UCRL-MA-140211

User Documentation for SensPVODE, a Variant of PVODE for Sensitivity Analysis

S.L. Lee, A.C. Hindmarsh and P.N. Brown

August 22, 2000

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Approved for public release; further dissemination unlimited

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Work performed under the auspices of the U. S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

USER DOCUMENTATION FOR SENSPVODE, A VARIANT OF PVODE FOR SENSITIVITY ANALYSIS*

STEVEN L. LEE , ALAN C. HINDMARSH AND PETER N. BROWN†

1. Introduction. SensPVODE and PVODE [3] are general-purpose ordinary differential equation (ODE) solvers for stiff and nonstiff initial-value problems. SensPVODE, however, is a variant of PVODE that includes options for simultaneously computing the ODE solution together with its first-order sensitivity coefficients with respect to model parameters. Both codes are written in ANSI-standard C for portability and use MPI (Message-Passing Interface [6]) to achieve parallelism. Furthermore, these codes are based on CVODE [4, 5] and are intended for a distributed memory SPMD (Single Program Multiple Data) environment in which all vectors are identically partitioned across processors. To implement this SPMD model, SensPVODE and PVODE use special versions of the modules contained in CVODE. For example, the revised `NVECTOR` vector module is designed to help the user assign a continuous segment of a given vector to each of the processors for parallel computation. The idea is for each processor to solve a certain fixed subset of the ODEs that describe the model problem and the first-order sensitivity coefficients of the solution.

SensPVODE includes all of the essential numerical methods contained in PVODE: backward differentiation formulas (BDF) or Adams-Moulton formulas for time integration; Inexact Newton methods or functional iteration for solving nonlinear equations; Krylov subspace methods (i.e., GMRES [2]) and preconditioning modules for solving linear systems. The linear solver and preconditioning modules allow for other Krylov methods to be easily included and for user-supplied preconditioners to be added. SensPVODE also retains the use of matrix-free methods [1] for approximating preconditioned matrix-vector products. As in PVODE, this approach obtains matrix-vector products within GMRES without explicitly computing and/or storing the linear system matrix. In contrast to PVODE, the current version of SensPVODE does not contain a direct method for solving linear systems. However, in a future release, the PVODE diagonal linear solver module `CVDIAG` may be added so that the two packages have identical linear solver capabilities. SensPVODE was developed and tested on a cluster of Sun-SPARC workstations.

The remainder of this document is organized as follows: Section 2 sets the mathematical notation and summarizes the basic approach to sensitivity analysis. Section 3 summarizes the organization of the SensPVODE solver, while Section 4 summarizes its usage. Section 5 describes a set of Fortran/C interface routines. Section 6 describes two example problems. Finally, Section 7 discusses the availability of SensPVODE.

2. Mathematical Considerations. In many complex, large-scale, computational simulations, the governing equations can often be spatially discretized and then numerically

* This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

† Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551

solved as a system of ODEs. Typically, these equations contain parameter values (e.g., chemical reaction rates) that are not precisely known. In addition to numerically solving the ODEs, it may be desirable to determine the sensitivity of the results with respect to the model parameters. Such sensitivity information is useful because it indicates which parameters are most influential in affecting the behavior of the simulation.

SensPVODE is a variant of PVODE that computes the first-order sensitivity of the ODE solution with respect to some or all of its model parameters. When computing sensitivities in this context, one is interested in solving the ODE system

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

where the solution vector $y(t)$ depends upon an additional vector of parameters p. The *sensitivities* are defined as

$$s_i(t) = \frac{\partial y(t)}{\partial p_i}, \; i = 1, \cdots, m,$$

and the equations for the sensitivities are obtained by differentiating the original ODE with respect to each component p_i of p. Thus, we have

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

The initial sensitivity vector $s_i(t_0)$ is either all zeros (if p_i occurs only in f), or has nonzeros according to how $y_0(p)$ depends on p_i. Given this preliminary formulation, we remark that PVODE can now be used to solve the initial-value problem for the sensitivities (2) in tandem with the ODEs for the model problem (1).

Several observations motivate a modification of the sensitivity ODEs (2) just derived. The first is the fact that the units for the ODE solution $y(t)$ and the sensitivity vectors $s_i(t)$ do not match. This mismatch in units can lead to scaling problems when attempting to numerically integrate this combined system of ODEs. In particular, we note that the sensitivity vectors will have units of $[y]/[p_i]$. One remedy is to scale each sensitivity vector $s_i(t)$ by $\overline{p}_i$, a nonzero constant that is dimensionally consistent with p_i. By doing so, we obtain the scaled sensitivity vector

$$w_i(t) = \overline{p}_i s_i(t), \tag{3}$$

where, typically, $\overline{p}_i = p_i$. By differentiating these scaled vectors with respect to time, we obtain the *scaled* sensitivity ODEs

$$w_i'(t) = \overline{p}_i s_i'(t) = \overline{p}_i \left(\frac{\partial f}{\partial y} s_i(t) + \frac{\partial f}{\partial p_i} \right) = \frac{\partial f}{\partial y} w_i(t) + \overline{p}_i \frac{\partial f}{\partial p_i}. \tag{4}$$

SensPVODE carries out the time integration of the combined system, (1) and (4), by viewing it as an ODE system of size $N(m+1)$. By defining

$$Y(t) = \begin{pmatrix} y(t) \\ w_1(t) \\ \vdots \\ w_m(t) \end{pmatrix} \quad \text{and} \quad F(t, Y, p) = \begin{pmatrix} f(t, y, p) \\ \frac{\partial f}{\partial y} w_1(t) + \overline{p}_1 \frac{\partial f}{\partial p_1} \\ \vdots \\ \frac{\partial f}{\partial y} w_m(t) + \overline{p}_m \frac{\partial f}{\partial p_m} \end{pmatrix},$$

the combined system is simply

$$Y'(t) = F(t, Y, p), \qquad Y(t_0) = Y_0(p).$$

For many large-scale applications, implicit time integration methods are necessary. SensPVODE uses backward differentiation formulas (BDFs) of order 1 through 5 to approximate $Y'(t)$. The first-order case is the backward Euler method, and in that case this approach yields the nonlinear system

$$\mathbf{0} = G(Y_{n+1}) \equiv Y_{n+1} - h\,F(t_{n+1}, Y_{n+1}, p) - Y_n \tag{5}$$

where $h = t_{n+1} - t_n$ is the current stepsize. Due to the form of F, the Jacobian matrix $\partial G/\partial Y$ has the lower block triangular structure

$$\frac{\partial G}{\partial Y} = I - h\frac{\partial F}{\partial Y} = \begin{pmatrix} I - hJ & & & \\ -hJ_1 & I - hJ & & \\ \vdots & & \ddots & \\ -hJ_m & & & I - hJ \end{pmatrix}$$

where

$$J = \frac{\partial f}{\partial y} \quad \text{and} \quad J_i = \frac{\partial}{\partial y}\left(Jw_i + \overline{p}_i \frac{\partial f}{\partial p_i}\right).$$

Higher-order BDFs also yield Jacobian matrices $\partial G/\partial Y$ with this same lower block triangular structure, and with identical block-diagonal entries of the form $I - h\beta_0 J$. SensPVODE solves the nonlinear systems $G(Y_{n+1}) = \mathbf{0}$ by using the simultaneous corrector method [7], a technique in which the Newton iteration uses the block-diagonal portion of $\partial G/\partial Y$ as the linear system matrix. This results in a decoupling that allows $I - h\beta_0 J$ to be used repeatedly in solving the $(m+1)$ linear systems that arise: one linear system for the Newton correction to the ODE variables; and m linear systems for the corrections to the m scaled sensitivity vectors. Because all of the Jacobian matrices are identical, the latter systems are solved using the same preconditioner and/or linear system solver that were specified for the original ODE problem (1). In particular, the SPGMR (scaled, preconditioned GMRES) method is used when solving stiff initial-value problems via BDF methods.

The integrator computes an estimate E_n of the local error at each time step, and strives to satisfy the inequality

$$\|E_n\|_{rms,W} < 1\,. \tag{6}$$

Here the weighted root-mean-square (rms) norm is defined by

$$\|E_n\|_{rms,W} = \left[\sum_{i=1}^{N(m+1)} \frac{1}{N(m+1)}\left(W_i E_{n,i}\right)^2\right]^{1/2} \tag{7}$$

where $E_{n,i}$ denotes the ith component of E_n, and the ith component of the weight vector is

$$W_i = \frac{1}{\text{RTOL}|Y_i| + \text{ATOL}_i}\,. \tag{8}$$

This permits an arbitrary combination of relative and absolute error control. The user-specified relative error tolerance is the scalar RTOL; the user-specified absolute error tolerance is ATOL, which may be a scalar or a vector. The value for RTOL indicates the number of digits of relative accuracy for a single time step. The specified value for ATOL_i indicates the values of the corresponding component of the solution vector which may be thought of as being zero, or at the noise level. In particular, if we set $\text{ATOL}_i = \text{RTOL} \times \text{FLOOR}_i$ then FLOOR_i represents the floor value for the ith component of the solution. The magnitude of FLOOR_i is the value for which there is a crossover from relative error control to absolute error control. In the vector case of tolerances for the sensitivity vectors, a typical default is to use the same ATOL as for the ODE variables. Since the tolerances define the allowed error per step, they should be chosen conservatively. Experience indicates that a conservative choice yields a more economical solution than error tolerances that are too large.

For estimating the scaled sensitivity derivatives $w_i'(t)$ in (4), SensPVODE has an option that applies centered differences to each term separately:

$$\frac{\partial f}{\partial y} w_i(t) \approx \frac{f(t, y + \delta_y w_i, p) - f(t, y - \delta_y w_i, p)}{2\,\delta_y} \tag{9}$$

and

$$\overline{p}_i \frac{\partial f}{\partial p_i} \approx \frac{f(t, y, p + \delta_i \overline{p}_i e_i) - f(t, y, p - \delta_i \overline{p}_i e_i)}{2\,\delta_i}. \tag{10}$$

As is typical for finite differences, the proper choice of perturbations δ_y and δ_i is a delicate matter. SensPVODE uses a δ_y and δ_i that takes into account several problem-related features; namely, the relative ODE error tolerance RTOL, the machine unit roundoff $\epsilon_{\text{machine}}$, and the weighted root-mean-square norm of the scaled sensitivity w_i. We then define

$$\delta_i = \sqrt{\max(\text{RTOL}, \epsilon_{\text{machine}})} \quad \text{and} \quad \delta_y = \frac{1}{\max(1/\delta_i, \|w_i\|_{rms,W})}. \tag{11}$$

The terms $\epsilon_{\text{machine}}$ and $1/\delta_i$ are included as divide-by-zero safeguards in case $\text{RTOL} = 0$ or $||w_i|| = 0$. Roughly speaking (i.e., if the safeguard terms are ignored), δ_i gives a $\sqrt{\text{RTOL}}$ relative perturbation to parameter i, and δ_y gives a unit weighted rms norm perturbation to y. Of course, the main drawback of this approach is that it requires four evaluations of $f(t, y, p)$.

Another technique for estimating the scaled sensitivity derivatives via centered differences is

$$w_i'(t) = \frac{\partial f}{\partial y} w_i + \overline{p}_i \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \delta w_i, p + \delta \overline{p}_i e_i) - f(t, y - \delta w_i, p - \delta \overline{p}_i e_i)}{2\,\delta} \tag{12}$$

in which

$$\delta = \min(\delta_i, \delta_y).$$

If $\delta_i = \delta_y$, a Taylor series analysis shows that the sum of (9)–(10) and (12) are equivalent to within $O(\delta^2)$. However, the latter approach is half as costly since it only requires two

evaluations of $f(t, y, p)$. To take advantage of this savings, it may also be desirable to use the latter formula when $\delta_i \approx \delta_y$. SensPVODE accommodates this possibility by allowing the user to specify a threshold parameter $\rho_{\max}$. In particular, if δ_i and δ_y are within a factor of $|\rho_{\max}|$ of each other then (12) is used to estimate the scaled sensitivity derivatives. Otherwise, the sum of (9)–(10) is used since δ_i and δ_y differ by a relatively large amount and the use of separate perturbations is prudent.

These procedures for choosing the perturbations (δ_i, δ_y, δ) and switching ($\rho_{\max}$) between centered difference formulas have also been implemented for first-order, forward difference formulas as well. In the latter case, forward differences can be applied to Jw_i and $\overline{p}_i \frac{\partial f}{\partial p_i}$ separately or the single forward difference

$$w_i'(t) = \frac{\partial f}{\partial y} w_i + \overline{p}_i \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \delta w_i, p + \delta \overline{p}_i e_i) - f(t, y, p)}{\delta} \tag{13}$$

can be used. In SensPVODE, the default value of $\rho_{\max} = 0$ indicates the use of the centered difference (12) exclusively. Otherwise, the magnitude of $\rho_{\max}$ and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

In contrast to the above notation used in describing the mathematical details, the sections that follow use new variable names in explaining the organization, usage, and example programs of SensPVODE. For convenient reference, we define these names as follows:

- `f` is the name of the user-supplied function that computes $y'(t) = f(t, y, p)$
- `Ny` is the number of ODEs contained in `f`
- `Np` is the number of parameters contained in p
- `Ns` is the number of sensitivity vectors to be computed
- `Ntotal` is the total number of ODEs to be solved by SensPVODE; usually `Ntotal` = `Ny` * (1 + `Ns`)
- `u` is the vector of length `Ntotal` that contains the `Ny` ODE variables and `Ns` scaled sensitivity vectors
- `rhomax` is the finite difference threshold parameter $\rho_{\max}$

3. Code Organization. One way to visualize SensPVODE is to think of the code as being organized in layers, as shown in Fig. 1. The user's main program resides at the top level. The main program creates the required data structures, makes various initializations, defines the ODE function `f`, defines the preconditioner setup and solve routines (if any), makes calls to the `SensCVSPGMR` and/or `SENSITIVITY` modules, and calls the `CVODE` module for the solution of the problem. The main program also manages input/output.

At the second level, the `SENSITIVITY` module contains several user-callable routines: `SensCVodeMalloc`, for memory allocation and basic initializations related to sensitivity analysis; `SensCVReInit` for reinitializing SensPVODE to solve a new sensitivity analysis problem of the same size as the one previously solved; `SensCVodeFree`, for memory deallocation; and `SensSetVecAtol`, for the vector case of setting absolute error tolerances for sensitivities. The `SENSITIVITY` routine `FDQ` is called by the `CVODE` module. `FDQ` is responsible for computing $Y'(t) = F(t, Y, p)$ by calling the user's `f` routine to evaluate $y'(t) = f(t, y, p)$ and by using various finite difference formulas to estimate the scaled sensitivity derivatives $w_i'(t)$.

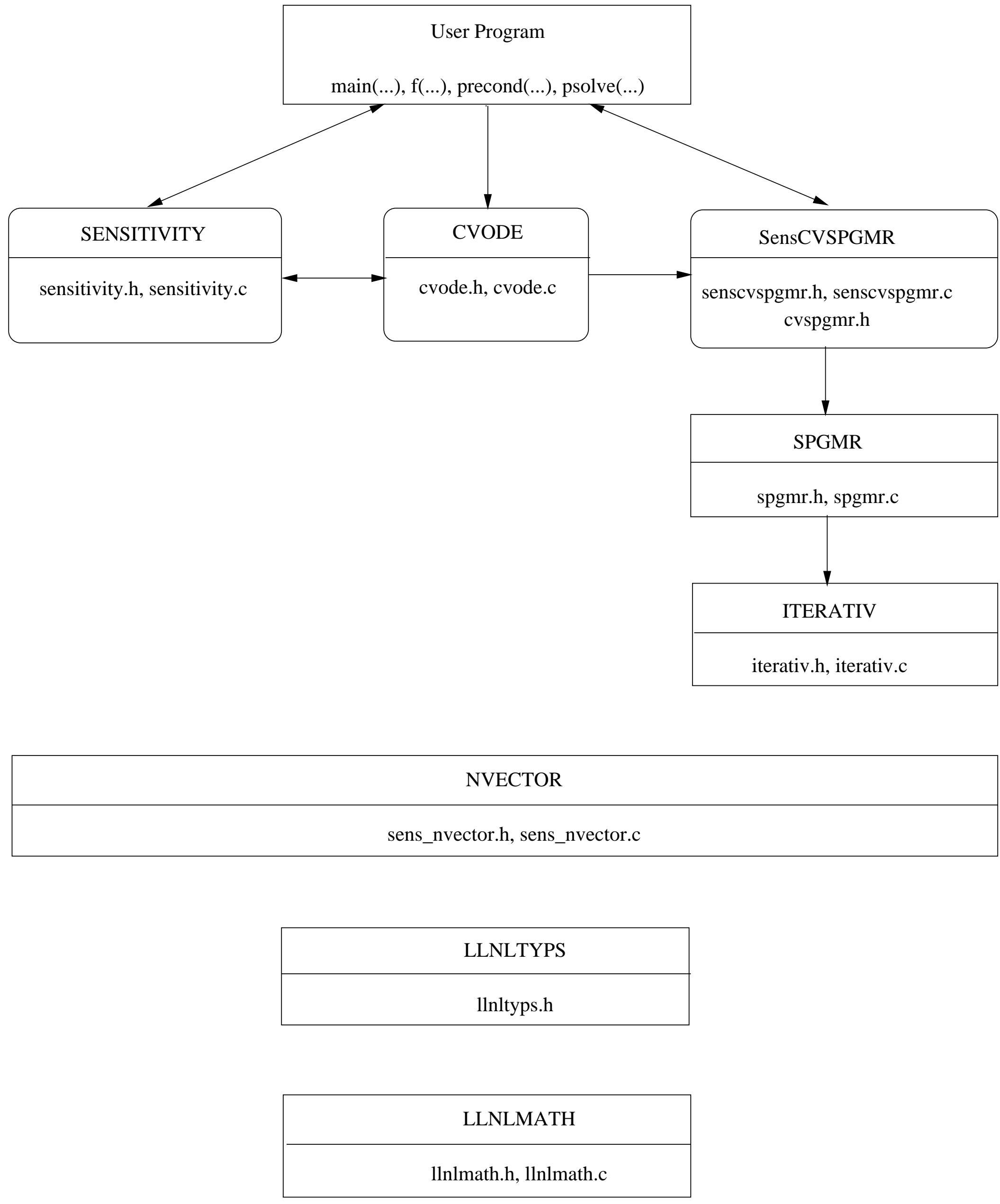

FIG. 1. *Overall structure of the SensPVODE package. Modules comprising the central solver are distinguished by rounded boxes, while the user program, linear solver, and auxiliary modules are in unrounded boxes.*

The SensCVSPGMR module is called by the main program to specify that the simultaneous corrector method, combined with SPGMR, is to be used in solving the linear systems that arise in sensitivity analysis. Of course, this assumes that the Newton method has been specified for the time integration nonlinear iteration. Note that the header file cvspgmr.h is included to specify the various data types, function prototypes and enumerations used in SensCVSPGMR. The module also contains routines which are called by the CVODE module whenever a linear system of size Ntotal needs to be solved. The SPGMR method is called by SensCVSPGMR to solve the (1+Ns) linear systems of size Ny. Note that the SPGMR method consists of the modules SPGMR and ITERATIV. SensCVSPGMR also accesses the user-supplied preconditioner solver routine, if specified, and possibly a user-supplied routine that computes and preprocesses the preconditioner by way of the Jacobian matrix or an approximation to it. Other linear system solvers may be added to SensPVODE in the future. Such additions will be independent of the CVode integrator and SENSITIVITY.

Finally, at the second level, the routine CVode within the CVODE module is used to manage the time integration. CVode makes calls to the SENSITIVITY and SensCVSPGMR modules in order to evaluate the ODE and the scaled sensitivity derivatives, and to solve the linear systems that arise at each Newton iteration.

The following modules reside below the levels just described. The LLNLTYPS module defines types real, integer, and boole (boolean), and facilitates changing the precision of the arithmetic in the package from double to single, or the reverse. The LLNLMATH module specifies power functions and provides a function to compute the machine unit roundoff. Finally, we now describe the NVECTOR module.

In creating SensPVODE from PVODE, we developed a sensitivity version of the NVECTOR module. A revised NVECTOR module is needed because the overall ODE system has length Ntotal, but it consists of 1+Ns ODE subsystems of length Ny; namely, the original nonlinear ODE system (1) and Ns scaled sensitivity ODE systems (4). Several steps are involved in partitioning and distributing the subsystems in a multiprocessor environment. First, each processor is responsible for solving a contiguous portion of each subsystem, of length Nlocal. Note that Nlocal need not be the same for each processor; however, the sum of all the Nlocal values must be Ny. Furthermore, the 1+Ns subsystems of size Ny are identically partitioned among the processors. This implementation is handled through the revised NVECTOR module and its use of abstract data types: type machEnvType, for the machine environment data block (e.g., Nlocal); and type N_Vector, a data structure for the partitioned and distributed vectors just described. To achieve parallelism for any vector operation, each processor performs the operation on its assigned portions of the input vectors, followed by a global reduction where needed. In this way, vector calculations can be performed simultaneously with each processor working on its own block-portions of the vector.

The version of SensPVODE described so far uses MPI (Message Passing Interface [6]) for all inter-processor communication. This achieves a high degree of portability, since MPI is becoming widely accepted as a standard for message passing software. For a different parallel computing environment, some rewriting of the vector module could allow the use of other specific machine-dependent instructions.

4. Using SensPVODE. This section describes the use of SensPVODE. We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines, and of the user-supplied routines. Finally, there are comments on usage under C++.

4.1. Overview of Usage. The following is a skeleton of the user's main program (or calling program) as an application of SensPVODE. The user program is to have these steps in the order indicated. For the sake of brevity, we defer many of the details to the later subsections.

1. The calling program must `#include` several header files so that various data types, macros, and enumerations can be used. These header files include: `llnltyps.h`, `llnlmath.h`, `cvode.h`, `nvector.h`, `mpi.h`; one or more of the files `dense.h`, `band.h`, `cvspgmr.h`, `iterativ.h`, `pvbbdpre.h` associated with the choice of preconditioner and/or linear system solvers; and `sensitivity.h`. Also, the calling program must `#define` or set the integer variables `Ny`, `Np`, `Ns` and `Ntotal`.
2. The calling program must define a pointer to a user-defined data block that is passed to the user's `f` routine. This data block must include a real pointer (e.g., `p`) that points to the array of real parameters used by `f` to evaluate $f(t, y, p)$. For example, if the pointer to the data block has the form `typedef struct {..., real *p;} *f_data;`, then `f_data->p = p;` must point to the real array in which `p[i-1]` = p_i, for $i = 1, \ldots, m$.
3. `MPI_Init(&argc, &argv);` to initialize MPI. Here `argc` and `argv` are the command line argument counter and array received by `main`.
4. `Nlocal =` the local vector length for this processor, and `Ny =` the global vector length for this processor. Note that `Ny` is the sum of all values of `Nlocal`.
5. `machEnv = PVecInitMPI(comm, Nlocal, Ny, &argc, &argv);` to initialize the `NVECTOR` module. Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active processors is to be used, `comm` must be set by suitable MPI calls. Otherwise, to specify that all processors are to be used, `comm` must be `MPI_COMM_WORLD`.
6. The calling program must declare a real pointer (e.g., `pbar`) and set an array of real values `pbar[i]` that are used to scale the `Ns` sensitivity vectors. Each `pbar[i]` must be set to a nonzero constant that is dimensionally consistent with `p[i]`. Typically, `pbar[i]=p[i]` whenever `p[i]` is nonzero.
7. Set the vector `u` of `Ntotal` initial values. If an existing data array `udata` contains the initial values of `u`, then call `u = SensN_VMAKE(Ntotal, udata, machEnv);`. Otherwise, make the call `u = N_VNew(Ntotal, machEnv);`. Conceptually, `u` consists of `(1+Ns)` vectors of length `Ny`. To load the ith sensitivity vector, use `usub = N_VSUB(u); N_VDATA(usub[i]) = wdata;` where `wdata` is an existing data array of length `Ny`. Note that `usub[0]` is a pointer to the `N_Vector` of ODE variables, and that `usub[i]` is a pointer to the `N_Vector` for the ith scaled sensitivity vector.
8. For the case of vector tolerances for absolute error control, a typical default is to use the same `atol` as for the ODE variables. Use `SensSetVecAtol(atol, Ns);` to do this. Then, alter the resulting vector elements if desired.

9. `cvode_mem = SensCVodeMalloc(...);` allocates internal memory for CVODE, initializes CVODE, and returns a pointer to the CVODE memory structure. (See details below.)
10. `SensCVSpgmr(...);` if Newton iteration is chosen. (See details below.)
11. `ier = CVode(cvode_mem, tout, u, &t, itask);` for each point `t = tout` at which output is desired. Set `itask` to `NORMAL` to have the integrator overshoot `tout` and interpolate, or `ONE_STEP` to take a single step and return. The *unscaled* sensitivity vector `s_i` is obtained by multiplying `usub[i]` by the reciprocal of `pbar[i];`. To do this, call `N_VScale(1.0/pbar[i], usub[i], s_i);`.
12. `SensCVodeFree(cvode_mem);` to free the memory allocated for CVODE.
13. The memory that was created for the vector `u` in Step 7 must be deallocated: call `N_VFree(u);` if `u` was allocated by `u = N_VNew(...);`, or the user must call `SensN_VDISPOSE(Ntotal, u);` if `u` was allocated by `u = SensN_VMAKE(...);`.
14. Before freeing the pointer to the user-defined data block `f_data`, release the arrays containing the scale factors `pbar` and the real parameters `p`: `free(pbar); free(f_data->p); free(f_data);`.
15. `PVecFreeMPI(machEnv);` to free machine-dependent data.
16. `MPI_Finalize();`

The form of the call to `SensCVodeMalloc` is

```
cvode_mem = SensCVodeMalloc(Ny, Ns, Ntotal, f, t0, u0, lmm, iter,
                  itol, &rtol, atol, f_data, errfp, optIn, iopt,
                  ropt, machEnv, p, pbar, rhomax)
```

Except for a few additions, the arguments in `SensCVodeMalloc` are the same as for the PVODE routine `CVodeMalloc`: the integer variables (`Ny`, `Ns`, `Ntotal`) and the real pointers (`p`, `pbar`) are described above; and the real variable `rhomax` is the finite difference threshold parameter ($\rho_{\max}$)—see the description relating (12) to (13) in §2. `f` is the C function to compute $f(t, y, p)$, `t0` is the initial value of t, and `u0` is the vector of length `Ntotal` containing the initial values of y (which can be the same as the vector `u` described above). The flag `lmm` is used to select the linear multistep method and may be one of two possible values: `ADAMS` or `BDF`. The type of iteration is selected by replacing `iter` with either `NEWTON` or `FUNCTIONAL`. The next three parameters are used to set the error control. The flag `itol` is replaced by either `SS` or `SV`, where `SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `SV` indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the ODE. The arguments `&rtol` and `atol` are pointers to the user's error tolerances, and `f_data` is a pointer to the user-defined data block passed directly to the user's `f` function. The file pointer `errfp` points to the file where error messages from SensPVODE are to be written (`NULL` for `stdout`). `iopt` and `ropt` are integer and real arrays for optional input and output. If `optIn` is replaced by `FALSE`, then the user is not going to provide optional input, while if it is `TRUE` then optional inputs are examined in `iopt` and `ropt`. `machEnv` is a pointer to machine environment-specific information.

The form of the call to `SensCVSpgmr` is

```
SensCVSpgmr(cvode_mem, pretype, gstype, maxl, delt, Precond,
```

```
        PSolve, P_data)
```
The arguments to `SensCVSpgmr` are the same as for `CVSpgmr`.

4.2. Reinitialization routines: `CVReInit` and `SensCVReInit`. The current version of PVODE now contains a reinitialization routine `CVReInit` that prepares to solve additional ODE problems using the memory initially allocated by `CVodeMalloc`. The reinitialization routine assumes that the size of the ODE problem remains the same, and that the initial memory allocation is sufficient for each new problem. The form of the call to `CVReInit` is

```
ier = CVReInit(cvode_mem, f, t0, y0, lmm, iter, itol, &rtol, atol,
          f_data, errfp, optIn, iopt[], ropt[], machEnv)
```

Its arguments have names and meanings identical to those of `SensCVodeMalloc`.

Analogously, the sensitivity version of `CVReInit` allows a sequence of sensitivity analysis problems to be solved—as long as the overall ODE problem size (`Ntotal`) remains the same, and the initial memory allocation is sufficient for each new problem. Except for Step 9 in §4.1, the reinitialization process requires Steps 7–11 to be repeated. In Step 9, `SensCVReInit` can be used instead of `SensCVodeMalloc`. The argument list for `SensCVReInit` is similar to the one for `SensCVodeMalloc`, except `cvode_mem` replaces the three arguments `Ny`, `Ns`, and `Ntotal` in the latter. The call has the form

```
ier = SensCVReInit(cvode_mem, f, t0, y0, lmm, iter, itol, &rtol, atol,
        f_data, errfp, optIn, iopt[], ropt[], machEnv, p, pbar, rhomax)
```

4.3. User-Supplied Functions. The user-supplied routines consist of one function defining the ODE, and (optionally) one or two functions that define the preconditioner for use in the SPGMR algorithm. All of the specifications are the same as when using PVODE; however, recall that `Ny` refers to the number of ODEs contained in `f`, and the user-supplied data structure `f_data` contains a pointer (e.g., `p`) that points to the array of real parameters used in `f`.

The first user-supplied C function must be of type `RhsFn`, and in the form

```
void f(integer Ny, real t, N_Vector y, N_Vector ydot, void *f_data)
```

This function includes as input the value of the independent variable `t`, and dependent variable vector `y`. The computed value of $f(t, y, p)$ is stored in `ydot`. There is no return value for a `RhsFn`.

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P may be either a left or a right preconditioner matrix. This C function must be of type `CVSpgmrPSolveFn`. The `Psolve` function has the following form:

```
int Psolve(integer Ny, real t, N_Vector y, N_Vector fy, N_Vector vtemp,
       real gamma, N_Vector ewt, real delta, long int *nfePtr,
       N_Vector r, int lr, void *P_data, N_Vector z)
```

Its input includes `t`, the current value of the independent variable; `y`, the current value of the dependent variable vector; `fy`, the current vector $f(t, y, p)$; `vtemp`, a pointer to memory allocated as an `N_vector` workspace; and `gamma`, the current value of the scalar $h\beta_0$ in the Jacobian matrix $I - h\beta_0 J$. Further input parameters are `ewt`, the error weight vector; `delta`, an input tolerance if `Psolve` is to use an iterative method; `nfePtr`, a pointer to the SensPVODE data `nfe`, the number of calls to the `f` routine; `r`, the right hand side vector in

the linear system; `lr`, an input flag set to `1` to indicate a left preconditioner or `2` for a right preconditioner. `P_data` is the pointer to the user preconditioner data passed to `SensCVSpgmr`. The only output argument is `z`, the vector computed by `Psolve`. The integer returned value is to be negative if the `Psolve` function failed with an unrecoverable error, 0 if `Psolve` was successful, or positive if there was a recoverable error.

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then this needs to be done in the optional user-supplied C function `Precond`. The `Precond` function has the form:

```
int Precond(integer Ny, real t, N_Vector y, N_Vector fy, boole jok,
      boole *jcurPtr, real gamma, N_Vector ewt, real h, real uround,
      long int *nfePtr, void *P_data, N_Vector vtemp1, N_Vector vtemp2,
      N_Vector vtemp3)
```

The arguments which have not been discussed previously are the following. The input flag `jok` indicates whether or not Jacobian-related data needs to be recomputed. If `jok == FALSE`, then it is to be recomputed from scratch. If `jok == TRUE`, and Jacobian-related data was saved from the previous call to `Precond`, then the data can be reused with the current value of `gamma`. The parameter `jcurPtr` is a pointer to a boolean output flag to be set by `Precond`. Set `*jcurPtr == TRUE` if the Jacobian data was recomputed, and set `*jcurPtr == FALSE` if the Jacobian data was not recomputed and saved data was reused. The last three arguments are temporary `N_vectors` available for workspace. The current stepsize `h` and unit roundoff `uround` are supplied for possible use in difference quotient calculations.

4.4. Band-Block-Diagonal Preconditioner Module. SensPVODE has the same `PVBBDPRE` preconditioner module that is included in PVODE. This preconditioner was developed to treat a rather broad class of problems based on solving partial differential equations (PDEs) using a method of lines approach. The modules generate a preconditioner that is a block-diagonal matrix, and each block contains a band matrix. The blocks need not have the same number of super- and sub-diagonals; these numbers may vary from block to block; the preconditioner matrix is of size `Ny`. The PVODE user's guide [3] gives a complete description of this preconditioning technique. However, the basic idea involves defining a new function $g(t, y, p)$ with two key properties: it approximates well the ODE right-hand side function $f(t, y, p)$; and the local dependencies of g on y provide a good approximation to the Jacobian matrix $\partial f/\partial y$.

To use this `PVBBDPRE` module with SensPVODE, the user must supply two functions which the module calls to construct the preconditioner matrix P. These are in addition to the user-supplied ODE function `f`.

- A function `gloc(Nlocal, t, ylocal, glocal, f_data)` must be supplied by the user to compute $g(t, y, p)$. It loads the real array `glocal` as a function of `t`, `ylocal`, and the parameters p contained in `f_data`.
- A function `cfn(Nlocal, t, y, f_data)` which must be supplied to perform all inter-processor communications necessary for the execution of the `gloc` function, using the input vector `y` of type `N_Vector`.

Both functions take as input the same pointer `f_data` as that passed by the user to `SensCVodeMalloc` and passed to the user's function `f`; neither function has a return value.

The user is responsible for providing space (presumably within `f_data`) for components of `y` that are communicated by `cfn` from the other processors, and that are then used by `gloc`. The function `gloc` is not expected to do any communication.

The usage of the `PVBBDPRE` module requires: (a) an additional call to `PVBBDAlloc` to supply required parameters; and (b) passing specific names for the preconditioning routines in the call to `SensCVSpgmr`. See [3] for details.

4.5. Use by a C++ Application. SensPVODE is written in a manner that permits it to be used by applications written in C++ as well as in C. For this purpose, each SensPVODE header file is wrapped with conditionally compiled lines reading `extern "C"` {...}, conditional on the variable `__cplusplus` being defined. This directive causes the C++ compiler to use C-style names when compiling the function prototypes encountered. Users with C++ applications should also be aware that we have defined, in `llnltyps.h`, a boolean variable type, `boole`, since C has no such type. The type `boole` is equated to type `int`, and so arguments in user calls, or calls to user-supplied routines, which are of type `boole` can be typed as either `boole` or `int` by the user. The same applies to vector kernels which have a type `boole` return value, if the user is providing these kernels. The name `boole` was chosen to avoid a conflict with the C++ type `bool`.

5. The Fortran/C Interface Package for SensPVODE. We anticipate that many users of SensPVODE will work from existing Fortran application programs. To accommodate them, we have provided a set of interface routines that make the required connections to SensPVODE with a minimum of changes to the application programs. Specifically, a Fortran/C interface package called SensFPVODE is a collection of C language functions and header files which enables the user to write a main program and all user-supplied subroutines in Fortran and to use the C language SensPVODE package. This package entails some compromises in portability, but we have kept these to a minimum by requiring fixed names for user-supplied routines, and by using a name-mapping scheme to set the names of externals in the Fortran/C linkages. The latter depends on two parameters, set in a small header file, which determine whether the Fortran external names are to be in upper case and whether they are to have an underscore character prefix. The Fortran/C interfaces have been tested on a cluster of Sun-SPARC workstations.

The usage of this module is summarized below. The argument lists are the same as for FPVODE, except that the user-supplied functions must now attach the array of real parameters to their call sequence. Further details can be found in the header file `sensfpvode.h`. Also, the user should check and, if necessary, reset the parameters in the file `fcmixpar.h`. The functions which are callable from the user's Fortran program are as follows:

- `FPVINITMPI` interfaces with `PVecInitMPI` and is used to initialize the `NVECTOR` module.
- `SFPVMALLOC` interfaces with `SensCVodeMalloc` and is used to initialize `CVode`.
- `SFPVREINIT` interfaces with `SensCVReInit` and is used to reinitialize `CVode` for solving additional sensitivity analysis problems of size `Ntotal`. This routine does no memory allocation, and assumes that the internal memory allocation created by `SFPVMALLOC` is sufficient for solving each new problem.

- SFCVSPGMR0, SFCVSPGMR1, SFCVSPGMR2 interface with SensCVSpgmr when SPGMR has been chosen as the linear system solver. These three interface routines correspond to the cases of no preconditioning, preconditioning with no saved matrix data, and preconditioning with saved matrix data, respectively.
- FCVODE interfaces with CVode.
- FCVDKY interfaces with CVodeDky and is used to compute a derivative of order k, $0 \leq k \leq$ qu, where qu is the order used for the most recent time step. The derivative is calculated at the current output time.
- SFCVFREE interfaces with SensCVodeFree and is used to free memory allocated for CVode.
- FPVFREEMPI interfaces with PVecFreeMPI and is used to free memory allocated for MPI.

The user-supplied Fortran subroutines are as follows. The names of these routines are fixed and are case-sensitive. Note that the array of real parameters, P or PAR, has been added onto the end of the call sequence.

- PVFUN which defines the function f, the right-hand side function of the system of ODEs. It has the form

```
      SUBROUTINE PVFUN (NLOC, T, Y, YDOT, P)
```

- PVPSOL which solves the preconditioner equation, and is required if preconditioning is used. It has the form

```
      SUBROUTINE PVPSOL (NLOC, T, Y, FY, VT, GAMMA, EWT, DELTA, NFE,
     1                   R, LR, Z, IER, PAR)
```

- PVPRECO which computes the preconditioner, and is required if preconditioning involves precomputed matrix data. It has the form

```
      SUBROUTINE PVPRECO (NLOC, T, Y, FY, JOK, JCUR, GAMMA, EWT, H,
     1                    UROUND, NFE, V1, V2, V3, IER, PAR)
```

A similar interface package, called SFPVBBD, has been written for the PVBBDPRE preconditioner module. It works in conjunction with the FPVODE interface package. The three additional user-callable functions here are: SFPVBBDIN, which interfaces with PVBBDAlloc and SensCVSpgmr; FPVBBDOPT, which accesses optional outputs; and FPVBBDF, which interfaces to PVBBDFree. The two user-supplied Fortran subroutines required, in addition to PVFUN to define f, are: PVLOCFN, which computes $g(t, y, p)$; and PVCOMMF, which performs the inter-processor communications needed by PVLOCFN.

6. Example Problems. Two sensitivity analysis test problems are described here. These problems are modified versions of the example problems presented in the PVODE user's guide [3]. The first example is a nonstiff problem that contains 2 parameters. The second example is a stiff problem that contains 8 parameters; however, we only compute the sensitivities with respect to 2 of the parameters. Both problems involve the method of lines solution of a partial differential equation (PDE).

6.1. Example problem 1 - A nonstiff PDE problem. This problem begins with a prototypical diffusion-advection equation for $u = u(t, x)$

$$\frac{\partial u}{\partial t} = p_1 \frac{\partial^2 u}{\partial x^2} + p_2 \frac{\partial u}{\partial x} \tag{14}$$

for $0 \le t \le 5$, $\quad 0 \le x \le 2$, $p_1 = 1.0$ and $p_2 = 0.5$.

The PDE is subject to homogeneous Dirichlet boundary conditions and the initial values are

$$\begin{aligned} u(t,0) &= 0 \\ u(t,2) &= 0 \\ u(0,x) &= x(2-x)\exp(2x). \end{aligned} \tag{15}$$

A system of MX ODEs is obtained by discretizing the x-axis with $MX + 2$ grid points and replacing the first and second order spatial derivatives with their central difference approximations. Since the value of u is constant at the two endpoints, the semi-discrete equations for those points can be eliminated. The resulting system of ODEs can now be written with u_i as the approximation to $u(t, x_i)$, $x_i = i(\Delta x)$, and $\Delta x = 2/(MX + 1)$:

$$u_i'(t) = p_1 \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} + p_2 \frac{u_{i+1} - u_{i-1}}{2(\Delta x)}. \tag{16}$$

The above equation holds for $i = 1, 2, \ldots, MX$ with the understanding that $u_0 = u_{MX+1} = 0$.

The file `spvnx.c` is included in the SensPVODE package and is the code for this problem. It uses the Adams (nonstiff) integration formula and functional iteration. As it stands, it is an unrealistically small, simple problem. The output shown below is for 10 grid points and four processors.

```
 1-D advection-diffusion equation, mesh size = 10
 Number of sensitivity vectors: Ns  =   2
 Number of PEs =   4

At t = 0.00    max.norm(u) =  1.569909e+01
sensitivity s_1:  max.norm =  0.000000e+00
sensitivity s_2:  max.norm =  0.000000e+00

At t = 0.50    max.norm(u) =  3.052880e+00   nst = 111
sensitivity s_1:  max.norm =  3.866800e+00
sensitivity s_2:  max.norm =  6.202004e-01

At t = 1.00    max.norm(u) =  8.753254e-01   nst = 179
sensitivity s_1:  max.norm =  2.174317e+00
sensitivity s_2:  max.norm =  1.890860e-01
```

```
At t = 1.50    max.norm(u) =  2.494878e-01   nst = 244
sensitivity s_1:  max.norm =  9.182392e-01
sensitivity s_2:  max.norm =  7.392083e-02

At t = 2.00    max.norm(u) =  7.109524e-02   nst = 319
sensitivity s_1:  max.norm =  3.466627e-01
sensitivity s_2:  max.norm =  2.822763e-02

At t = 2.50    max.norm(u) =  2.025933e-02   nst = 385
sensitivity s_1:  max.norm =  1.230116e-01
sensitivity s_2:  max.norm =  1.008525e-02

At t = 3.00    max.norm(u) =  5.773203e-03   nst = 453
sensitivity s_1:  max.norm =  4.195618e-02
sensitivity s_2:  max.norm =  3.455627e-03

At t = 3.50    max.norm(u) =  1.645103e-03   nst = 521
sensitivity s_1:  max.norm =  1.391418e-02
sensitivity s_2:  max.norm =  1.167859e-03

At t = 4.00    max.norm(u) =  4.694338e-04   nst = 594
sensitivity s_1:  max.norm =  4.533121e-03
sensitivity s_2:  max.norm =  3.944522e-04

At t = 4.50    max.norm(u) =  1.350271e-04   nst = 672
sensitivity s_1:  max.norm =  1.463116e-03
sensitivity s_2:  max.norm =  1.266590e-04

At t = 5.00    max.norm(u) =  3.894741e-05   nst = 753
sensitivity s_1:  max.norm =  4.668291e-04
sensitivity s_2:  max.norm =  4.070255e-05


Final Statistics..

nst = 753     nfe  = 7006    nni = 0        ncfn = 140     netf = 2
```

6.2. Example problem 2 - A stiff PDE system. This test problem is based on a two-dimensional system of two PDEs involving diurnal kinetics, advection, and diffusion. The PDEs can be written as

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y}\left(K_v(y)\frac{\partial c^i}{\partial y}\right) + R^i(c^1, c^2, t) \quad (i = 1, 2), \tag{17}$$

where the superscripts i are used to distinguish the chemical species, the reaction terms are given by

$$
\begin{aligned}
R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2 \quad \text{and} \\
R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 - q_4(t) c^2,
\end{aligned} \tag{18}
$$

and $K_v(y) = K_0 \exp(y/5)$. The spatial domain is $0 \le x \le 20,\ 30 \le y \le 50$. Some of the parameters for this problem are $K_h = 4.0 \times 10^{-6}$, $V = 10^{-3}$, $K_0 = 10^{-8}$, $q_1 = 1.63 \times 10^{-16}$, $q_2 = 4.66 \times 10^{-16}$, and $c^3 = 3.7 \times 10^{16}$. The diurnal rate constants are

$$
\begin{aligned}
q_i(t) &= \exp[-a_i / \sin \omega t] \quad \text{for } \sin \omega t > 0 \quad \text{and} \\
q_i(t) &= 0, \qquad\qquad\qquad \text{for } \sin \omega t \le 0,
\end{aligned}
$$

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

Homogeneous Neumann boundary conditions are imposed on each boundary and the initial conditions are

$$
\begin{aligned}
c^1(x, z, 0) &= 10^6 \alpha(x) \beta(y) \\
c^2(x, z, 0) &= 10^{12} \alpha(x) \beta(y) \\
\alpha(x) &= 1 - (0.1x - 1)^2 + (0.1x - 1)^4/2 \\
\beta(y) &= 1 - (0.1y - 4)^2 + (0.1y - 4)^4/2.
\end{aligned} \tag{19}
$$

The PDE system is spatially discretized on a uniform mesh using centered finite difference approximations.

These equations represent a simplified model for the transport, production, and loss of the oxygen singlet and ozone in the upper atmosphere. For the purpose of sensitivity analysis, we identify the following 8 parameters associated with this problem: $p_1 = q_1$, $p_2 = q_2$, $p_3 = c_3$, $p_4 = a_3$, $p_5 = a_4$, $p_6 = K_h$, $p_7 = V$, and $p_8 = K_v$. However, only sensitivities with respect to p_1 and p_2 are computed in the example.

The code listing for this example is given in the Appendix, while the code itself is in the file `spvkx.c` in the SensPVODE package. The purpose of this code is to provide a more complicated example than Example 1, and to provide a template for the sensitivity analysis of a stiff ODE system arising from a PDE system. The solution method is BDF with Newton iteration and SPGMR. The left preconditioner is the block-diagonal part of the Newton matrix, with 2×2 blocks, and the corresponding diagonal blocks of the Jacobian are saved each time the preconditioner is generated (for reuse later under certain conditions).

Sample output from `spvkx.c` follows. The output is for four processors (in a 2×2 array) with a 5×5 subgrid on each processor.

```
2-species diurnal advection-diffusion problem
Number of sensitivity vectors: Ns  =   2
Number of PEs =   4
```

```
t = 7.20e+03   no. steps = 953   order = 3   stepsize = 3.16e+01
At bottom left:  c1, c2 =    1.047e+04    2.527e+11
At top right:    c1, c2 =    1.119e+04    2.700e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -6.420e+19    7.118e+19
At top right:       s_1 =   -6.860e+19    7.656e+19
sensitivity wrt p_2:
At bottom left:     s_2 =   -4.385e+14   -2.441e+18
At top right:       s_2 =   -5.006e+14   -2.784e+18

t = 1.44e+04   no. steps = 1181   order = 3   stepsize = 4.51e+01
At bottom left:  c1, c2 =    6.659e+06    2.582e+11
At top right:    c1, c2 =    7.301e+06    2.833e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -4.085e+22    5.955e+22
At top right:       s_1 =   -4.478e+22    6.717e+22
sensitivity wrt p_2:
At bottom left:     s_2 =   -4.523e+17   -6.542e+21
At top right:       s_2 =   -5.432e+17   -7.831e+21

t = 2.16e+04   no. steps = 1299   order = 4   stepsize = 1.92e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -1.635e+23    3.820e+23
At top right:       s_1 =   -1.798e+23    4.499e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -7.660e+18   -7.646e+22
At top right:       s_2 =   -9.443e+18   -9.450e+22

t = 2.88e+04   no. steps = 1413   order = 2   stepsize = 4.07e+01
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -5.338e+22    5.449e+23
At top right:       s_1 =   -5.919e+22    6.743e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -4.886e+18   -1.719e+23
At top right:       s_2 =   -6.104e+18   -2.152e+23
```

```
t = 3.60e+04   no. steps = 1521   order = 4   stepsize = 5.17e+01
At bottom left:  c1, c2 =    1.404e+04    3.387e+11
At top right:    c1, c2 =    1.561e+04    3.765e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -8.614e+19    5.272e+23
At top right:       s_1 =   -9.576e+19    6.603e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -8.433e+15   -1.844e+23
At top right:       s_2 =   -1.055e+16   -2.310e+23

t = 4.32e+04   no. steps = 1724   order = 4   stepsize = 1.80e+02
At bottom left:  c1, c2 =   -4.972e-11    3.382e+11
At top right:    c1, c2 =   -4.926e-11    3.804e+11

sensitivity wrt p_1:
At bottom left:     s_1 =    6.206e+07    5.275e+23
At top right:       s_1 =    7.013e+07    6.745e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -3.067e+05   -1.845e+23
At top right:       s_2 =   -2.300e+05   -2.360e+23

t = 5.04e+04   no. steps = 1752   order = 5   stepsize = 3.96e+02
At bottom left:  c1, c2 =    3.435e-07    3.358e+11
At top right:    c1, c2 =    3.394e-07    3.864e+11

sensitivity wrt p_1:
At bottom left:     s_1 =    1.526e+11    5.207e+23
At top right:       s_1 =    1.510e+11    6.967e+23
sensitivity wrt p_2:
At bottom left:     s_2 =    3.498e+06   -1.821e+23
At top right:       s_2 =    4.094e+06   -2.437e+23

t = 5.76e+04   no. steps = 1771   order = 5   stepsize = 2.12e+02
At bottom left:  c1, c2 =    8.096e-08    3.320e+11
At top right:    c1, c2 =    7.736e-08    3.909e+11

sensitivity wrt p_1:
At bottom left:     s_1 =    6.389e+10    5.083e+23
At top right:       s_1 =    6.114e+10    7.121e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -2.000e+07   -1.778e+23
```

```
At top right:       s_2 =   -2.291e+07   -2.491e+23

t = 6.48e+04   no. steps = 1949   order = 4   stepsize = 2.70e+02
At bottom left:  c1, c2 =   -5.432e-11    3.313e+11
At top right:    c1, c2 =   -4.728e-11    3.963e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -5.679e+08    5.044e+23
At top right:       s_1 =   -4.970e+08    7.328e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -6.058e+04   -1.765e+23
At top right:       s_2 =   -6.359e+04   -2.563e+23

t = 7.20e+04   no. steps = 1967   order = 4   stepsize = 4.38e+02
At bottom left:  c1, c2 =    3.648e-12    3.330e+11
At top right:    c1, c2 =    3.175e-12    4.039e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -4.644e+06    5.078e+23
At top right:       s_1 =   -4.028e+06    7.638e+23
sensitivity wrt p_2:
At bottom left:     s_2 =    6.910e+01   -1.777e+23
At top right:       s_2 =    7.253e+01   -2.672e+23

t = 7.92e+04   no. steps = 1983   order = 4   stepsize = 4.38e+02
At bottom left:  c1, c2 =    9.015e-19    3.334e+11
At top right:    c1, c2 =   -2.221e-18    4.120e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -1.639e+01    5.073e+23
At top right:       s_1 =   -1.477e+01    7.996e+23
sensitivity wrt p_2:
At bottom left:     s_2 =   -7.758e-05   -1.775e+23
At top right:       s_2 =   -8.703e-05   -2.797e+23

t = 8.64e+04   no. steps = 1995   order = 5   stepsize = 6.72e+02
At bottom left:  c1, c2 =    1.207e-19    3.352e+11
At top right:    c1, c2 =    2.268e-20    4.163e+11

sensitivity wrt p_1:
At bottom left:     s_1 =   -8.837e-01    5.117e+23
At top right:       s_1 =   -6.380e-01    8.214e+23
sensitivity wrt p_2:
```

```
At bottom left:     s_2 =   -2.156e-06   -1.790e+23
At top right:       s_2 =   -8.659e-07   -2.874e+23


Final Statistics..

lenrw    =  6000    leniw =     0
llrw     =  2046    lliw  =     0
nst      =  1995    nfe   = 20377
nni      =  2693    nli   =  6901
nsetups =   266    netf  =    81
npe      =    38    nps   = 14866
ncfn     =     2    ncfl  =     0
```

A third example is provided with the SensPVODE package, in the file `spvkxb.c`. It uses the same ODE system as in the above stiff example, but a slightly different solution method. It uses the `PVBBDPRE` preconditioner module to generate a band-block-diagonal preconditioner, using half-bandwidths equal to 2.

7. Availability. The SensPVODE package is being released for general distribution at this time. Interested users should contact Alan Hindmarsh (alanh@llnl.gov) or Steven Lee (slee@llnl.gov).

REFERENCES

[1] P. N. Brown and A. C. Hindmarsh, *Reduced Storage Matrix Methods in Stiff ODE Systems*, J. Appl. Math. & Comp., **31** (1989), pp. 40-91.

[2] Y. Saad and M. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., **11** (1990), 856-869.

[3] George D. Byrne and Alan C. Hindmarsh, *User Documentation for PVODE, an ODE Solver for Parallel Computers*, Lawrence Livermore National Laboratory report UCRL-ID-130884, May 1998. See also the Addenda in the doc subdirectory of the current version of *PVODE*.

[4] Scott D. Cohen and Alan C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, Sept. 1994.

[5] Scott D. Cohen and Alan C. Hindmarsh, *CVODE, a Stiff/Nonstiff ODE Solver in C*, Computers in Physics, **10**, No. 2 (1996), pp. 138-143.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.

[7] T. Maly and L. R. Petzold, *Numerical Methods and Software for Sensitivity Analysis of Differential-algebraic systems*, Appl. Numer. Math., **20** (1996), pp. 57-79.

8. Appendix: Listing of Stiff Example Program, with Sensitivity Analysis.

```
/*********************************************************************
 *                                                                   *
 * File:       spvkx.c                                               *
 * Version of: 25 August 2000                                        *
 *                                                                   *
 *-------------------------------------------------------------------*
 * Example problem, with sensitivity analysis                        *
 * An ODE system is generated from the following 2-species diurnal   *
 * kinetics advection-diffusion PDE system in 2 space dimensions:    *
 *                                                                   *
 * dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)  *
 *                 + Ri(c1,c2,t)      for i = 1,2,   where           *
 *   R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,    *
 *   R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,                 *
 *   Kv(y) = Kv0*exp(y/5) ,                                          *
 * Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)     *
 * vary diurnally.   The problem is posed on the square              *
 *   0 <= x <= 20,    30 <= y <= 50   (all in km),                   *
 * with homogeneous Neumann boundary conditions, and for time t in   *
 *   0 <= t <= 86400 sec (1 day).                                    *
 * The PDE system is treated by central differences on a uniform     *
 * mesh, with simple polynomial initial profiles.                    *
 *                                                                   *
 * The problem is solved by SensPVODE on NPE processors, treated as a  *
 * rectangular process grid of size NPEX by NPEY, with NPE = NPEX*NPEY. *
 * Each processor contains a subgrid of size MXSUB by MYSUB of the   *
 * (x,y) mesh.  Thus the actual mesh sizes are MX = MXSUB*NPEX and   *
 * MY = MYSUB*NPEY, and the ODE system size is Ny = 2*MX*MY.         *
 *                                                                   *
 * The solution with SensPVODE is done with the BDF/GMRES method (i.e.  *
 * using the SensCVSPGMR linear solver) and the block-diagonal part of  *
 * the Newton matrix as a left preconditioner.                       *
 * A copy of the block-diagonal part of the Jacobian is saved and    *
 * conditionally reused within the Precond routine.                  *
 *                                                                   *
 * Performance data and sampled solution and sensitivity values are  *
 * printed at selected output times.                                 *
 * All performance counters are printed on completion.               *
 *                                                                   *
 * This version uses MPI for user routines, and the SensPVODE solver.  *
 * Execution: spvkx -npes N  with N = NPEX*NPEY (see constants below).  *
```

```
 ***********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "llnltyps.h"  /* definitions of real, integer, boole, TRUE,FALSE */
#include "cvode.h"     /* main CVODE header file                          */
#include "iterativ.h"  /* contains the enum for types of preconditioning  */
#include "cvspgmr.h"   /* use CVSPGMR linear solver each internal step    */
#include "smalldense.h" /* use generic DENSE solver in preconditioning    */
#include "nvector.h"   /* definitions of type N_Vector, macro N_VDATA     */
#include "llnlmath.h"  /* contains SQR macro                              */
#include "mpi.h"       /* MPI data types and prototypes                   */
#include "sensitivity.h" /* sensitivity data types and prototypes         */


/* Problem Constants */

#define NVARS        2             /* number of species         */

#define C1_SCALE     1.0e6         /* coefficients in initial profiles    */
#define C2_SCALE     1.0e12

#define T0           0.0           /* initial time */
#define NOUT         12            /* number of output times */
#define TWOHR        7200.0        /* number of seconds in two hours  */
#define HALFDAY      4.32e4        /* number of seconds in a half day */
#define PI       3.1415926535898   /* pi */

#define XMIN          0.0          /* grid boundaries in x  */
#define XMAX         20.0
#define YMIN         30.0          /* grid boundaries in y  */
#define YMAX         50.0

#define NPEX         2              /* no. PEs in x direction of PE array */
#define NPEY         2              /* no. PEs in y direction of PE array */
                                    /* Total no. PEs = NPEX*NPEY */
#define MXSUB        5              /* no. x points per subgrid */
#define MYSUB        5              /* no. y points per subgrid */

#define MX           (NPEX*MXSUB)   /* MX = number of x mesh points */
#define MY           (NPEY*MYSUB)   /* MY = number of y mesh points */
                                    /* Spatial mesh is MX by MY */
```

```
#define Ny          NVARS*MX*MY    /* number of equations          */
#define Np          8              /* number of parameters         */
#define Ns          2              /* number of sensitivities      */

#define ZERO        RCONST(0.0)    /* real 0.0 */
#define ONE         RCONST(1.0)    /* real 1.0 */
#define TWO         RCONST(2.0)    /* real 2.0 */

/* CVodeMalloc Constants */

#define RTOL    1.0e-5           /* scalar relative tolerance */
#define FLOOR   100.0            /* value of C1 or C2 at which tolerances */
                                 /* change from relative to absolute      */
#define ATOL    (RTOL*FLOOR)     /* scalar absolute tolerance */


/* User-defined matrix accessor macro: IJth */

/* IJth is defined in order to write code which indexes into small dense
   matrices with a (row,column) pair, where 1 <= row,column <= NVARS.

   IJth(a,i,j) references the (i,j)th entry of the small matrix real **a,
   where 1 <= i,j <= NVARS. The small matrix routines in dense.h
   work with matrices stored by column in a 2-dimensional array. In C,
   arrays are indexed starting at 0, not 1. */

#define IJth(a,i,j)        (a[j-1][i-1])


/* Type : UserData
   contains problem constants, preconditioner blocks, pivot arrays,
   grid constants, and processor indices */

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

typedef struct {
```

```
  void *f_data;
  real **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
  integer *pivot[MXSUB][MYSUB];
} *PreconData;


/* Private Helper Functions */

static PreconData AllocPreconData(UserData data);
static void InitUserData(integer my_pe, MPI_Comm comm, UserData data);
static void FreePreconData(PreconData pdata);
static void SetInitialProfiles(N_Vector u, UserData data);
static void PrintOutput(integer my_pe, MPI_Comm comm, long int iopt[],
                        real ropt[], N_Vector u, real t);
static void SensPrintOutput(integer my_pe, MPI_Comm comm, long int iopt[],
                        real ropt[], N_Vector w, real *pbar, integer i, real t);
static void PrintFinalStats(long int iopt[]);
static void BSend(MPI_Comm comm, integer my_pe, integer isubx, integer isuby,
                  integer dsizex, integer dsizey, real udata[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[], integer my_pe,
      integer isubx, integer isuby,
      integer dsizex, integer dsizey,
      real uext[], real buffer[]);
static void BRecvWait(MPI_Request request[], integer isubx, integer isuby,
      integer dsizex, real uext[], real buffer[]);
static void ucomm(integer N, real t, N_Vector u, UserData data);
static void fcalc(integer N, real t, real udata[], real dudata[], UserData data);

/* Functions Called by the CVODE Solver */

static void f(integer N, real t, N_Vector u, N_Vector udot, void *f_data);

static int Precond(integer N, real tn, N_Vector u, N_Vector fu, boole jok,
                   boole *jcurPtr, real gamma, N_Vector ewt, real h,
                   real uround, long int *nfePtr, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);

static int PSolve(integer N, real tn, N_Vector u, N_Vector fu, N_Vector vtemp,
                  real gamma, N_Vector ewt, real delta, long int *nfePtr,
                  N_Vector r, int lr, void *P_data, N_Vector z);


/****************************** Main Program *******************************/
```

```
main(int argc, char *argv[])
{
  real abstol, reltol, t, tout, ropt[OPT_SIZE];
  long int iopt[OPT_SIZE];
  N_Vector u;
  UserData data;
  PreconData predata;
  void *cvode_mem;
  int iout, flag, i;
  integer local_N, my_pe, npes;
  machEnvType machEnv;
  MPI_Comm comm;
  real Q1, Q2, C3, A3, A4, KH, VEL, KV0;
  N_Vector *usub;
  integer Ntotal;
  real *pbar, rhomax;

  /* Set problem size */
  Ntotal = (1+Ns)*Ny;

  /* Set problem parameters */
  Q1 = 1.63e-16;
  Q2 = 4.66e-16;
  C3 = 3.7e16;
  A3 = 22.62;
  A4 = 7.601;
  KH = 4.0e-6;
  VEL = 0.001;
  KV0 = 1.0e-8;

  /* Get processor number and total number of pe's */
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

  if (npes != NPEX*NPEY) {
    if (my_pe == 0)
      printf("\n npes=%d is not equal to NPEX*NPEY=%d\n", npes,NPEX*NPEY);
    return(1);
  }
```

```
/* Set local length */
local_N = NVARS*MXSUB*MYSUB;

/* Set machEnv block */
machEnv = PVecInitMPI(comm, local_N, Ny, &argc, &argv);
if (machEnv == NULL) return(1);

/* Allocate and load user data block */
data = (UserData) malloc(sizeof *data);
data->p = (real *) malloc(Np*sizeof(real));
data->p[0] = Q1;
data->p[1] = Q2;
data->p[2] = C3;
data->p[3] = A3;
data->p[4] = A4;
data->p[5] = KH;
data->p[6] = VEL;
data->p[7] = KV0;
InitUserData(my_pe, comm, data);

/* Scaling factor for each sensitivity equation */
pbar = (real *) malloc(Np*sizeof(real));
for (i = 1; i <= Ns; i++)
  pbar[i-1] = data->p[i-1];

/* Allocate preconditioner block */
predata = AllocPreconData (data);

/* Allocate u, and set initial values and tolerances */
u = N_VNew(Ntotal, machEnv);
SetInitialProfiles(u, data);
abstol = ATOL; reltol = RTOL;

/* Set initial values for sensitivity variables */
usub = N_VSUB(u);
for (i = 1; i <= Ns; i++)
  N_VConst(ZERO, usub[i]);

/* Set (optional) inputs in iopt and ropt arrays */
for (i = 0; i < OPT_SIZE; i++) {
  iopt[i] = 0;
  ropt[i] = 0.0;
}
```

```
  iopt[MXSTEP] = 1000;

  /* rhomax selects the finite difference formula for estimating */
  /* scaled sensitivity vectors.                                 */
  /* rhomax = 0.0 is the default value.                          */
  rhomax = ZERO;

/* Call SensCVodeMalloc to initialize CVODE:

     Ny      is the number of ODEs in u' = f(t,u,p)
     Ns      is the number of sensitivity vectors to compute
     Ntotal  is the problem size = total number of ODEs = Ny*(Ns+1)
     f       is the user's right hand side function in u' = f(t,u,p)
     T0      is the initial time
     u       is the initial dependent variable vector of length Ntotal
     BDF     specifies the Backward Differentiation Formula
     NEWTON  specifies a Newton iteration
     SS      specifies scalar relative and absolute tolerances
     &reltol and &abstol are pointers to the scalar tolerances
     data    is the pointer to the user-defined block of coefficients
     TRUE    indicates there are optional inputs in iopt and ropt
     iopt    and ropt arrays communicate optional integer and real input/output
     data->p is a pointer to the parameter values
     pbar    is a pointer to the sensitivity scaling factors
     rhomax  selects the formula for estimating scaled sensitivity derivatives

     A pointer to CVODE problem memory is returned and stored in cvode_mem.  */

  cvode_mem = SensCVodeMalloc(Ny, Ns, Ntotal, f, T0, u, BDF, NEWTON, SS,
      &reltol, &abstol, data, NULL, TRUE, iopt,
      ropt, machEnv, data->p, pbar, rhomax);

  if (cvode_mem == NULL) { printf("SensCVodeMalloc failed."); return(1); }

  /* Call SensCVSpgmr to specify the CVODE linear solver CVSPGMR with
     left preconditioning, modified Gram-Schmidt orthogonalization,
     default values for the maximum Krylov dimension maxl and the tolerance
     parameter delt, preconditioner setup and solve routines Precond and
     PSolve, and the pointer to the preconditioner data block.            */

  SensCVSpgmr(cvode_mem, LEFT, MODIFIED_GS, 0, 0.0, Precond, PSolve, predata);

  if (my_pe == 0) {
```

```
    printf("\n2-species diurnal advection-diffusion problem\n");
    printf("Number of sensitivity vectors: Ns  = %3d \n", Ns);
    printf("Number of PEs = %3d \n\n",npes);
  }

  /* In loop over output points, call CVode, print results, test for error */

  for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CVode(cvode_mem, tout, u, &t, NORMAL);
    PrintOutput(my_pe, comm, iopt, ropt, u, t);
    for (i = 1; i <= Ns; i++) {
      if (my_pe == 0) {
printf("sensitivity wrt p_%d:\n", i);
      }
      SensPrintOutput(my_pe, comm, iopt, ropt, usub[i], pbar, i, t);
      if ((my_pe == 0) && (i == Ns)) printf("\n");
    }
    if (flag != SUCCESS) {
      if (my_pe == 0) printf("CVode failed, flag=%d.\n", flag);
      break;
    }
  }

  /* Free memory and print final statistics */
  if (my_pe == 0) PrintFinalStats(iopt);

  SensCVodeFree(cvode_mem);

  N_VFree(u);
  free(pbar);
  free(data->p);
  free(data);

  FreePreconData(predata);
  PVecFreeMPI(machEnv);
  MPI_Finalize();
  return(0);
}


/*********************** Private Helper Functions ************************/

/* Allocate memory for data structure of type UserData */
```

```
static PreconData AllocPreconData(UserData fdata)
{
  int lx, ly;
  PreconData pdata;

  pdata = (PreconData) malloc(sizeof *pdata);

  pdata->f_data = fdata;

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      (pdata->P)[lx][ly] = denalloc(NVARS);
      (pdata->Jbd)[lx][ly] = denalloc(NVARS);
      (pdata->pivot)[lx][ly] = denallocpiv(NVARS);
    }
  }

  return(pdata);
}

/* Load constants in data */

static void InitUserData(integer my_pe, MPI_Comm comm, UserData data)
{
  integer isubx, isuby;
  real KH, VEL, KV0;

  /* Load problem coefficients and parameters */
  KH = data->p[5];
  VEL = data->p[6];
  KV0 = data->p[7];

  /* Set problem constants */
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/((real)(MX-1));
  data->dy = (YMAX-YMIN)/((real)(MY-1));

  /* Set machine-related constants */
  data->comm = comm;
  data->my_pe = my_pe;
  /* isubx and isuby are the PE grid indices corresponding to my_pe */
  isuby = my_pe/NPEX;
```

```
  isubx = my_pe - isuby*NPEX;
  data->isubx = isubx;
  data->isuby = isuby;
  /* Set the sizes of a boundary x-line in u and uext */
  data->nvmxsub = NVARS*MXSUB;
  data->nvmxsub2 = NVARS*(MXSUB+2);

}

/* Free data memory */

static void FreePreconData(PreconData pdata)
{
  int lx, ly;

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      denfree((pdata->P)[lx][ly]);
      denfree((pdata->Jbd)[lx][ly]);
      denfreepiv((pdata->pivot)[lx][ly]);
    }
  }

  free(pdata);
}

/* Set initial conditions in u */

static void SetInitialProfiles(N_Vector u, UserData data)
{
  integer isubx, isuby, lx, ly, jx, jy, offset;
  real dx, dy, x, y, cx, cy, xmid, ymid;
  real *udata;

  /* Set pointer to data array in vector u */

  udata = N_VDATA(u);

  /* Get mesh spacings, and subgrid indices for this PE */

  dx = data->dx;          dy = data->dy;
  isubx = data->isubx;    isuby = data->isuby;
```

```
  /* Load initial profiles of c1 and c2 into local u vector.
  Here lx and ly are local mesh point indices on the local subgrid,
  and jx and jy are the global mesh point indices. */

  offset = 0;
  xmid = .5*(XMIN + XMAX);
  ymid = .5*(YMIN + YMAX);
  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    y = YMIN + jy*dy;
    cy = SQR(0.1*(y - ymid));
    cy = 1.0 - cy + 0.5*SQR(cy);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      x = XMIN + jx*dx;
      cx = SQR(0.1*(x - xmid));
      cx = 1.0 - cx + 0.5*SQR(cx);
      udata[offset  ] = C1_SCALE*cx*cy;
      udata[offset+1] = C2_SCALE*cx*cy;
      offset = offset + 2;
    }
  }
}

/* Print current t, step count, order, stepsize, and sampled c1,c2 values */

static void PrintOutput(integer my_pe, MPI_Comm comm, long int iopt[],
                        real ropt[], N_Vector u, real t)
{
  real *udata, tempu[2];
  integer npelast, i0, i1;
  MPI_Status status;

  npelast = NPEX*NPEY - 1;
  udata = N_VDATA(u);

  /* Send c1,c2 at top right mesh point to PE 0 */
  if (my_pe == npelast) {
    i0 = NVARS*MXSUB*MYSUB - 2;
    i1 = i0 + 1;
    if (npelast != 0)
      MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
```

```
      tempu[0] = udata[i0];
      tempu[1] = udata[i1];
    }
  }

  /* On PE 0, receive c1,c2 at top right, then print performance data
     and sampled solution values */
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    printf("t = %.2e   no. steps = %d   order = %d   stepsize = %.2e\n",
           t, iopt[NST], iopt[QU], ropt[HU]);
    printf("At bottom left:  c1, c2 = %12.3e %12.3e \n", udata[0], udata[1]);
    printf("At top right:    c1, c2 = %12.3e %12.3e \n\n", tempu[0], tempu[1]);
  }
}

static void SensPrintOutput(integer my_pe, MPI_Comm comm, long int iopt[],
                            real ropt[], N_Vector w, real *pbar, integer i,
    real t)
{
  real *wdata, tempw[2];
  integer npelast, i0, i1;
  MPI_Status status;

  npelast = NPEX*NPEY - 1;
  wdata = N_VDATA(w);

  /* w is one of the NS sensitivity vectors */
  /* Send w at top right mesh point to PE 0 */
  if (my_pe == npelast) {
    i0 = NVARS*MXSUB*MYSUB - 2;
    i1 = i0 + 1;
    if (npelast != 0)
      MPI_Send(&wdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
      tempw[0] = wdata[i0];
      tempw[1] = wdata[i1];
    }
  }

  /* On PE 0, receive w at top right, then print performance data
     and sampled solution values */
```

```
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&tempw[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    printf("At bottom left:     s_%d = %12.3e %12.3e \n",
   i,wdata[0]/pbar[i-1],wdata[1]/pbar[i-1]);
    printf("At top right:       s_%d = %12.3e %12.3e \n",
   i,tempw[0]/pbar[i-1],tempw[1]/pbar[i-1]);
  }
}

/* Print final statistics contained in iopt */

static void PrintFinalStats(long int iopt[])
{
  printf("\nFinal Statistics.. \n\n");
  printf("lenrw   = %5ld    leniw = %5ld\n", iopt[LENRW], iopt[LENIW]);
  printf("llrw    = %5ld    lliw  = %5ld\n", iopt[SPGMR_LRW], iopt[SPGMR_LIW]);
  printf("nst     = %5ld    nfe   = %5ld\n", iopt[NST], iopt[NFE]);
  printf("nni     = %5ld    nli   = %5ld\n", iopt[NNI], iopt[SPGMR_NLI]);
  printf("nsetups = %5ld    netf  = %5ld\n", iopt[NSETUPS], iopt[NETF]);
  printf("npe     = %5ld    nps   = %5ld\n", iopt[SPGMR_NPE], iopt[SPGMR_NPS]);
  printf("ncfn    = %5ld    ncfl  = %5ld\n \n", iopt[NCFN], iopt[SPGMR_NCFL]);
}

/* Routine to send boundary data to neighboring PEs */

static void BSend(MPI_Comm comm, integer my_pe, integer isubx, integer isuby,
                  integer dsizex, integer dsizey, real udata[])
{
  int i, ly;
  integer offsetu, offsetbuf;
  real bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];

  /* If isuby > 0, send data from bottom x-line of u */

  if (isuby != 0)
    MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

  /* If isuby < NPEY-1, send data from top x-line of u */

  if (isuby != NPEY-1) {
    offsetu = (MYSUB-1)*dsizex;
    MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
```

```
  }

  /* If isubx > 0, send data from left y-line of u (via bufleft) */

  if (isubx != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = ly*dsizex;
      for (i = 0; i < NVARS; i++)
        bufleft[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
  }

  /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */

  if (isubx != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
      for (i = 0; i < NVARS; i++)
        bufright[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
  }

}

/* Routine to start receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB real entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvPost(MPI_Comm comm, MPI_Request request[], integer my_pe,
      integer isubx, integer isuby,
      integer dsizex, integer dsizey,
      real uext[], real buffer[])
{
  integer offsetue;
  /* Have bufleft and bufright use the same buffer */
  real *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
```

```
  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE,
      my_pe-NPEX, 0, comm, &request[0]);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1) {
    offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
    MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
                                      my_pe+NPEX, 0, comm, &request[1]);
  }

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
                                      my_pe-1, 0, comm, &request[2]);
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
                                      my_pe+1, 0, comm, &request[3]);
  }

}

/* Routine to finish receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB real entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvWait(MPI_Request request[], integer isubx, integer isuby,
      integer dsizex, real uext[], real buffer[])
{
  int i, ly;
  integer dsizex2, offsetue, offsetbuf;
  real *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NVARS;
```

```
  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Wait(&request[0],&status);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1)
    MPI_Wait(&request[1],&status);

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Wait(&request[2],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+1)*dsizex2;
      for (i = 0; i < NVARS; i++)
        uext[offsetue+i] = bufleft[offsetbuf+i];
    }
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Wait(&request[3],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+2)*dsizex2 - NVARS;
      for (i = 0; i < NVARS; i++)
uext[offsetue+i] = bufright[offsetbuf+i];
    }
  }

}

/* ucomm routine.  This routine performs all communication
   between processors of data needed to calculate f. */

static void ucomm(integer N, real t, N_Vector u, UserData data)
{
```

```
  real *udata, *uext, buffer[2*NVARS*MYSUB];
  MPI_Comm comm;
  integer my_pe, isubx, isuby, nvmxsub, nvmysub;
  MPI_Request request[4];

  udata = N_VDATA(u);


  /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */

  comm = data->comm;  my_pe = data->my_pe;
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;
  nvmysub = NVARS*MYSUB;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs */

  BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs */

  BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, udata);

  /* Finish receiving boundary data from neighboring PEs */

  BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);



}

/* fcalc routine. Compute f(t,y).  This routine assumes that communication
   between processors of data needed to calculate f has already been done,
   and this data is in the work array uext. */

static void fcalc(integer N, real t, real udata[], real dudata[], UserData data)
{
  real *uext;
  real q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  real c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
  real qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
  real q4coef, dely, verdco, hordco, horaco;
```

```
int i, lx, ly, jx, jy;
integer isubx, isuby, nvmxsub, nvmxsub2, offsetu, offsetue;
real Q1, Q2, C3, A3, A4, KH, VEL, KV0;

/* Load problem coefficients and parameters */
Q1 = data->p[0];
Q2 = data->p[1];
C3 = data->p[2];
A3 = data->p[3];
A4 = data->p[4];
KH = data->p[5];
VEL = data->p[6];
KV0 = data->p[7];

/* Get subgrid indices, data sizes, extended work array uext */
isubx = data->isubx;   isuby = data->isuby;
nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
uext = data->uext;

/* Copy local segment of u vector into the working extended array uext */

offsetu = 0;
offsetue = nvmxsub2 + NVARS;
for (ly = 0; ly < MYSUB; ly++) {
  for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
  offsetu = offsetu + nvmxsub;
  offsetue = offsetue + nvmxsub2;
}

/* To facilitate homogeneous Neumann boundary conditions, when this is
a boundary PE, copy data from the first interior mesh line of u to uext */

/* If isuby = 0, copy x-line 2 of u to uext */
if (isuby == 0) {
  for (i = 0; i < nvmxsub; i++) uext[NVARS+i] = udata[nvmxsub+i];
}

/* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
if (isuby == NPEY-1) {
  offsetu = (MYSUB-2)*nvmxsub;
  offsetue = (MYSUB+1)*nvmxsub2 + NVARS;
  for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
}
```

```
/* If isubx = 0, copy y-line 2 of u to uext */
if (isubx == 0) {
  for (ly = 0; ly < MYSUB; ly++) {
    offsetu = ly*nvmxsub + NVARS;
    offsetue = (ly+1)*nvmxsub2;
    for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
  }
}

/* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
if (isubx == NPEX-1) {
  for (ly = 0; ly < MYSUB; ly++) {
    offsetu = (ly+1)*nvmxsub - 2*NVARS;
    offsetue = (ly+2)*nvmxsub2 - NVARS;
    for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
  }
}

/* Make local copies of problem variables, for efficiency */

dely = data->dy;
verdco = (1.0/SQR(data->dy))*KV0;
hordco = KH/SQR(data->dx);
horaco = VEL/(2.0*data->dx);

/* Set diurnal rate coefficients as functions of t, and save q4 in
data block for use by preconditioner evaluation routine */

s = sin((data->om)*t);
if (s > 0.0) {
  q3 = exp(-A3/s);
  q4coef = exp(-A4/s);
} else {
  q3 = 0.0;
  q4coef = 0.0;
}
data->q4 = q4coef;


/* Loop over all grid points in local subgrid */

for (ly = 0; ly < MYSUB; ly++) {
```

```
jy = ly + isuby*MYSUB;

/* Set vertical diffusion coefficients at jy +- 1/2 */

ydn = YMIN + (jy - .5)*dely;
yup = ydn + dely;
cydn = verdco*exp(0.2*ydn);
cyup = verdco*exp(0.2*yup);
for (lx = 0; lx < MXSUB; lx++) {

  jx = lx + isubx*MXSUB;

  /* Extract c1 and c2, and set kinetic rate terms */

  offsetue = (lx+1)*NVARS + (ly+1)*nvmxsub2;
  c1 = uext[offsetue];
  c2 = uext[offsetue+1];
  qq1 = Q1*c1*C3;
  qq2 = Q2*c1*c2;
  qq3 = q3*C3;
  qq4 = q4coef*c2;
  rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
  rkin2 = qq1 - qq2 - qq4;

  /* Set vertical diffusion terms */

  c1dn = uext[offsetue-nvmxsub2];
  c2dn = uext[offsetue-nvmxsub2+1];
  c1up = uext[offsetue+nvmxsub2];
  c2up = uext[offsetue+nvmxsub2+1];
  vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
  vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);

  /* Set horizontal diffusion and advection terms */

  c1lt = uext[offsetue-2];
  c2lt = uext[offsetue-1];
  c1rt = uext[offsetue+2];
  c2rt = uext[offsetue+3];
  hord1 = hordco*(c1rt - 2.0*c1 + c1lt);
  hord2 = hordco*(c2rt - 2.0*c2 + c2lt);
  horad1 = horaco*(c1rt - c1lt);
```

```
      horad2 = horaco*(c2rt - c2lt);

      /* Load all terms into dudata */

      offsetu = lx*NVARS + ly*nvmxsub;
      dudata[offsetu]   = vertd1 + hord1 + horad1 + rkin1;
      dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
    }
  }

}


/***************** Functions Called by the CVODE Solver ******************/

/* f routine.  Evaluate f(t,y).  First call ucomm to do communication of
   subgrid boundary data into uext.  Then calculate f by a call to fcalc. */

static void f(integer N, real t, N_Vector u, N_Vector udot, void *f_data)
{
  real *udata, *dudata;
  UserData data;

  udata = N_VDATA(u);
  dudata = N_VDATA(udot);
  data = (UserData) f_data;


  /* Call ucomm to do inter-processor communicaiton */

  ucomm (N, t, u, data);

  /* Call fcalc to calculate all right-hand sides */

  fcalc (N, t, udata, dudata, data);

}


/* Preconditioner setup routine. Generate and preprocess P. */

static int Precond(integer N, real tn, N_Vector u, N_Vector fu, boole jok,
                   boole *jcurPtr, real gamma, N_Vector ewt, real h,
```

```
                   real uround, long int *nfePtr, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
  real c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
  real **(*P)[MYSUB], **(*Jbd)[MYSUB];
  integer nvmxsub, *(*pivot)[MYSUB], ier, offset;
  int lx, ly, jx, jy, isubx, isuby;
  real *udata, **a, **j;
  PreconData predata;
  UserData data;
  real Q1, Q2, C3, A3, A4, KH, VEL, KV0;

  /* Make local copies of pointers in P_data, pointer to u's data,
     and PE index pair */

  predata = (PreconData) P_data;
  data = (UserData) (predata->f_data);
  P = predata->P;
  Jbd = predata->Jbd;
  pivot = predata->pivot;
  udata = N_VDATA(u);
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;

  /* Load problem coefficients and parameters */
  Q1 = data->p[0];
  Q2 = data->p[1];
  C3 = data->p[2];
  A3 = data->p[3];
  A4 = data->p[4];
  KH = data->p[5];
  VEL = data->p[6];
  KV0 = data->p[7];

  if (jok) {

  /* jok = TRUE: Copy Jbd to P */

    for (ly = 0; ly < MYSUB; ly++)
      for (lx = 0; lx < MXSUB; lx++)
        dencopy(Jbd[lx][ly], P[lx][ly], NVARS);

  *jcurPtr = FALSE;
```

```
}

else {

/* jok = FALSE: Generate Jbd from scratch and copy to P */

/* Make local copies of problem variables, for efficiency */

q4coef = data->q4;
dely = data->dy;
verdco = (1.0/SQR(data->dy))*KV0;
hordco = KH/SQR(data->dx);

/* Compute 2x2 diagonal Jacobian blocks (using q4 values
   computed on the last f call).  Load into P. */

  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    ydn = YMIN + (jy - .5)*dely;
    yup = ydn + dely;
    cydn = verdco*exp(0.2*ydn);
    cyup = verdco*exp(0.2*yup);
    diag = -(cydn + cyup + 2.0*hordco);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      offset = lx*NVARS + ly*nvmxsub;
      c1 = udata[offset];
      c2 = udata[offset+1];
      j = Jbd[lx][ly];
      a = P[lx][ly];
      IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
      IJth(j,1,2) = -Q2*c1 + q4coef;
      IJth(j,2,1) = Q1*C3 - Q2*c2;
      IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
      dencopy(j, a, NVARS);
    }
  }

*jcurPtr = TRUE;

}
```

```
  /* Scale by -gamma */

    for (ly = 0; ly < MYSUB; ly++)
      for (lx = 0; lx < MXSUB; lx++)
        denscale(-gamma, P[lx][ly], NVARS);

  /* Add identity matrix and do LU decompositions on blocks in place */

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      denaddI(P[lx][ly], NVARS);
      ier = gefa(P[lx][ly], NVARS, pivot[lx][ly]);
      if (ier != 0) return(1);
    }
  }

  return(0);
}


/* Preconditioner solve routine */

static int PSolve(integer N, real tn, N_Vector u, N_Vector fu, N_Vector vtemp,
                  real gamma, N_Vector ewt, real delta, long int *nfePtr,
                  N_Vector r, int lr, void *P_data, N_Vector z)
{
  real **(*P)[MYSUB];
  integer nvmxsub, *(*pivot)[MYSUB];
  int lx, ly;
  real *zdata, *v;
  PreconData predata;
  UserData data;

  /* Extract the P and pivot arrays from P_data */

  predata = (PreconData) P_data;
  data = (UserData) (predata->f_data);
  P = predata->P;
  pivot = predata->pivot;

  /* Solve the block-diagonal system Px = r using LU factors stored
     in P and pivot data in pivot, and return the solution in z.
     First copy vector r to z. */
```

```
  N_VScale(1.0, r, z);

  nvmxsub = data->nvmxsub;
  zdata = N_VDATA(z);

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      v = &(zdata[lx*NVARS + ly*nvmxsub]);
      gesl(P[lx][ly], NVARS, pivot[lx][ly], v);
    }
  }

  return(0);
}
```