]), ad_adj_0,
            DERIV_grad(x[row + 1]), ad_loc_2, DERIV_grad(hydhx));
        DERIV_VAL(uxx) = ad_loc_3;
      }
      {
        ad_loc_0 = DERIV_VAL(two) * DERIV_VAL(u);
        ad_loc_1 = ad_loc_0 - DERIV_VAL(x[row - gxm]);
        ad_loc_2 = ad_loc_1 - DERIV_VAL(x[row + gxm]);
        ad_loc_3 = ad_loc_2 * DERIV_VAL(hxdhy);
        ad_adj_0 =  - DERIV_VAL(hxdhy);
        ad_adj_1 =  - DERIV_VAL(hxdhy);
        ad_adj_2 = DERIV_VAL(two) * DERIV_VAL(hxdhy);
        ad_adj_3 = DERIV_VAL(u) * DERIV_VAL(hxdhy);
        ad_grad_axpy_5(DERIV_grad(uyy), ad_adj_3, DERIV_grad(two), ad_adj_2,
            DERIV_grad(u), ad_adj_1, DERIV_grad(x[row - gxm]), ad_adj_0,
            DERIV_grad(x[row + gxm]), ad_loc_2, DERIV_grad(hxdhy));
        DERIV_VAL(uyy) = ad_loc_3;
      }
     DERIV_VAL(ad_var_2) = exp( DERIV_VAL(u));
      ad_adji_0 = DERIV_VAL(ad_var_2);
      ad_grad_axpy_1(DERIV_grad(ad_var_2), ad_adji_0, DERIV_grad(u));
      {
        ad_loc_0 = DERIV_VAL(uxx) + DERIV_VAL(uyy);
        ad_loc_1 = DERIV_VAL(sc) * DERIV_VAL(ad_var_2);
        ad_loc_2 = ad_loc_0 - ad_loc_1;
        ad_adj_0 =  - DERIV_VAL(sc);
        ad_adj_1 =  - DERIV_VAL(ad_var_2);
        ad_grad_axpy_4(DERIV_grad(f[row]), 1.000000000000000e+00,
            DERIV_grad(uxx), 1.000000000000000e+00, DERIV_grad(uyy), ad_adj_1,
            DERIV_grad(sc), ad_adj_0, DERIV_grad(ad_var_2));
        DERIV_VAL(f[row]) = ad_loc_2;
      }
           }
       }
  return 0;
}
int  ad_grad_size;
int  ad_GrAd_SiZe;
void  ad_ad_init() {
  ad_GrAd_SiZe = ad_grad_size;
}
```

D Dense FormJacobian

```
C ---------------------------------------------------------------------
C
C  FormJacobian - Evaluates Jacobian matrix.
C
C  Input Parameters:
C  snes - the SNES context
```

```
C  x - input vector
C  ptr - optional user-defined context, as set by SNESSetJacobian()
C
C  Output Parameters:
C  A - Jacobian matrix
C  B - optionally different preconditioning matrix
C  flag - flag indicating matrix structure
C
C  Notes:
C  Due to grid point reordering with DAs, we must always work
C  with the local grid points, and then transform them to the new
C  global numbering with the "ltog" mapping (via DAGetGlobalIndices()).
C  We cannot work directly with the global numbers for the original
C  uniprocessor grid!
C
      subroutine FormJacobian(snes,X,jac,B,flag,user)
      implicit none

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
#include "include/FINCLUDE/da.h"
#include "include/FINCLUDE/mat.h"
#include "include/FINCLUDE/pc.h"
#include "include/FINCLUDE/snes.h"

      SNES     snes
      DA       da
      Vec      X, localX, localF
      Mat      jac, B
      integer  flag, user(*), ierr
      integer  nloc, xs, ys, xm, ym, gxs, gys, gxm, gym

C  Declarations for use with local arrays

      Scalar   lx_v(1), lf_v(1)
      integer  ltog_a(0:1), ltog_i, lx_i, lf_i

      integer           g$pmax$, max_x, max_f
      parameter         (g$pmax$ = 100, max_x = 100, max_f = 100)
      double precision  g$lx_a(g$pmax$, max_x), g$lf_a(g$pmax$, max_f)
      double precision  der(0:max_x)
      integer           ind(0:max_x)
      integer           nsize_x, nsize_f, outlen, i, j

C  Macro to facilitate use of the local-to-global mapping
C  (obtained via DAGetGlobalIndices) in Fortran.  Note
C  that this mapping is always 0-based (following the
C  C convention) rather than 1-based.

#define ltog_v(ib)  ltog_a(ltog_i + (ib))

C  Set parameters

      da     = user(3)
      localX = user(4)
      localF = user(5)

C  Scatter ghost points to local vector, using the 2-step process
```

```
C     DAGlobalToLocalBegin(), DAGlobalToLocalEnd()
C  Computations can be done while messages are in transition,
C  by placing code between these two statements.

      call DAGlobalToLocalBegin(da,X,INSERT_VALUES,localX,ierr)
      call DAGlobalToLocalEnd(da,X,INSERT_VALUES,localX,ierr)

C  Get a pointer to vector data

      call VecGetArray(localX,lx_v,lx_i,ierr)
      call VecGetArray(localF,lf_v,lf_i,ierr)

C  Get local grid boundaries

      call DAGetCorners(da,xs,ys,PETSC_NULL,xm,ym,PETSC_NULL,ierr)
      call DAGetGhostCorners(da,gxs,gys,PETSC_NULL,gxm,gym,
     &                                        PETSC_NULL,ierr)

C  Get the global node numbers for all local nodes, including ghost points
C    - Note that the Fortran interface to DAGetGlobalIndices() differs from
C      the C version.  See the Fortran chapter of the users manual for details.

      call DAGetGlobalIndices(da,nloc,ltog_a,ltog_i,ierr)

C
      call VecGetLocalSize(localX,nsize_x,ierr)
      call VecGetLocalSize(localF,nsize_f,ierr)

C Initialize seed matrix & call ADIFOR code.

      do i = 1, nsize_x
         do j = 1, nsize_x
             g$lx_a(j, i) = 0.0
         enddo
         g$lx_a(i, i) = 1.0
      enddo
      call g$ADFunction(nsize_x,
     +                 lx_a(0), g$lx_a, g$pmax$,
     +                 lf_a(0), g$lf_a, g$pmax$,
     +                 xs, ys, xm, ym, gxs, gys, gxm, gym, user)

C Store derivatives into jac matrix.

      do i = 1, nsize_f
          outlen = 0
          do j = 1, nsize_x
              if (g$lf_a(j, i) .ne. 0.0) then
                  der(outlen) = g$lf_a(j, i)
          ind(outlen) = ltog_v(j - 1)
          outlen = outlen + 1
             endif
         enddo
         if (outlen .ne. 0) then
             call MatSetValues(jac,
     +                 1, ltog_v(i - 1),
     +                 outlen, ind,
     +                 der, INSERT_VALUES, ierr)
         endif
```

```
      enddo

C  Assemble matrix, using the 2-step process:
C     MatAssemblyBegin(), MatAssemblyEnd()
C  Computations can be done while messages are in transition,
C  by placing code between these two statements.

      call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)
      call VecRestoreArray(localX,lx_v,lx_i,ierr)
      call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)

C  Set flag to indicate that the Jacobian matrix retains an identical
C  nonzero structure throughout all nonlinear iterations (although the
C  values of the entries change). Thus, we can save some work in setting
C  up the preconditioner (e.g., no need to redo symbolic factorization for
C  ILU/ICC preconditioners).
C   - If the nonzero structure of the matrix is different during
C     successive linear solves, then the flag DIFFERENT_NONZERO_PATTERN
C     must be used instead.  If you are unsure whether the matrix
C     structure has changed or not, use the flag DIFFERENT_NONZERO_PATTERN.
C   - Caution:  If you specify SAME_NONZERO_PATTERN, PETSc
C     believes your assertion and does not check the structure
C     of the matrix.  If you erroneously claim that the structure
C     is the same when it actually is not, the new preconditioner
C     will not function correctly.  Thus, use this optimization
C     feature with caution!

      flag = SAME_NONZERO_PATTERN

      return
      end
```

E Sparse FormJacobian

```
C ---------------------------------------------------------------------
C
C  FormJacobian - Evaluates Jacobian matrix.
C
C  Input Parameters:
C  snes - the SNES context
C  x - input vector
C  ptr - optional user-defined context, as set by SNESSetJacobian()
C
C  Output Parameters:
C  A - Jacobian matrix
C  B - optionally different preconditioning matrix
C  flag - flag indicating matrix structure
C
C  Notes:
C  Due to grid point reordering with DAs, we must always work
C  with the local grid points, and then transform them to the new
C  global numbering with the "ltog" mapping (via DAGetGlobalIndices()).
C  We cannot work directly with the global numbers for the original
C  uniprocessor grid!
C
      subroutine FormJacobian(snes,X,jac,B,flag,user)
      implicit none
```

```
#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
#include "include/FINCLUDE/da.h"
#include "include/FINCLUDE/mat.h"
#include "include/FINCLUDE/pc.h"
#include "include/FINCLUDE/snes.h"

      SNES     snes
      DA       da
      Vec      X, localX, localF
      Mat      jac, B
      integer  flag, user(*), ierr
      integer  nloc, xs, ys, xm, ym, gxs, gys, gxm, gym

C  Declarations for use with local arrays

      Scalar   lx_v(1), lf_v(1)
      integer  ltog_a(0:1), ltog_i, lx_i, lf_i

      integer          max_x, max_f
      parameter        (max_x = 100, max_f = 100)
      integer          g_lx_a(max_x), g_lf_a(max_f)
      double precision der(max_x)
      integer          ind(max_x)
      double precision one
      integer          nsize_x, nsize_f, outlen, info, i, j

C  Macro to facilitate use of the local-to-global mapping
C  (obtained via DAGetGlobalIndices) in Fortran.  Note
C  that this mapping is always 0-based (following the
C  C convention) rather than 1-based.

#define ltog_v(ib)  ltog_a(ltog_i + (ib))

C  Set parameters

      da     = user(3)
      localX = user(4)
      localF = user(5)

C  Scatter ghost points to local vector, using the 2-step process
C     DAGlobalToLocalBegin(), DAGlobalToLocalEnd()
C  Computations can be done while messages are in transition,
C  by placing code between these two statements.

      call DAGlobalToLocalBegin(da,X,INSERT_VALUES,localX,ierr)
      call DAGlobalToLocalEnd(da,X,INSERT_VALUES,localX,ierr)

C  Get a pointer to vector data

      call VecGetArray(localX,lx_v,lx_i,ierr)
      call VecGetArray(localF,lf_v,lf_i,ierr)

C  Get local grid boundaries

      call DAGetCorners(da,xs,ys,PETSC_NULL,xm,ym,PETSC_NULL,ierr)
      call DAGetGhostCorners(da,gxs,gys,PETSC_NULL,gxm,gym,
```

```
     &                                         PETSC_NULL,ierr)

C  Get the global node numbers for all local nodes, including ghost points
C    - Note that the Fortran interface to DAGetGlobalIndices() differs from
C      the C version.  See the Fortran chapter of the users manual for details.

      call DAGetGlobalIndices(da,nloc,ltog_a,ltog_i,ierr)

C
      call VecGetLocalSize(localX,nsize_x,ierr)
      call VecGetLocalSize(localF,nsize_f,ierr)

C Initialize seed matrix & call ADIFOR code.

      one = 1.0
      do i = 1, nsize_x
         call dspsd(g_lx_a(i), i, one, 1)
      enddo
      call g_ADFunction(lx_a(0), g_lx_a,
     +                  lf_a(0), g_lf_a,
     +                  xs, ys, xm, ym, gxs, gys, gxm, gym, user)

C Store derivatives into jac matrix.

      do i = 1, nsize_f
         call dspxsq(ind, der, max_x, g_lf_a(i), outlen, info)
         if (outlen .ne. 0) then
            do j = 1, outlen
          ind(j) = ltog_v(ind(j) - 1)
            enddo
            call MatSetValues(jac,
     +                  1, ltog_v(i - 1),
     +                  outlen, ind,
     +                  der, INSERT_VALUES, ierr)
         endif
      enddo

C  Assemble matrix, using the 2-step process:
C     MatAssemblyBegin(), MatAssemblyEnd()
C  Computations can be done while messages are in transition,
C  by placing code between these two statements.

      call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)
      call VecRestoreArray(localX,lx_v,lx_i,ierr)
      call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)

C  Set flag to indicate that the Jacobian matrix retains an identical
C  nonzero structure throughout all nonlinear iterations (although the
C  values of the entries change). Thus, we can save some work in setting
C  up the preconditioner (e.g., no need to redo symbolic factorization for
C  ILU/ICC preconditioners).
C   - If the nonzero structure of the matrix is different during
C     successive linear solves, then the flag DIFFERENT_NONZERO_PATTERN
C     must be used instead.  If you are unsure whether the matrix
C     structure has changed or not, use the flag DIFFERENT_NONZERO_PATTERN.
C   - Caution:  If you specify SAME_NONZERO_PATTERN, PETSc
C     believes your assertion and does not check the structure
C     of the matrix.  If you erroneously claim that the structure
```

```
C     is the same when it actually is not, the new preconditioner
C     will not function correctly.  Thus, use this optimization
C     feature with caution!

      flag = SAME_NONZERO_PATTERN

      return
      end
```

F Colored FormJacobian

```
C ---------------------------------------------------------------------
C
C  FormJacobian - Evaluates Jacobian matrix.
C
C  Input Parameters:
C  snes - the SNES context
C  x - input vector
C  ptr - optional user-defined context, as set by SNESSetJacobian()
C
C  Output Parameters:
C  A - Jacobian matrix
C  B - optionally different preconditioning matrix
C  flag - flag indicating matrix structure
C
C  Notes:
C  Due to grid point reordering with DAs, we must always work
C  with the local grid points, and then transform them to the new
C  global numbering with the "ltog" mapping (via DAGetGlobalIndices()).
C  We cannot work directly with the global numbers for the original
C  uniprocessor grid!
C
      subroutine FormJacobian(snes,X,jac,B,flag,user)
      implicit none

#include "include/FINCLUDE/petsc.h"
#include "include/FINCLUDE/vec.h"
#include "include/FINCLUDE/da.h"
#include "include/FINCLUDE/mat.h"
#include "include/FINCLUDE/pc.h"
#include "include/FINCLUDE/snes.h"

C ADIFOR-PETSc include file

#include "ad_petsc.inc"

      SNES     snes
      DA       da
      Vec      X, localX, localF
      Mat      jac, B
      integer  flag, user(*), ierr
      integer  nloc, xs, ys, xm, ym, gxs, gys, gxm, gym

C  Declarations for use with local arrays

      Scalar   lx_v(1), lf_v(1)
      integer  ltog_a(0:1), ltog_i, lx_i, lf_i
```

```
      double precision  g$lx_a(g$pmax$, max_x), g$lf_a(g$pmax$, max_f)
      integer           g_lx_a(max_x), g_lf_a(max_f)
      integer           ind(max_x)
      integer           nsize_x, nsize_f, outlen, i, j
      integer           indrow(1), indcol(1)

C  Macro to facilitate use of the local-to-global mapping
C  (obtained via DAGetGlobalIndices) in Fortran.  Note
C  that this mapping is always 0-based (following the
C  C convention) rather than 1-based.

#define ltog_v(ib)  ltog_a(ltog_i + (ib))

C  Set parameters

      da     = user(3)
      localX = user(4)
      localF = user(5)

C  Scatter ghost points to local vector, using the 2-step process
C     DAGlobalToLocalBegin(), DAGlobalToLocalEnd()
C  Computations can be done while messages are in transition,
C  by placing code between these two statements.

      call DAGlobalToLocalBegin(da,X,INSERT_VALUES,localX,ierr)
      call DAGlobalToLocalEnd(da,X,INSERT_VALUES,localX,ierr)

C  Get a pointer to vector data

      call VecGetArray(localX,lx_v,lx_i,ierr)
      call VecGetArray(localF,lf_v,lf_i,ierr)

C  Get local grid boundaries

      call DAGetCorners(da,xs,ys,PETSC_NULL,xm,ym,PETSC_NULL,ierr)
      call DAGetGhostCorners(da,gxs,gys,PETSC_NULL,gxm,gym,
     &                                         PETSC_NULL,ierr)

C  Get the global node numbers for all local nodes, including ghost points
C    - Note that the Fortran interface to DAGetGlobalIndices() differs from
C      the C version.  See the Fortran chapter of the users manual for details.

      call DAGetGlobalIndices(da,nloc,ltog_a,ltog_i,ierr)

C
      call VecGetLocalSize(localX,nsize_x,ierr)
      call VecGetLocalSize(localF,nsize_f,ierr)

C Initialize seed matrix & call ADIFOR code.

      call d_ADFunction(nsize_x, lx_a(0), g_lx_a, g$lx_a,
     +                  nsize_f, lf_a(0), g_lf_a, g$lf_a,
     +                  xs, ys, xm, ym, gxs, gys, gxm, gym, user,
     +                  .FALSE., 0, indrow, indcol)

C Store derivatives into jac matrix.
```

```
      do i = 1, nsize_f
         outlen = ptr_F(i+1) - ptr_F(i)
         if (outlen .ne. 0) then
            do j = 1, outlen
          ind(j) = ltog_v(id_F(ptr_F(i) + j - 1) - 1)
            enddo
            call MatSetValues(jac,
     +                 1, ltog_v(i - 1),
     +                 outlen, ind,
     +                 d_F(ptr_F(i)), INSERT_VALUES, ierr)
         endif
      enddo

C  Assemble matrix, using the 2-step process:
C     MatAssemblyBegin(), MatAssemblyEnd()
C  Computations can be done while messages are in transition,
C  by placing code between these two statements.

      call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)
      call VecRestoreArray(localX,lx_v,lx_i,ierr)
      call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)

C  Set flag to indicate that the Jacobian matrix retains an identical
C  nonzero structure throughout all nonlinear iterations (although the
C  values of the entries change). Thus, we can save some work in setting
C  up the preconditioner (e.g., no need to redo symbolic factorization for
C  ILU/ICC preconditioners).
C   - If the nonzero structure of the matrix is different during
C     successive linear solves, then the flag DIFFERENT_NONZERO_PATTERN
C     must be used instead.  If you are unsure whether the matrix
C     structure has changed or not, use the flag DIFFERENT_NONZERO_PATTERN.
C   - Caution:  If you specify SAME_NONZERO_PATTERN, PETSc
C     believes your assertion and does not check the structure
C     of the matrix.  If you erroneously claim that the structure
C     is the same when it actually is not, the new preconditioner
C     will not function correctly.  Thus, use this optimization
C     feature with caution!

      flag = SAME_NONZERO_PATTERN

      return
      end

C---------------------------------------------------------------------------

      subroutine d_ADFunction(nnx, X, g_X, g$X, nnf, F, g_F, g$F,
     *                 xs, ys, xm, ym, gxs, gys, gxm, gym, user,
     *                 vcheck, npairs, indrow, indcol)
      implicit none
c
      include 'ad_petsc.inc'
c
#define MERGE_TOL 0.4
c
c compute the ADIFOR derivatives by SparsLinC and convert the result to
c FORTRAN sparse format.
c
      integer          nnx, nnf, g_X(*), g_F(*), xs, ys, xm, ym,
```

```
     *                  gxs, gys, gxm, gym, user(*), npairs,
     *                  indrow(*), indcol(*)
      double precision  X(*), F(*), g$X(g$pmax$, *), g$F(g$pmax$, *)
      logical           vcheck
c
      integer           info, ierr, nnz1, ndiff1, ndiff2, i,
     *                  iwa(liwa), ir1(max_nnz), ic1(max_nnz)
      double precision  one, diff
      character*8       ad_type
c

#ifdef OUT_TIME
c
      double precision  proc_time1(3), proc_time2(3)

      call second(proc_time1)
#endif

      if (npairs .GT. 0) then
         ad_color = .TRUE.

         nnz_total = npairs
         call copy_vec(ir, indrow, nnz_total)
         call copy_vec(ic, indcol, nnz_total)
         call dsm(nnf, nnx, nnz_total, ir, ic, ngrp, maxgrp, mingrp,
     *            info, ipntr, jpntr, iwa, liwa)

#ifdef OUT_INFO
c
         write (*, *) ' maxgrp = ', maxgrp, ' info = ', info
#endif

         call ad_seed(nnx, g$X, vcheck)
      endif

      ierr = 0
 9999 if (ad_color) then
         ad_type = 'AD-Color'
         call g$ADFunction(maxgrp1, X, g$X, g$pmax$, F, g$F,
     *                  g$pmax$, xs, ys, xm, ym, gxs, gys, gxm, gym,
     *                  user)
      else
         ad_type = 'AD-SpLnC'
         one = 1.0
         do i = 1, nnx
             call dspsd(g_X(i), i, one, 1)
         enddo
         call g_ADFunction(X, g_X, F, g_F, xs, ys, xm, ym,
     *                  gxs, gys, gxm, gym, user)
      endif

#ifdef OUT_TIME
c
      call second(proc_time2)
      write (*,100) ad_type,
     *            proc_time2(1)-proc_time1(1),
     *            proc_time2(2)-proc_time1(2),
     *            proc_time2(3)-proc_time1(3)
```

```
  100 format (' ', A8, ' Timing: user = ', F5.2, ', system = ', F5.2,
     *        ', wall-clock = ', F5.2)
#endif

      if (ad_color) then
         call conv$der(nnf, ptr_F, id_F, d_F, g$F, g$pmax$, nnz_total,
     *                 g$X, ipntr, ic, ngrp, maxgrp1, vcheck, ierr)

c
c rerun by SparsLinC if error in coloring scheme
c
         if (ierr .eq. 1) then
             ad_color = .FALSE.

#ifdef OUT_INFO
c
             write (*,*) '      Sparse Pattern Error: Re-run SparsLinC'
#endif

#ifdef OUT_TIME
c
             call second(proc_time1)
#endif

             goto 9999
         endif
      else
         call conv_der(nnf, ptr_F, id_F, d_F, g_F, max_nnz, nnz_total)
      endif

#ifdef OUT_TIME
c
      call second(proc_time1)
      write (*,110) proc_time1(1)-proc_time2(1),
     *              proc_time1(2)-proc_time2(2),
     *              proc_time1(3)-proc_time2(3)
  110 format (' Convert  Timing: user = ', F5.2, ', system = ', F5.2,
     *        ', wall-clock = ', F5.2)
#endif

c
c     compute sparse pattern
c
      if (.NOT. ad_color) then
         ad_color = .TRUE.

         call color_ind(nnf, ptr_F, id_F, ir, ic)
         if (ierr .EQ. 1) then
             call merge_ind(nnz0, ir0, ic0, nnz_total, ir, ic,
     *                      nnz1, ir1, ic1, ndiff1, ndiff2)
             diff = 1.0 * (ndiff1 + ndiff2) / nnz0
             if (diff .LE. MERGE_TOL) then

#ifdef OUT_INFO
c
          write (*, 115) 'Use Merged Sparse Pattern', ndiff1,
     *                           ndiff2, nnz0, diff
#endif
```

```
          call copy_vec(ir, ir1, nnz1)
          call copy_vec(ic, ic1, nnz1)
          nnz_total = nnz1
             else

#ifdef OUT_INFO
c
          write (*, 115) 'Use New Sparse Pattern', ndiff1,
     *                        ndiff2, nnz0, diff
#endif

             endif
         endif

#ifdef OUT_INFO
c
  115     format('      ', A, ', minus(', I5, ')+plus(', I5, ')/total(',
     *           I5, ')=', F5.2)
#endif

         call copy_vec(ir0, ir, nnz_total)
         call copy_vec(ic0, ic, nnz_total)
         call dsm(nnf, nnx, nnz_total, ir, ic, ngrp, maxgrp, mingrp,
     *            info, ipntr, jpntr, iwa, liwa)

         nnz0 = nnz_total

#ifdef OUT_INFO
c
         write (*, *) ' maxgrp = ', maxgrp, ' info = ', info
#endif

         call ad_seed(nnx, g$X, vcheck)
      endif
c
      return
      end



      subroutine ad_seed(nnx, g$X, vcheck)
      implicit none
c
c initial seed matrix of coloring scheme
c
      include 'ad_petsc.inc'
c
c
      integer           nnx
      double precision  g$X(g$pmax$, *)
      logical           vcheck
c
      integer           i, j
c
c
c initial seed matrix of coloring scheme
c
```

```
      do i = 1, nnx
         do j = 1, g$pmax$
             g$X(j, i) = 0.0
         enddo
      enddo

      do i = 1, nnx
         g$X(ngrp(i), i) = 1.0
      enddo
c
c extra-column seed to verify derivative value
c
      if (vcheck) then
         maxgrp1 = maxgrp + 1
         do i = 1, nnx
             call get_random(g$X(maxgrp1, i))
         enddo
      else
         maxgrp1 = maxgrp
      endif
c
      return
      end



      subroutine copy_vec(vecA, vecB, no)
      implicit none
c
c copy vector, vecA = vecB
c
      integer   vecA(*), vecB(*), no
c
      integer   i
c
      do i = 1, no
         vecA(i) = vecB(i)
      enddo
      return
      end



      subroutine color_ind(m, ptr_d, id_d, indrow, indcol)
      implicit none
c
c compute sparse index vectors indrow & indcol from ptr_d & id_d
c
      integer   m, ptr_d(*), id_d(*), indrow(*), indcol(*)
c
      integer   nstart, nend, i, j

      do i = 1, m
         nstart = ptr_d(i)
         nend   = ptr_d(i+1) - 1
         do j = nstart, nend
             indrow(j) = i
             indcol(j) = id_d(j)
```

```
         enddo
      enddo

      return
      end



      subroutine merge_ind(nnzA, irA, icA, nnzB, irB, icB,
     *                     nnzC, irC, icC, ndiff1, ndiff2)
      implicit none
c
c merge 2 sparse index pairs (irA,icA) & (irB,icB) into pair (irC, icC)
c where ndiff1 = # of A - B & ndiff2 = # of B - A
c
      integer   nnzA, irA(*), icA(*), nnzB, irB(*), icB(*),
     *          nnzC, irC(*), icC(*), ndiff1, ndiff2
c
      integer   indA, indB, indC, ir1, ir2, ic1, ic2, i
c
      ndiff1 = 0
      ndiff2 = 0
      indA = 1
      indB = 1
      indC = 1
 9999 if     (indA .GT. nnzA) then
         do i = indB, nnzB
             irC(indC) = irB(indB)
             icC(indC) = icB(indB)
             indC = indC + 1
             indB = indB + 1
             ndiff2 = ndiff2 + 1
         enddo
      elseif (indB .GT. nnzB) then
         do i = indA, nnzA
             irC(indC) = irA(indA)
             icC(indC) = icA(indA)
             indC = indC + 1
             indA = indA + 1
             ndiff1 = ndiff1 + 1
         enddo
      else
         ir1 = irA(indA)
         ir2 = irB(indB)
         if     (ir1 .LT. ir2) then
   10         irC(indC) = ir1
             icC(indC) = icA(indA)
             indC = indC + 1
             indA = indA + 1
             ndiff1 = ndiff1 + 1
             if ((indA .LE. nnzA) .AND. (irA(indA) .EQ. ir1)) then
          goto 10
             endif
         elseif (ir1 .GT. ir2) then
   20         irC(indC) = ir2
             icC(indC) = icB(indB)
             indC = indC + 1
             indB = indB + 1
```

```
            ndiff2 = ndiff2 + 1
            if ((indB .LE. nnzB) .AND. (irB(indB) .EQ. ir2)) then
         goto 20
            endif
         else
            ic1 = icA(indA)
            ic2 = icB(indB)
            if     (ic1 .LT. ic2) then
         irC(indC) = ir1
         icC(indC) = ic1
         indC = indC + 1
         indA = indA + 1
         ndiff1 = ndiff1 + 1
            elseif (ic1 .GT. ic2) then
         irC(indC) = ir2
         icC(indC) = ic2
         indC = indC + 1
         indB = indB + 1
         ndiff2 = ndiff2 + 1
            else
         irC(indC) = ir1
         icC(indC) = ic1
         indC = indC + 1
         indA = indA + 1
         indB = indB + 1
            endif
         endif
         goto 9999
      endif
      nnzC = indC - 1
c
      return
      end



      subroutine conv_der(nnf, ptr_d, id_d, d_d, g_d, max_nnz, nnz)
      implicit none
c
c convert result in the SparsLinC format into FORTRAN sparse format.
c
      integer          nnf, ptr_d(*), id_d(*), g_d(*), max_nnz, nnz
      double precision d_d(*)
c
      integer          offset, outlen, info, i

      offset = 1
      do i = 1, nnf
         ptr_d(i) = offset
         call dspxsq(id_d(offset), d_d(offset), max_nnz, g_d(i),
     *               outlen, info)
         offset = offset + outlen
      enddo
      ptr_d(nnf+1) = offset
      nnz = offset - 1

      return
      end
```

```
      subroutine conv$der(nnf, ptr_d, id_d, d_d, g$d, max_p, nnz,
     *                    g$x, ipntr, ic, ngrp, maxgrp1, vcheck, ierr)
      implicit none
c
#define SMALL_NUM 1.d-10
#define ERROR_TOL 1.d-2
c
c convert result in the Coloring DSM format into FORTRAN sparse format.
c
      integer          nnf, ptr_d(*), id_d(*), max_p, nnz,
     *                 ipntr(*), ic(*), ngrp(*), maxgrp1, ierr
      double precision d_d(*), g$d(max_p, *), g$x(max_p, *)
      logical          vcheck
c
      integer          offset, i, j, k
      double precision der, chk_val, vchk, diff

      ierr   = 0
      offset = 1

      if (vcheck) then
         do i = 1, nnf
            ptr_d(i) = offset
            chk_val  = 0.0
            do j = ipntr(i), ipntr(i+1) - 1
	       k   = ic(j)
	       der = g$d(ngrp(k), i)
	       if (der .ne. 0.0) then
		   id_d(offset) = k
		    d_d(offset) = der
		   chk_val = chk_val + der * g$x(maxgrp1, k)
		   offset = offset + 1
	       endif
            enddo

            diff = abs(g$d(maxgrp1, i) - chk_val)
            vchk = abs(chk_val)
            if (vchk .lt. SMALL_NUM) then
	       if (diff .gt. ERROR_TOL) then
		   ierr = 1
		   return
	       endif
            else if ((diff / vchk) .gt. ERROR_TOL) then
	       ierr = 1
	       return
            endif
         enddo
      else
         do i = 1, nnf
            ptr_d(i) = offset
            do j = ipntr(i), ipntr(i+1) - 1
	       k   = ic(j)
	       der = g$d(ngrp(k), i)
	       if (der .ne. 0.0) then
		   id_d(offset) = k
```

```
             d_d(offset) = der
            offset = offset + 1
        endif
           enddo
       enddo
      endif

      ptr_d(nnf+1) = offset
      nnz = offset - 1

      return
      end
```

G ADIC FormJacobian

```
/* ------------------------------------------------------------------- */
/*
   FormJacobian - Evaluates Jacobian matrix.

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  ptr - optional user-defined context, as set by SNESSetJacobian()

   Output Parameters:
.  A - Jacobian matrix
.  B - optionally different preconditioning matrix
.  flag - flag indicating matrix structure

   Notes:
   Due to grid point reordering with DAs, we must always work
   with the local grid points, and then transform them to the new
   global numbering with the "ltog" mapping (via DAGetGlobalIndices()).
   We cannot work directly with the global numbers for the original
   uniprocessor grid!
*/
int FormJacobian(SNES snes,Vec X,Mat *J,Mat *B,MatStructure *flag,void *ptr)
{
  AppCtx  *user = (AppCtx *) ptr;  /* user-defined applicatin context */
  Mat     jac = *J;                /* Jacobian matrix */
  Vec     localX = user->localX, localF = user->localF; /* local vector */
  int     *ltog;                   /* local-to-global mapping */
  int     ierr, i, j;
  int     nloc, xs, ys, xm, ym, gxs, gys, gxm, gym;
  Scalar  *x;

  DERIV_TYPE    ad_x[max_x], ad_f[max_f] ;
  double        der[max_x] ;
  int           ind[max_x] ;
  double        grad ;
  int           nsize_x, nsize_f, outlen ;
  ad_AppCtx     ad_user ;

  /*
     Scatter ghost points to local vector, using the 2-step process
        DAGlobalToLocalBegin(), DAGlobalToLocalEnd()
     Computations can be done while messages are in transition,
```

```
   by placing code between these two statements.
*/
ierr = DAGlobalToLocalBegin(user->da,X,INSERT_VALUES,localX); CHKERRQ(ierr);
ierr = DAGlobalToLocalEnd(user->da,X,INSERT_VALUES,localX); CHKERRQ(ierr);

/*
   Get pointer to vector data
*/
ierr = VecGetArray(localX,&x); CHKERRQ(ierr);

/*
   Get local grid boundaries
*/
ierr = DAGetCorners(user->da,&xs,&ys,PETSC_NULL,&xm,&ym,PETSC_NULL);
CHKERRQ(ierr);
ierr = DAGetGhostCorners(user->da,&gxs,&gys,PETSC_NULL,&gxm,&gym,PETSC_NULL);
CHKERRQ(ierr);

/*
   Get the global node numbers for all local nodes, including ghost points
*/
ierr = DAGetGlobalIndices(user->da,&nloc,&ltog); CHKERRQ(ierr);

ierr = VecGetLocalSize(localX, &nsize_x); CHKERRQ(ierr);
ierr = VecGetLocalSize(localF, &nsize_f); CHKERRQ(ierr);

/*
   Compute entries for the locally owned part of the Jacobian.
    - Currently, all PETSc parallel matrix formats are partitioned by
      contiguous chunks of rows across the processors. The "grow"
      parameter computed below specifies the global row number
      corresponding to each local grid point.
    - Each processor needs to insert only elements that it owns
      locally (but any non-local elements will be sent to the
      appropriate processor during matrix assembly).
    - Always specify global row and columns of matrix entries.
    - Here, we set all entries for a particular row at once.
*/

ad_INIT_VAR_ARRAY(ad_x, x, nsize_x) ;

ad_INIT_VAR(ad_user.param, user->param) ;
ad_user.mx     = user->mx ;
ad_user.my     = user->my ;
ad_user.localX = user->localX ;
ad_user.localF = user->localF ;
ad_user.da     = user->da ;
ad_user.rank   = user->rank ;

ad_INIT_INDEPENDENTS() ;
ad_SET_INDEPENDENT_ARRAY(ad_x, nsize_x) ;
ad_ADFunction(ad_x, ad_f, xs, ys, xm, ym, gxs, gys, gxm, gym, &ad_user) ;

for (i = 0 ; i < nsize_f ; i++) {
  outlen = 0 ;
  for (j = 0 ; j < nsize_x ; j++) {
    ad_GET_DERIV(grad, ad_f[i], j) ;
    if (grad != 0.0) {
```

```
        der[outlen] = grad ;
        ind[outlen] = ltog[j] ;
        ++outlen ;
       }
     }
     if (outlen != 0) {
       ierr = MatSetValues(jac,1,&ltog[i],outlen,ind,der,INSERT_VALUES); CHKERRQ(ierr);
     }
   }

   /*
      Assemble matrix, using the 2-step process:
        MatAssemblyBegin(), MatAssemblyEnd()
      Computations can be done while messages are in transition,
      by placing code between these two statements.
   */
   ierr = MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
   ierr = VecRestoreArray(localX,&x); CHKERRQ(ierr);
   ierr = MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

   /*
      Set flag to indicate that the Jacobian matrix retains an identical
      nonzero structure throughout all nonlinear iterations (although the
      values of the entries change). Thus, we can save some work in setting
      up the preconditioner (e.g., no need to redo symbolic factorization for
      ILU/ICC preconditioners).
       - If the nonzero structure of the matrix is different during
         successive linear solves, then the flag DIFFERENT_NONZERO_PATTERN
         must be used instead.  If you are unsure whether the matrix
         structure has changed or not, use the flag DIFFERENT_NONZERO_PATTERN.
       - Caution:  If you specify SAME_NONZERO_PATTERN, PETSc
         believes your assertion and does not check the structure
         of the matrix.  If you erroneously claim that the structure
         is the same when it actually is not, the new preconditioner
         will not function correctly.  Thus, use this optimization
         feature with caution!
   */
   *flag = SAME_NONZERO_PATTERN;
   return 0;
 }
```

References

[1] B. Averick, J. Moré, C. Bischof, A. Carle, and A. Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[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 for Scientific Computing*. Birkhauser Boston, Cambridge, MA, 1997.

[3] 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.

[4] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR 2.0 user's guide (Revision C). Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[5] C. Bischof and L. Roh. ADIC user guide, 1996. Unpublished information, Argonne National Laboratory.

[6] C. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. Preprint ANL/MCS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.