# Example Programs for CVODES v2.2.0

Alan C. Hindmarsh and Radu Serban
*Center for Applied Scientific Computing*
*Lawrence Livermore National Laboratory*

April 2005

UCRL-SM-208115

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

# Contents

# 1 Introduction

This report is intended to serve as a companion document to the User Documentation of CVODES [2]. It provides details, with listings, on the example programs supplied with the CVODES distribution package.

The CVODE distribution contains examples of the following types: serial and parallel examples for IVP integration, serial and parallel examples for forward sensitivity analysis, and serial and parallel examples for adjoint sensitivity analysis. These examples, summarized below, are shortly described next.

| | Serial examples | Parallel examples |
|---|---|---|
| IVP | `cvbx cvdx cvdemd cvkx cvkxb cvdemk` | `pvkx pvkxb pvfnx` |
| FSA | `cvfdx cvfkx cvfnx` | `pvfnx pvfkx` |
| ASA | `cvabx cvadx cvakx cvakxb` | `pvanx pvakx` |

Supplied in the `sundials/cvodes/examples_ser` directory are the following thirteen serial examples (using the NVECTOR_SERIAL module):

- `cvdx` solves a chemical kinetics problem consisting of three rate equations.

  This program solves the problem with the BDF method and Newton iteration, with the CVDENSE linear solver and a user-supplied Jacobian routine. It also uses the rootfinding feature of CVODES.

- `cvbx` solves the semi-discrete form of an advection-diffusion equation in 2-D.

  This program solves the problem with the BDF method and Newton iteration, with the CVBAND linear solver and a user-supplied Jacobian routine.

- `cvkx` solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D.

  The problem is solved with the BDF/GMRES method (i.e. using the CVSPGMR 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 preconditioner setup routine.

- `cvkxb` solves the same problem as `cvkx`, with the BDF/GMRES method and a banded preconditioner, generated by difference quotients, using the module CVBANDPRE.

  The problem is solved twice: with preconditioning on the left, then on the right.

- `cvdxe` is the same as `cvdx` but demonstrates the user-supplied error weight function feature of CVODES.

- `cvdemd` is a demonstration program for CVODES with direct linear solvers.

  Two separate problems are solved using both the Adams and BDF linear multistep methods in combination with functional and Newton iterations.

  The first problem is the Van der Pol oscillator for which the Newton iteration cases use the following types of Jacobian approximations: (1) dense (user-supplied), (2) dense (difference quotient approximation), (3) diagonal approximation. The second problem is a linear ODE with a banded lower triangular matrix derived from a 2-D advection PDE. In this case, the Newton iteration cases use the following types of Jacobian approximation: (1) banded (user-supplied), (2) banded (difference quotient approximation), (3) diagonal approximation.

- `cvdemk` is a demonstration program for CVODES with the Krylov linear solver.

  This program solves a stiff ODE system that arises from a system of partial differential equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions.

  The ODE system is solved using Newton iteration and the CVSPGMR linear solver (scaled preconditioned GMRES).

  The preconditioner matrix used is the product of two matrices: (1) a matrix, only implicitly defined, based on a fixed number of Gauss-Seidel iterations using the diffusion terms only; and (2) a block-diagonal matrix based on the partial derivatives of the interaction terms only, using block-grouping.

  Four different runs are made for this problem. The product preconditoner is applied on the left and on the right. In each case, both the modified and classical Gram-Schmidt options are tested.

- `cvfdx` solves a chemical kinetics problem consisting of three rate equations.

  CVODES computes both its solution and solution sensitivities with respect to the three reaction rate constants appearing in the model. This program solves the problem with the BDF method, Newton iteration with the CVDENSE linear solver, and a user-supplied Jacobian routine. It also uses the user-supplied error weight function feature of CVODES.

- `cvfkx` solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D space.

  CVODES computes both its solution and solution sensitivities with respect to two parameters affecting the kinetic rate terms. The problem is solved with the BDF/GMRES method (i.e. using the CVSPGMR linear solver) and the block-diagonal part of the Newton matrix as a left preconditioner.

- `cvfnx` solves the semi-discrete form of an advection-diffusion equation in 1-D.

  CVODES computes both its solution and solution sensitivities with respect to the advection and diffusion coefficients. This program solves the problem with the option for nonstiff systems, i.e. Adams method and functional iteration.

- `cvabx` solves the semi-discrete form of an advection-diffusion equation in 2-D.

  The adjoint capability of CVODES is used to compute gradients of the average (over both time and space) of the solution with respect to the initial conditions. This program solves both the forward and backward problems with the BDF method, Newton iteration with the CVBAND linear solver, and user-supplied Jacobian routines.

- `cvadx` solves a chemical kinetics problem consisting of three rate equations.

  The adjoint capability of CVODES is used to compute gradients of a functional of the solution with respect to the three reaction rate constants appearing in the model.

This program solves both the forward and backward problems with the BDF method, Newton iteration with the CVDENSE linear solver, and user-supplied Jacobian routines.

- `cvakx` solves a stiff ODE system that arises from a system of partial differential equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions.

  The adjoint capability of CVODES is used to compute gradients of the average (over both time and space) of the concentration of a selected species with respect to the initial conditions of all six species. Both the forward and backward problems are solved with the BDF/GMRES method (i.e. using the CVSPGMR linear solver) and the block-diagonal part of the Newton matrix as a left preconditioner.

- `cvakxb` solves the same problem as `cvakx`, but computes gradients of the average over space at the final time of the concentration of a selected species with respect to the initial conditions of all six species.

Supplied in the `sundials/cvode/examples_par` directory are the following six parallel examples (using the NVECTOR_PARALLEL module):

- `pvnx` solves the semi-discrete form of an advection-diffusion equation in 1-D.

  This program solves the problem with the option for nonstiff systems, i.e. Adams method and functional iteration.

- `pvkx` is the parallel implementation of `cvkx`.

- `pvkxb` solves the same problem as `pvkx`, with the BDF/GMRES method and a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the module CVBBDPRE.

- `pvfnx` is the parallel version of `cvfnx`.

- `pvfkx` is the parallel version of `cvfkx`.

- `pvanx` solves the semi-discrete form of an advection-diffusion equation in 1-D.

  The adjoint capability of CVODES is used to compute gradients of the average over space of the solution at the final time with respect to both the initial conditions and the advection and diffusion coefficients in the model. This program solves both the forward and backward problems with the option for nonstiff systems, i.e. Adams method and functional iteration.

- `pvakx` solves an adjoint sensitivity problem for an advection-diffusion PDE in 2-D or 3-D using the BDF/GMRES method and the CVBBDPRE preconditioner module on both the forward and backward phases.

  The adjoint capability of CVODES is used to compute the gradient of the space-time average of the squared solution norm with respect to problem parameters which parametrize a distributed volume source.

In the following sections, we give detailed descriptions of some (but not all) of the sensitivity analysis examples. We do not discuss any of the examples for IVP integration. The interested reader should consult the CVODE Examples Document [1]. Any CVODE problem will work with CVODES with only one modification: the main program should include the header file `cvodes.h` instead of `cvode.h`.

The Appendices contain complete listings of the examples described below. We also give our output files for each of these examples, but users should be cautioned that their results may differ slightly from these. Differences in solution values may differ within the tolerances, and differences in cumulative counters, such as numbers of steps or Newton iterations, may differ from one machine environment to another by as much as 10% to 20%.

The final section of this report describes a set of tests done with CVODES in a parallel environment (using NVECTOR_PARALLEL) on a modification of the `pvkx` example.

In the descriptions below, we make frequent references to the CVODES User Guide [2]. All citations to specific sections (e.g. §5.2) are references to parts of that user guide, unless explicitly stated otherwise.

**Note** The examples in the CVODES distribution were written in such a way as to compile and run for any combination of configuration options during the installation of SUNDIALS (see §2). As a consequence, they contain portions of code that will not typically be present in a user program. For example, all example programs make use of the variables `SUNDIALS_EXTENDED_PRECISION` and `SUNDIALS_DOUBLE_PRECISION` to test if the solver libraries were built in extended- or double-precision and use the appropriate conversion specifiers in `printf` functions. Similarly, all forward sensitivity examples can be run with or without sensitivity computations enabled and, in the former case, with various combinations of methods and error control strategies. This is achieved in these example through the program arguments.

## 2 Forward sensitivity analysis example problems

For all the above examples, any of three sensitivity methods (CV_SIMULTANEOUS, CV_STAGGERED, or CV_STAGGERED1) can be used, and sensitivities may be included in the error test or not (error control set on TRUE or FALSE, respectively).

The next two sections describe in detail a serial example (cvfdx) and a parallel one (pvfkx). For details on the other examples, the reader is directed to the comments in their source files.

### 2.1 A serial nonstiff example: cvfnx

As a first example of using CVODES for forward sensitivity analysis, we treat the simple advection-diffusion equation for $u = u(t, x)$

$$\frac{\partial u}{\partial t} = q_1 \frac{\partial^2 u}{\partial x^2} + q_2 \frac{\partial u}{\partial x} \tag{1}$$

for $0 \leq t \leq 5$, $0 \leq x \leq 2$, and subject to homogeneous Dirichlet boundary conditions and initial values given by

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

The nominal values of the problem parameters are $q_1 = 1.0$ and $q_2 = 0.5$. 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. With $u_i$ as the approximation to $u(t, x_i)$, $x_i = i(\Delta x)$, and $\Delta x = 2/(\mathtt{MX}+1)$, the resulting system of ODEs, $\dot{u} = f(t, u)$, can now be written:

$$\dot{u}_i = q_1 \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} + q_2 \frac{u_{i+1} - u_{i-1}}{2(\Delta x)}\,. \tag{3}$$

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

The sensitivity systems for $s^1 = \partial u / \partial q_1$ and $s^2 = \partial u / \partial q_2$ are simply

$$\begin{aligned} \frac{ds_i^1}{dt} &= q_1 \frac{s_{i+1}^1 - 2s_i^1 + s_{i-1}^1}{(\Delta x)^2} + q_2 \frac{s_{i+1}^1 - s_{i-1}^1}{2(\Delta x)} + \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} \\ s_i^1(0) &= 0.0 \end{aligned} \tag{4}$$

and

$$\begin{aligned} \frac{ds_i^2}{dt} &= q_1 \frac{s_{i+1}^2 - 2s_i^2 + s_{i-1}^2}{(\Delta x)^2} + q_2 \frac{s_{i+1}^2 - s_{i-1}^2}{2(\Delta x)} + \frac{u_{i+1} - u_{i-1}}{2(\Delta x)} \\ s_i^1(0) &= 0.0\,. \end{aligned} \tag{5}$$

The source file for this problem, cvfnx.c, is listed in Appendix A. It uses the Adams (non-stiff) integration formula and functional iteration. This problem is unrealistically simple *, but serves to illustrate use of the forward sensitivity capabilities in CVODES.

*Increasing the number of grid points to better resolve the PDE spatially will lead to a stiffer ODE for which the Adams integration formula will not be suitable

The `cvfnx.c` file begins by including several header files, including the main CVODES header file, the `sundialtypes.h` header file for the definition of the `realtype` type, and the NVECTOR_SERIAL header file for the definitions of the serial `N_Vector` type and operations on such vectors. Following that are definitions of problem constants and a data block for communication with the `f` routine. That block includes the problem parameters and the mesh dimension.

The `main` program begins by processing and verifying the program arguments, followed by allocation and initialization of the user-defined data structure. Next, the vector of initial conditions is created (by calling `N_VNew_Serial`) and initialized (in the function `SetIC`). The next code block creates and allocates memory for the CVODES object.

If sensitivity calculations were turned on through the command line arguments, the main program continues with setting the scaling parameters `pbar` and the array of flags `plist`. In this example, the scaling factors `pbar` are used both for the finite difference approximation to the right-hand sides of the sensitivity systems (4) and (5) and in calculating the absolute tolerances for the sensitivity variables. The flags in `plist` are set to indicate that sensitivities with respect to both problem parameters are desired. The array of `NS` $= 2$ vectors `uS` for the sensitivity variables is created by calling `N_VNewVectorArray_Serial` and set to contain the initial values ($s_i^1(0) = 0.0$, $s_i^2(0) = 0.0$).

The next three calls set optional inputs for sensitivity calculations: the sensitivity variables are included or excluded from the error test (the boolean variable `err_con` is passed as a command line argument), the control variable `rho` is set to a value `ZERO` $= 0$ to indicate the use of second-order centered directional derivative formulas for the approximations to the sensitivity right-hand sides, and the array of scaling factors `pbar` is passed to CVODES. Memory for sensitivity calculations is allocated by calling `CVodeSensMalloc` which also specifies the sensitivity solution method (`sensi_meth` is passed as a command line argument), the problem parameters `p`, and the initial conditions for the sensitivity variables.

Next, in a loop over the `NOUT` output times, the program calls the integration routine `CVode`. On a successful return, the program prints the maximum norm of the solution $u$ at the current time and, if sensitivities were also computed, extracts and prints the maximum norms of $s^1(t)$ and $s^2(t)$. The program ends by printing some final integration statistics and freeing all allocated memory.

The `f` function is a straightforward implementation of (3). The rest of the file `cvfnx.c` contains definitions of private functions. The last two, `PrintFinalStats` and `check_flag`, can be used with minor modifications by any CVODES user code to print final CVODES statistics and to check return flags from CVODES interface functions, respectively.

Results generated by `cvfnx` are shown in Fig. 1. The output generated by `cvfnx` when computing sensitivities with the `CV_SIMULTANEOUS` method and full error control (`cvfnx -sensi sim t`) is:

cvfnx sample output

```

1-D advection-diffusion equation, mesh size = 10
Sensitivity: YES ( SIMULTANEOUS + FULL ERROR CONTROL )

===========================================================
     T     Q       H      NST                     Max norm
===========================================================
5.000e-01  4  7.656e-03   115
                                Solution        3.0529e+00
```

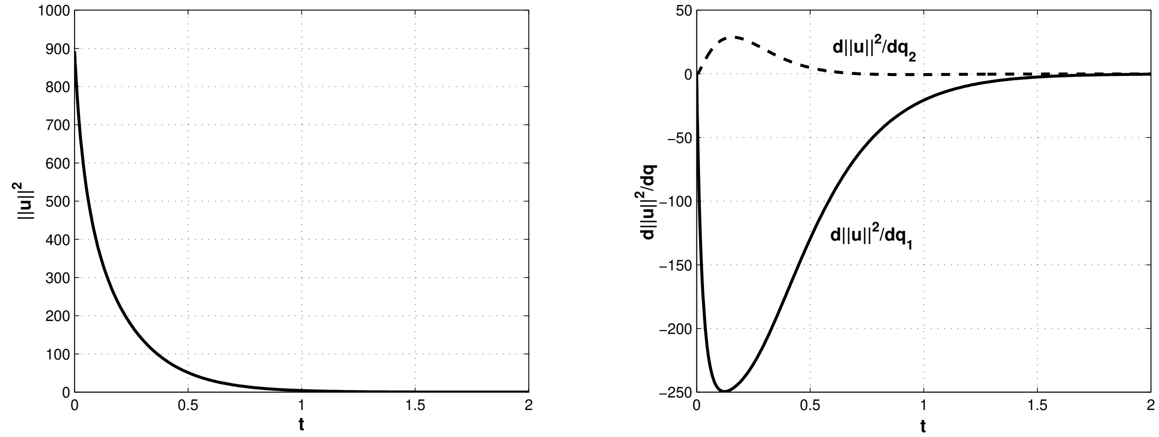


Figure 1: Results for the `cvfnx` example problem. The time evolution of the squared solution norm, $||u||^2$, is shown on the left. The figure on the right shows the evolution of the sensitivities of $||u||^2$ with respect to the two problem parameters.

```
                                  Sensitivity 1     3.8668e+00
                                  Sensitivity 2     6.2020e-01
------------------------------------------------------------
1.000e+00   4   9.525e-03   182
                                  Solution          8.7533e-01
                                  Sensitivity 1     2.1743e+00
                                  Sensitivity 2     1.8909e-01
------------------------------------------------------------
1.500e+00   3   1.040e-02   255
                                  Solution          2.4949e-01
                                  Sensitivity 1     9.1825e-01
                                  Sensitivity 2     7.3922e-02
------------------------------------------------------------
2.000e+00   2   1.271e-02   330
                                  Solution          7.1097e-02
                                  Sensitivity 1     3.4667e-01
                                  Sensitivity 2     2.8228e-02
------------------------------------------------------------
2.500e+00   2   1.629e-02   402
                                  Solution          2.0260e-02
                                  Sensitivity 1     1.2301e-01
                                  Sensitivity 2     1.0085e-02
------------------------------------------------------------
3.000e+00   2   3.820e-03   473
                                  Solution          5.7734e-03
                                  Sensitivity 1     4.1956e-02
                                  Sensitivity 2     3.4556e-03
------------------------------------------------------------
3.500e+00   2   8.988e-03   540
                                  Solution          1.6451e-03
                                  Sensitivity 1     1.3922e-02
                                  Sensitivity 2     1.1669e-03
------------------------------------------------------------
4.000e+00   2   1.199e-02   617
```

```
                                Solution         4.6945e-04
                                Sensitivity 1    4.5300e-03
                                Sensitivity 2    3.8674e-04
------------------------------------------------------------
4.500e+00  3  4.744e-03   680
                                Solution         1.3422e-04
                                Sensitivity 1    1.4548e-03
                                Sensitivity 2    1.2589e-04
------------------------------------------------------------
5.000e+00  1  4.010e-03   757
                                Solution         3.8656e-05
                                Sensitivity 1    4.6451e-04
                                Sensitivity 2    4.0616e-05
------------------------------------------------------------

Final Statistics

nst     =   757

nfe     =  1372
netf    =     1    nsetups  =     0
nni     =  1369    ncfn     =   117

nfSe    =  2744    nfeS     =  5488
netfs   =     0    nsetupsS =     0
nniS    =     0    ncfnS    =     0
```

The output generated by `cvfnx` when computing sensitivities with the `CV_STAGGERED1` method and partial error control (`cvfnx -sensi stg1 f`) is:

cvfnx sample output

```
1-D advection-diffusion equation, mesh size = 10
Sensitivity: YES ( STAGGERED + PARTIAL ERROR CONTROL )

============================================================
     T     Q       H      NST                    Max norm
============================================================
5.000e-01  3  7.876e-03   115
                                Solution         3.0529e+00
                                Sensitivity 1    3.8668e+00
                                Sensitivity 2    6.2020e-01
------------------------------------------------------------
1.000e+00  3  1.145e-02   208
                                Solution         8.7533e-01
                                Sensitivity 1    2.1743e+00
                                Sensitivity 2    1.8909e-01
------------------------------------------------------------
1.500e+00  2  9.985e-03   287
                                Solution         2.4948e-01
                                Sensitivity 1    9.1826e-01
                                Sensitivity 2    7.3913e-02
------------------------------------------------------------
```

```
2.000e+00  2  4.223e-03   388
                                Solution         7.1096e-02
                                Sensitivity 1    3.4667e-01
                                Sensitivity 2    2.8228e-02
------------------------------------------------------------
2.500e+00  2  4.220e-03   507
                                Solution         2.0261e-02
                                Sensitivity 1    1.2301e-01
                                Sensitivity 2    1.0085e-02
------------------------------------------------------------
3.000e+00  2  4.220e-03   625
                                Solution         5.7738e-03
                                Sensitivity 1    4.1957e-02
                                Sensitivity 2    3.4557e-03
------------------------------------------------------------
3.500e+00  2  4.220e-03   744
                                Solution         1.6454e-03
                                Sensitivity 1    1.3923e-02
                                Sensitivity 2    1.1670e-03
------------------------------------------------------------
4.000e+00  2  4.220e-03   862
                                Solution         4.6887e-04
                                Sensitivity 1    4.5282e-03
                                Sensitivity 2    3.8632e-04
------------------------------------------------------------
4.500e+00  2  4.220e-03   981
                                Solution         1.3364e-04
                                Sensitivity 1    1.4502e-03
                                Sensitivity 2    1.2546e-04
------------------------------------------------------------
5.000e+00  2  4.220e-03  1099
                                Solution         3.8105e-05
                                Sensitivity 1    4.5891e-04
                                Sensitivity 2    4.0166e-05
------------------------------------------------------------

Final Statistics

nst     =  1099

nfe     =  3157
netf    =     3    nsetups  =     0
nni     =  1657    ncfn     =    11

nfSe    =  4838    nfeS     =  9676
netfs   =     0    nsetupsS =     0
nniS    =  2418    ncfnS    =   398
```

## 2.2 A serial dense example: `cvfdx`

This example is a modification of the chemical kinetics problem described in [1] which computes, in addition to the solution of the IVP, sensitivities of the solution with respect to the three reaction rates involved in the model. The ODEs are written as:

$$
\begin{aligned}
\dot{y}_1 &= -p_1 y_1 + p_2 y_2 y_3 \\
\dot{y}_2 &= p_1 y_1 - p_2 y_2 y_3 - p_3 y_2^2 \\
\dot{y}_3 &= p_3 y_2^2 \,,
\end{aligned}
\tag{6}
$$

with initial conditions at $t_0 = 0$, $y_1 = 1$ and $y_2 = y_3 = 0$. The nominal values of the reaction rate constants are $p_1 = 0.04$, $p_2 = 10^4$ and $p_3 = 3 \cdot 10^7$. The sensitivity systems that are solved together with (6) are

$$
\begin{aligned}
&\dot{s}_i = \begin{bmatrix} -p_1 & p_2 y_3 & p_2 y_2 \\ p_1 & -p_2 y_3 - 2 p_3 y_2 & -p_2 y_2 \\ 0 & 2 p_3 y_2 & 0 \end{bmatrix} s_i + \frac{\partial f}{\partial p_i} \,, \quad s_i(t_0) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \,, \quad i = 1,2,3 \\
&\frac{\partial f}{\partial p_1} = \begin{bmatrix} -y_1 \\ y_1 \\ 0 \end{bmatrix} , \quad \frac{\partial f}{\partial p_2} = \begin{bmatrix} y_2 y_3 \\ -y_2 y_3 \\ 0 \end{bmatrix} , \quad \frac{\partial f}{\partial p_3} = \begin{bmatrix} 0 \\ -y_2^2 \\ y_2^2 \end{bmatrix} .
\end{aligned}
\tag{7}
$$

The source code for this example is listed in App. B. The main program is described below with emphasis on the sensitivity related components. These explanations, together with those given for the code `cvdx` in [1], will also provide the user with a template for instrumenting an existing simulation code to perform forward sensitivity analysis. As will be seen from this example, an existing simulation code can be modified to compute sensitivity variables (in addition to state variables) by only inserting a few CVODES calls into the main program.

First note that no new header files need be included. In addition to the constants already defined in `cvdx`, we define the number of model parameters, `NP` ($= 3$), the number of sensitivity parameters, `NS` ($= 3$), and a constant `ZERO` $= 0.0$.

As mentioned in §6.1, the user data structure `f_data` must provide access to the array of model parameters as the only way for CVODES to communicate parameter values to the right-hand side function `f`. In the `cvfdx` example this is done by defining `f_data` to be of type `UserData`, i.e. a pointer to a structure which contains an array of `NP` `realtype` values.

Four user-supplied functions are defined. The function `f`, passed to `CVodeMalloc`, computes the righ-hand side of the ODE (6), while `Jac` computes the dense Jacobian of the problem and is attached to the dense linear solver module CVDENSE through a call to `CVDenseSetJacFn`. The function `fS` computes the right-hand side of each sensitivity system (7) for one parameter at a time and is therefore of type `SensRhs1`. Finally, the function `ewt` computes the error weights for the WRMS norm estimations within CVODES.

The program prologue ends by defining six private helper functions. The first two, `ProcessArgs` and `WrongArgs` (which would not be present in a typical user code), parse and verify the command line arguments to `cvfdx`, respectively. After each successful return from the main CVODES integrator, the functions `PrintOutput` and `PrintOutputS` print the state and sensitivity variables, respectively. The function `PrintFinalStats` is caled after completion of the integration to print solver statistics. The function `check_flag` is used to check the return flag from any of the CVODES interface functions called by `cvfdx`.

The `main` function begins with definitions and type declarations. Among these, it defines the vector `pbar` of `NS` scaling factors for the model parameters `p` and the array `yS` of `N_Vector` which will contain the initial conditions and solutions for the sensitivity variables. It also declares the variable `data` of type `UserData` which will contain the user-defined data structure to be passed to CVODES and used in the evaluation of the ODE right-hand sides.

The first code block in `main` deals with reading and interpreting the command line arguments. `cvfdx` can be run with or without sensitivity computations turned on and with different selections for the sensitivity method and error control strategy.

The user's data structure is then allocated and its field $p$ is set to contain the values of the three problem parameters. The next block of code is identical to that in `cvdx.c` (see [1]) and involves allocation and initialization of the state variables and creation and initialization of `cvode_mem`, the CVODES solver memory. It specifies that a user-provided function (`ewt`) is to be used for computing the error weights. It also attaches CVDENSE, with a non-`NULL` Jacobian function, as the linear solver to be used in the Newton nonlinear solver.

If sensitivity analysis is enabled (through the command line arguments), the main program will then set the scaling parameters `pbar` ($\mathtt{pbar}_i = \mathtt{p}_i$, which can typically be used for nonzero model parameters). Next, the program allocates memory for `yS`, by calling the NVECTOR_SERIAL function `N_VNewVectorArray_Serial`, and initializaes all sensitivity variables to 0.0.

The call to `CVodeSensMalloc` specifies the sensitivity solution method through `sensi_meth` (read from the command line arguments) as `CV_SIMULTANEOUS`, `CV_STAGGERED`, or `CV_STAGGERED1`.

The next four calls specify optional inputs for forward sensitivity analysis: the user-defined routine for evaluation of the right-hand sides of sensitivity equations, the error control strategy (read from the command line arguments), the pointer to user data to be passed to `fS` whenever it is called, and the information on the model parameters. In this example, only `pbar` is needed for the estimation of absolute senisitivity variables tolerances. Neither `p` nor `plist` are required since the sensitivity right-hand sides are computed in a user-provided function (`fS`). As a consequance, we pass `NULL` for the corresponding arguments in `CVodeSetSensParams`.

Note that this example uses the default estimates for the relative and absolute tolerances `rtolS` and `atolS` for sensitivity variables, based on the tolerances for state variables and the scaling parameters `pbar` (see §3.2 for details).

Next, in a loop over the `NOUT` output times, the program calls the integration routine `CVode` which, if sensitivity analysis was initialized through the call to `CVodeSensMalloc`, computes both state and sensitivity variables. However, `CVode` returns only the state solution at `tout` in the vector `y`. The program tests the return from `CVode` for a value other than `CV_SUCCESS` and prints the state variables. Sensitivity variables at `tout` are loaded into `yS` by calling `CVodeGetSens`. The program tests the return from `CVodeGetSens` for a value other than `CV_SUCCESS` and then prints the sensitivity variables.

Finally, the program prints some statistics (function `PrintFinalStats`) and deallocates memory through calls to `N_VDestroy_Serial`, `N_VDestroyVectorArray_Serial`, `CVodeFree`, and `free` for the user data structure.

The user-supplied functions `f` for the right-hand side of the original ODEs and `Jac` for the system Jacobian are identical to those in `cvdx.c` with the notable exeption that model parameters are extracted from the user-defined data structure `f_data`, which must first be cast to the `UserData` type. similarly, the user-supplied function `ewt` is identical to that in `cvdx.c`. The user-supplied function `fS` computes the sensitivity right-hand side for the

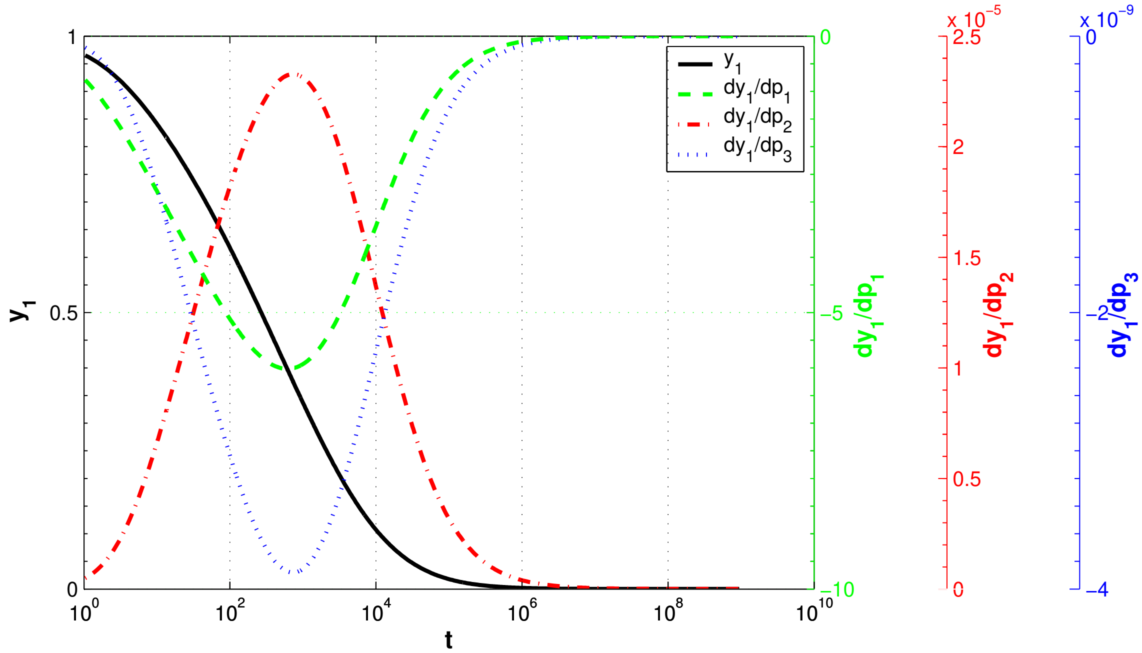


Figure 2: Results for the cvfdx example problem: time evolution of $y_1$ and its sensitivities with respect to the three problem parameters.

iS-th sensitivity equation.

Results generated by cvfdx are shown in Fig. 2. Sample outputs from cvfdx, for two different combinations of command line arguments, follows. The command to execute this program must have the form:

```
% cvfdx -nosensi
```

if no sensitivity calculations are desired, or

```
% cvfdx -sensi sensi_meth err_con
```

where `sensi_meth` must be one of `sim`, `stg`, or `stg1` to indicate the `CV_SIMULTANEOUS`, `CV_STAGGERED`, or `CV_STAGGERED1` method, respectively and `err_con` must be one of `t` or `f` to include or exclude, respectively, the sensitivity variables from the error test.

The output generated by `cvfdx` when computing sensitivities with the `CV_SIMULTANEOUS` method and full error control (`cvfdx -sensi sim t`) is:

cvfdx sample output

```

3-species chemical kinetics problem
Sensitivity: YES ( SIMULTANEOUS + FULL ERROR CONTROL )

===========================================================================
     T      Q        H       NST                      y1            y2            y3
===========================================================================
4.000e-01   3   4.881e-02   115
                            Solution           9.8517e-01    3.3864e-05    1.4794e-02
                            Sensitivity 1     -3.5595e-01    3.9025e-04    3.5556e-01
```

```
                             Sensitivity 2    9.5431e-08  -2.1309e-10  -9.5218e-08
                             Sensitivity 3   -1.5833e-11  -5.2900e-13   1.6362e-11
-------------------------------------------------------------------------------------
4.000e+00  5  2.363e-01    138
                             Solution         9.0552e-01   2.2405e-05   9.4459e-02
                             Sensitivity 1   -1.8761e+00   1.7922e-04   1.8759e+00
                             Sensitivity 2    2.9614e-06  -5.8305e-10  -2.9608e-06
                             Sensitivity 3   -4.9334e-10  -2.7626e-13   4.9362e-10
-------------------------------------------------------------------------------------
4.000e+01  3  1.485e+00    219
                             Solution         7.1583e-01   9.1856e-06   2.8416e-01
                             Sensitivity 1   -4.2475e+00   4.5913e-05   4.2475e+00
                             Sensitivity 2    1.3731e-05  -2.3573e-10  -1.3730e-05
                             Sensitivity 3   -2.2883e-09  -1.1380e-13   2.2884e-09
-------------------------------------------------------------------------------------
4.000e+02  3  8.882e+00    331
                             Solution         4.5052e-01   3.2229e-06   5.4947e-01
                             Sensitivity 1   -5.9584e+00   3.5431e-06   5.9584e+00
                             Sensitivity 2    2.2738e-05  -2.2605e-11  -2.2738e-05
                             Sensitivity 3   -3.7896e-09  -4.9948e-14   3.7897e-09
-------------------------------------------------------------------------------------
4.000e+03  2  1.090e+02    486
                             Solution         1.8317e-01   8.9403e-07   8.1683e-01
                             Sensitivity 1   -4.7500e+00  -5.9957e-06   4.7500e+00
                             Sensitivity 2    1.8809e-05   2.3136e-11  -1.8809e-05
                             Sensitivity 3   -3.1348e-09  -1.8757e-14   3.1348e-09
-------------------------------------------------------------------------------------
4.000e+04  3  1.178e+03    588
                             Solution         3.8977e-02   1.6215e-07   9.6102e-01
                             Sensitivity 1   -1.5748e+00  -2.7620e-06   1.5748e+00
                             Sensitivity 2    6.2869e-06   1.1002e-11  -6.2869e-06
                             Sensitivity 3   -1.0478e-09  -4.5362e-15   1.0478e-09
-------------------------------------------------------------------------------------
4.000e+05  3  1.514e+04    645
                             Solution         4.9387e-03   1.9852e-08   9.9506e-01
                             Sensitivity 1   -2.3639e-01  -4.5861e-07   2.3639e-01
                             Sensitivity 2    9.4525e-07   1.8334e-12  -9.4525e-07
                             Sensitivity 3   -1.5751e-10  -6.3629e-16   1.5751e-10
-------------------------------------------------------------------------------------
4.000e+06  4  2.323e+05    696
                             Solution         5.1684e-04   2.0684e-09   9.9948e-01
                             Sensitivity 1   -2.5667e-02  -5.1064e-08   2.5667e-02
                             Sensitivity 2    1.0266e-07   2.0424e-13  -1.0266e-07
                             Sensitivity 3   -1.7111e-11  -6.8513e-17   1.7111e-11
-------------------------------------------------------------------------------------
4.000e+07  4  1.776e+06    753
                             Solution         5.2039e-05   2.0817e-10   9.9995e-01
                             Sensitivity 1   -2.5991e-03  -5.1931e-09   2.5991e-03
                             Sensitivity 2    1.0396e-08   2.0772e-14  -1.0397e-08
                             Sensitivity 3   -1.7330e-12  -6.9328e-18   1.7330e-12
-------------------------------------------------------------------------------------
4.000e+08  4  2.766e+07    802
                             Solution         5.2106e-06   2.0842e-11   9.9999e-01
                             Sensitivity 1   -2.6063e-04  -5.2149e-10   2.6063e-04
```

```
                           Sensitivity 2    1.0425e-09   2.0859e-15  -1.0425e-09
                           Sensitivity 3   -1.7366e-13  -6.9467e-19   1.7367e-13
--------------------------------------------------------------------------------
4.000e+09  2  4.183e+08   836
                           Solution         5.1881e-07   2.0752e-12   1.0000e-00
                           Sensitivity 1   -2.5907e-05  -5.1717e-11   2.5907e-05
                           Sensitivity 2    1.0363e-10   2.0687e-16  -1.0363e-10
                           Sensitivity 3   -1.7293e-14  -6.9174e-20   1.7293e-14
--------------------------------------------------------------------------------
4.000e+10  2  3.799e+09   859
                           Solution         6.5181e-08   2.6072e-13   1.0000e-00
                           Sensitivity 1   -2.4884e-06  -3.3032e-12   2.4884e-06
                           Sensitivity 2    9.9534e-12   1.3213e-17  -9.9534e-12
                           Sensitivity 3   -2.1727e-15  -8.6908e-21   2.1727e-15
--------------------------------------------------------------------------------

Final Statistics

nst     =   859

nfe     =  1221
netf    =    29    nsetups  =   142
nni     =  1218    ncfn     =     4

nfSe    =  3663    nfeS     =     0
netfs   =     0    nsetupsS =     0
nniS    =     0    ncfnS    =     0

njeD    =    24    nfeD     =     0
```

The output generated by `cvfdx` when computing sensitivities with the `CV_STAGGERED1` method and partial error control (`cvfdx -sensi stg1 f`) is:

cvfdx sample output

```
3-species chemical kinetics problem
Sensitivity: YES ( STAGGERED + PARTIAL ERROR CONTROL )

================================================================================
     T     Q       H      NST                   y1           y2           y3
================================================================================
4.000e-01  3  1.205e-01    59
                           Solution         9.8517e-01   3.3863e-05   1.4797e-02
                           Sensitivity 1   -3.5611e-01   3.9023e-04   3.5572e-01
                           Sensitivity 2    9.4831e-08  -2.1325e-10  -9.4618e-08
                           Sensitivity 3   -1.5733e-11  -5.2897e-13   1.6262e-11
--------------------------------------------------------------------------------
4.000e+00  4  5.316e-01    74
                           Solution         9.0552e-01   2.2404e-05   9.4461e-02
                           Sensitivity 1   -1.8761e+00   1.7922e-04   1.8760e+00
                           Sensitivity 2    2.9612e-06  -5.8308e-10  -2.9606e-06
                           Sensitivity 3   -4.9330e-10  -2.7624e-13   4.9357e-10
--------------------------------------------------------------------------------
```

```
4.000e+01  3  1.445e+00   116
                          Solution         7.1584e-01   9.1854e-06   2.8415e-01
                          Sensitivity 1   -4.2474e+00   4.5928e-05   4.2473e+00
                          Sensitivity 2    1.3730e-05  -2.3573e-10  -1.3729e-05
                          Sensitivity 3   -2.2883e-09  -1.1380e-13   2.2884e-09
-------------------------------------------------------------------------------
4.000e+02  3  1.605e+01   164
                          Solution         4.5054e-01   3.2228e-06   5.4946e-01
                          Sensitivity 1   -5.9582e+00   3.5498e-06   5.9582e+00
                          Sensitivity 2    2.2737e-05  -2.2593e-11  -2.2737e-05
                          Sensitivity 3   -3.7895e-09  -4.9947e-14   3.7896e-09
-------------------------------------------------------------------------------
4.000e+03  3  1.474e+02   227
                          Solution         1.8321e-01   8.9422e-07   8.1679e-01
                          Sensitivity 1   -4.7501e+00  -5.9934e-06   4.7501e+00
                          Sensitivity 2    1.8809e-05   2.3126e-11  -1.8809e-05
                          Sensitivity 3   -3.1348e-09  -1.8759e-14   3.1348e-09
-------------------------------------------------------------------------------
4.000e+04  3  2.331e+03   307
                          Solution         3.8978e-02   1.6215e-07   9.6102e-01
                          Sensitivity 1   -1.5749e+00  -2.7623e-06   1.5749e+00
                          Sensitivity 2    6.2868e-06   1.1001e-11  -6.2868e-06
                          Sensitivity 3   -1.0479e-09  -4.5364e-15   1.0479e-09
-------------------------------------------------------------------------------
4.000e+05  3  2.342e+04   349
                          Solution         4.9410e-03   1.9861e-08   9.9506e-01
                          Sensitivity 1   -2.3638e-01  -4.5834e-07   2.3638e-01
                          Sensitivity 2    9.4515e-07   1.8319e-12  -9.4515e-07
                          Sensitivity 3   -1.5757e-10  -6.3653e-16   1.5757e-10
-------------------------------------------------------------------------------
4.000e+06  4  1.723e+05   391
                          Solution         5.1690e-04   2.0686e-09   9.9948e-01
                          Sensitivity 1   -2.5662e-02  -5.1036e-08   2.5662e-02
                          Sensitivity 2    1.0264e-07   2.0412e-13  -1.0264e-07
                          Sensitivity 3   -1.7110e-11  -6.8509e-17   1.7110e-11
-------------------------------------------------------------------------------
4.000e+07  4  4.952e+06   439
                          Solution         5.1984e-05   2.0795e-10   9.9995e-01
                          Sensitivity 1   -2.5970e-03  -5.1903e-09   2.5970e-03
                          Sensitivity 2    1.0388e-08   2.0761e-14  -1.0388e-08
                          Sensitivity 3   -1.7312e-12  -6.9256e-18   1.7312e-12
-------------------------------------------------------------------------------
4.000e+08  3  2.444e+07   491
                          Solution         5.2121e-06   2.0849e-11   9.9999e-01
                          Sensitivity 1   -2.6067e-04  -5.2146e-10   2.6067e-04
                          Sensitivity 2    1.0427e-09   2.0858e-15  -1.0427e-09
                          Sensitivity 3   -1.7385e-13  -6.9541e-19   1.7385e-13
-------------------------------------------------------------------------------
4.000e+09  4  1.450e+08   525
                          Solution         5.0539e-07   2.0216e-12   1.0000e-00
                          Sensitivity 1   -2.6111e-05  -5.3906e-11   2.6111e-05
                          Sensitivity 2    1.0445e-10   2.1562e-16  -1.0445e-10
                          Sensitivity 3   -1.7437e-14  -6.9746e-20   1.7437e-14
-------------------------------------------------------------------------------
```

```
4.000e+10  5  7.934e+08   579
                          Solution         5.9422e-08   2.3769e-13   1.0000e-00
                          Sensitivity 1   -2.8007e-06  -5.2605e-12   2.8007e-06
                          Sensitivity 2    1.1203e-11   2.1042e-17  -1.1203e-11
                          Sensitivity 3   -1.7491e-15  -6.9963e-21   1.7491e-15
-------------------------------------------------------------------------------

Final Statistics

nst     =   579

nfe     =  1379
netf    =    25    nsetups  =   109
nni     =   797    ncfn     =     0

nfSe    =  2829    nfeS     =     0
netfs   =     0    nsetupsS =     3
nniS    =   942    ncfnS    =     0

njeD    =    11    nfeD     =     0
```

### 2.3 An SPGMR parallel example with user preconditioner: `pvfkx`

As an example of using the forward sensitivity capabilities in CVODES with the Krylov linear solver CVSPGMR and the NVECTOR_PARALLEL module, we describe a test problem based on the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D space, for which we compute solution sensitivities with respect to problem parameters ($q_1$ and $q_2$) that appear in the kinetic rate terms. The PDE is

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y} K_v(y) \frac{\partial c^i}{\partial y} + R^i(c^1, c^2, t) \quad (i = 1, 2)\,, \tag{8}$$

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

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2 q_3(t) c^3 + q_4(t) c^2 \ , \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 - q_4(t) c^2 \ . \end{aligned} \tag{9}$$

The spatial domain is $0 \le x \le 20,\ 30 \le y \le 50$ (in *km*). The various constants and parameters are: $K_h = 4.0 \cdot 10^{-6},\ V = 10^{-3},\ K_v = 10^{-8} \exp(y/5),\ q_1 = 1.63 \cdot 10^{-16},\ q_2 = 4.66 \cdot 10^{-16},\ c^3 = 3.7 \cdot 10^{16}$, and the diurnal rate constants are defined as:

$$q_i(t) = \left\{ \begin{array}{ll} \exp[-a_i / \sin \omega t], & \text{for } \sin \omega t > 0 \\ 0, & \text{for } \sin \omega t \le 0 \end{array} \right\} \quad (i = 3, 4)\,,$$

where $\omega = \pi/43200,\ a_3 = 22.62,\ a_4 = 7.601$. 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

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

We discretize the PDE system with central differencing, to obtain an ODE system $\dot{u} = f(t, u)$ representing (8). In this case, the discrete solution vector is distributed across many processes. Specifically, we may think of the processes as being laid out in a rectangle, and each process being assigned a subgrid of size `MXSUB`×`MYSUB` of the $x - y$ grid. If there are `NPEX` processes in the $x$ direction and `NPEY` processes in the $y$ direction, then the overall grid size is `MX`×`MY` with `MX`=`NPEX`×`MXSUB` and `MY`=`NPEY`×`MYSUB`, and the size of the ODE system is 2·`MX`·`MY`.

To compute $f$ in this setting, the processes pass and receive information as follows. The solution components for the bottom row of grid points assigned to the current process are passed to the process below it, and the solution for the top row of grid points is received from the process below the current process. The solution for the top row of grid points for the current process is sent to the process above the current process, while the solution for the bottom row of grid points is received from that process by the current process. Similarly, the solution for the first column of grid points is sent from the current process to the process to its left, and the last column of grid points is received from that process by the current process. The communication for the solution at the right edge of the process

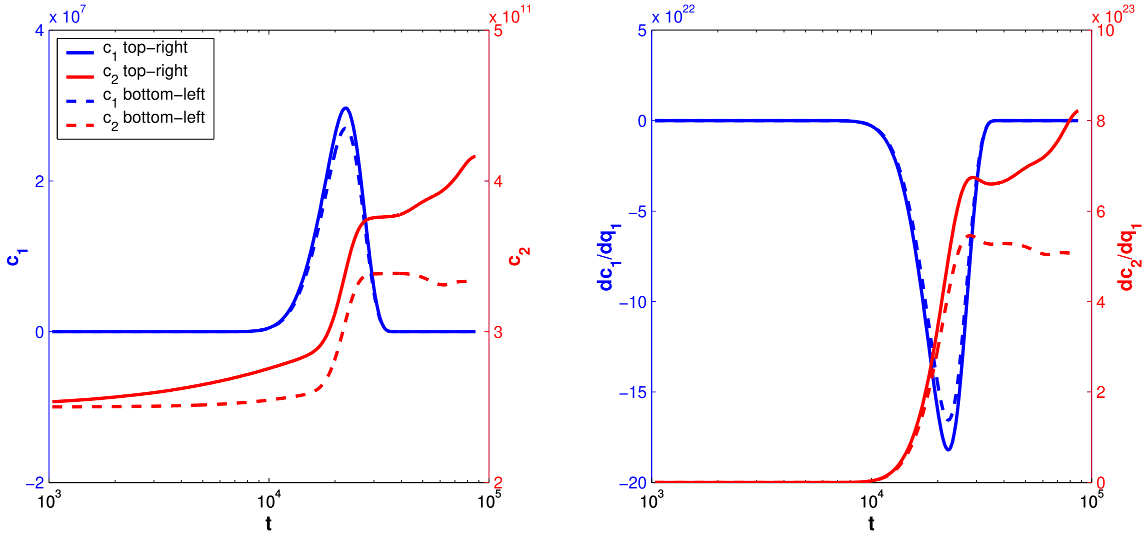


Figure 3: Results for the pvfkx example problem: time evolution of $c_1$ and $c_2$ at the bottom-left and top-right corners (left) and of their sensitivities with respect to $q_1$.

is similar. If this is the last process in a particular direction, then message passing and receiving are bypassed for that direction.

The source code for this example is listed in App. C. The overall structure of the `main` function is very similar to that of the code `cvfdx` described above with differences arising from the use of the parallel NVECTOR module - NVECTOR_PARALLEL. On the other hand, the user-supplied routines in `pvfkx`, `f` for the right-hand side of the original system, `Precond` for the preconditioner setup, and `PSolve` for the preconditioner solve, are identical to those defined for the sample program `pvkx` described in [1]. The only difference is in the routine `fcalc`, which operates on local data only and contains the actual calculation of $f(t, u)$, where the problem parameters are first extracted from the user data structure `data`. The program `pvfkx` defines no additional user-supplied routines, as it uses the CVODES internal difference quotient routines to compute the sensitivity equation right-hand sides.

Sample results generated by `pvfkx` are shown in Fig. 3. These results were generated on a $(2 \times 40) \times (2 \times 40)$ grid.

Sample outputs from `pvfkx`, for two different combinations of command line arguments, follow. The command to execute this program must have the form:

```
% mpirun -np nproc pvfkx -nosensi
```

if no sensitivity calculations are desired, or

```
% mpirun -np nproc pvfkx -sensi sensi_meth err_con
```

where `nproc` is the number of processes, `sensi_meth` must be one of `sim`, `stg`, or `stg1` to indicate the `CV_SIMULTANEOUS`, `CV_STAGGERED`, or `CV_STAGGERED1` method, respectively, and `err_con` must be one of `t` or `f` to select the full or partial error control strategy, respectively.

The output generated by `pvfkx` when computing sensitivities with the `CV_SIMULTANEOUS` method and full error control (`mpirun -np 4 pvfkx -sensi sim t`) is:

pvfkx sample output

```
2-species diurnal advection-diffusion problem
Sensitivity: YES ( SIMULTANEOUS + FULL ERROR CONTROL )

========================================================================
     T     Q       H      NST                   Bottom left  Top right
========================================================================
7.200e+03  3  3.190e+01   408
                                Solution         1.0468e+04   1.1185e+04
                                                 2.5267e+11   2.6998e+11
                                ----------------------------------------
                                Sensitivity 1   -6.4201e+19  -6.8598e+19
                                                 7.1178e+19   7.6557e+19
                                ----------------------------------------
                                Sensitivity 2   -4.3853e+14  -5.0065e+14
                                                -2.4407e+18  -2.7842e+18
------------------------------------------------------------------------
1.440e+04  3  6.016e+01   722
                                Solution         6.6590e+06   7.3008e+06
                                                 2.5819e+11   2.8329e+11
                                ----------------------------------------
                                Sensitivity 1   -4.0848e+22  -4.4785e+22
                                                 5.9550e+22   6.7173e+22
                                ----------------------------------------
                                Sensitivity 2   -4.5235e+17  -5.4318e+17
                                                -6.5419e+21  -7.8316e+21
------------------------------------------------------------------------
2.160e+04  4  1.654e+02   891
                                Solution         2.6650e+07   2.9308e+07
                                                 2.9928e+11   3.3134e+11
                                ----------------------------------------
                                Sensitivity 1   -1.6346e+23  -1.7976e+23
                                                 3.8203e+23   4.4991e+23
                                ----------------------------------------
                                Sensitivity 2   -7.6601e+18  -9.4433e+18
                                                -7.6459e+22  -9.4502e+22
------------------------------------------------------------------------
2.880e+04  2  3.838e+01  1163
                                Solution         8.7021e+06   9.6501e+06
                                                 3.3804e+11   3.7510e+11
                                ----------------------------------------
                                Sensitivity 1   -5.3375e+22  -5.9188e+22
                                                 5.4487e+23   6.7431e+23
                                ----------------------------------------
                                Sensitivity 2   -4.8855e+18  -6.1040e+18
                                                -1.7194e+23  -2.1518e+23
------------------------------------------------------------------------
3.600e+04  3  2.170e+01  1314
                                Solution         1.4040e+04   1.5609e+04
                                                 3.3868e+11   3.7652e+11
                                ----------------------------------------
                                Sensitivity 1   -8.6141e+19  -9.5762e+19
```

```
                                                 5.2719e+23   6.6030e+23
                                ----------------------------------------
                                Sensitivity 2   -8.4328e+15  -1.0549e+16
                                                -1.8439e+23  -2.3096e+23
------------------------------------------------------------------------
4.320e+04  4  2.092e+02  1536
                                Solution         3.6878e-11  -1.0815e-09
                                                 3.3823e+11   3.8035e+11
                                ----------------------------------------
                                Sensitivity 1   -1.9731e+07  -7.0822e+08
                                                 5.2753e+23   6.7448e+23
                                ----------------------------------------
                                Sensitivity 2    1.2858e+05   9.8076e+04
                                                -1.8455e+23  -2.3595e+23
------------------------------------------------------------------------
5.040e+04  4  1.157e+02  1582
                                Solution         2.3269e-08   4.7447e-08
                                                 3.3582e+11   3.8645e+11
                                ----------------------------------------
                                Sensitivity 1    1.0705e+10   2.1719e+10
                                                 5.2067e+23   6.9665e+23
                                ----------------------------------------
                                Sensitivity 2    6.5892e+07   1.5308e+08
                                                -1.8214e+23  -2.4371e+23
------------------------------------------------------------------------
5.760e+04  4  3.571e+02  1625
                                Solution        -3.4188e-09  -7.4739e-09
                                                 3.3203e+11   3.9090e+11
                                ----------------------------------------
                                Sensitivity 1    8.5523e+08   1.8651e+09
                                                 5.0826e+23   7.1206e+23
                                ----------------------------------------
                                Sensitivity 2    4.8332e+04   1.2647e+05
                                                -1.7780e+23  -2.4910e+23
------------------------------------------------------------------------
6.480e+04  4  1.407e+02  1665
                                Solution        -1.0371e-07  -2.4825e-07
                                                 3.3130e+11   3.9634e+11
                                ----------------------------------------
                                Sensitivity 1    7.3888e+08   1.7759e+09
                                                 5.0443e+23   7.3274e+23
                                ----------------------------------------
                                Sensitivity 2    2.4183e+05   8.3391e+05
                                                -1.7646e+23  -2.5633e+23
------------------------------------------------------------------------
7.200e+04  4  3.400e+02  1710
                                Solution         1.1661e-12   2.9058e-12
                                                 3.3297e+11   4.0389e+11
                                ----------------------------------------
                                Sensitivity 1    8.0876e+05   2.0026e+06
                                                 5.0784e+23   7.6383e+23
                                ----------------------------------------
                                Sensitivity 2    2.3124e+01   6.8345e+01
                                                -1.7766e+23  -2.6721e+23
```

```
------------------------------------------------------------------------
7.920e+04  5  5.172e+02  1725
                               Solution        -4.1258e-15  -1.0944e-14
                                                3.3344e+11   4.1203e+11
                               ----------------------------------------
                               Sensitivity 1   -9.7653e+04  -2.5314e+05
                                                5.0731e+23   7.9960e+23
                               ----------------------------------------
                               Sensitivity 2    7.4987e-01   2.1107e+00
                                               -1.7747e+23  -2.7972e+23
------------------------------------------------------------------------
8.640e+04  5  5.172e+02  1739
                               Solution         1.7392e-18   2.4267e-18
                                                3.3518e+11   4.1625e+11
                               ----------------------------------------
                               Sensitivity 1   -2.0753e+03  -5.5491e+03
                                                5.1171e+23   8.2143e+23
                               ----------------------------------------
                               Sensitivity 2   -5.2491e-02  -1.6658e-01
                                               -1.7901e+23  -2.8736e+23
------------------------------------------------------------------------

Final Statistics

nst     =  1739

nfe     =  2421
netf    =   103    nsetups  =   315
nni     =  2418    ncfn     =     3

nfSe    =  4842    nfeS     =  9684
netfs   =     0    nsetupsS =     0
nniS    =     0    ncfnS    =     0
```

The output generated by pvfkx when computing sensitivities with the CV_STAGGERED1 method and partial error control (mpirun -np 4 pvfkx -sensi stg1 f) is:

pvfkx sample output

```
2-species diurnal advection-diffusion problem
Sensitivity: YES ( STAGGERED + PARTIAL ERROR CONTROL )

========================================================================
     T     Q       H      NST                     Bottom left  Top right
========================================================================
7.200e+03  5  1.587e+02   219
                               Solution         1.0468e+04   1.1185e+04
                                                2.5267e+11   2.6998e+11
                               ----------------------------------------
                               Sensitivity 1   -6.4201e+19  -6.8598e+19
                                                7.1178e+19   7.6555e+19
                               ----------------------------------------
                               Sensitivity 2   -4.3853e+14  -5.0065e+14
```

```
                                               -2.4407e+18  -2.7842e+18
------------------------------------------------------------------------
1.440e+04  5  3.772e+02   251
                                Solution         6.6590e+06   7.3008e+06
                                                 2.5819e+11   2.8329e+11
                                ----------------------------------------
                                Sensitivity 1   -4.0848e+22  -4.4785e+22
                                                 5.9550e+22   6.7173e+22
                                ----------------------------------------
                                Sensitivity 2   -4.5235e+17  -5.4317e+17
                                                -6.5418e+21  -7.8315e+21
------------------------------------------------------------------------
2.160e+04  5  2.746e+02   277
                                Solution         2.6650e+07   2.9308e+07
                                                 2.9928e+11   3.3134e+11
                                ----------------------------------------
                                Sensitivity 1   -1.6346e+23  -1.7976e+23
                                                 3.8203e+23   4.4991e+23
                                ----------------------------------------
                                Sensitivity 2   -7.6601e+18  -9.4433e+18
                                                -7.6459e+22  -9.4502e+22
------------------------------------------------------------------------
2.880e+04  4  1.038e+02   308
                                Solution         8.7021e+06   9.6500e+06
                                                 3.3804e+11   3.7510e+11
                                ----------------------------------------
                                Sensitivity 1   -5.3375e+22  -5.9187e+22
                                                 5.4487e+23   6.7430e+23
                                ----------------------------------------
                                Sensitivity 2   -4.8855e+18  -6.1040e+18
                                                -1.7194e+23  -2.1518e+23
------------------------------------------------------------------------
3.600e+04  4  7.257e+01   346
                                Solution         1.4040e+04   1.5609e+04
                                                 3.3868e+11   3.7652e+11
                                ----------------------------------------
                                Sensitivity 1   -8.6140e+19  -9.5761e+19
                                                 5.2718e+23   6.6029e+23
                                ----------------------------------------
                                Sensitivity 2   -8.4328e+15  -1.0549e+16
                                                -1.8439e+23  -2.3096e+23
------------------------------------------------------------------------
4.320e+04  4  3.835e+02   407
                                Solution        -5.2385e-08   5.9808e-07
                                                 3.3823e+11   3.8035e+11
                                ----------------------------------------
                                Sensitivity 1    8.8900e+08  -2.8682e+09
                                                 5.2753e+23   6.7448e+23
                                ----------------------------------------
                                Sensitivity 2    2.6162e+07   2.2623e+07
                                                -1.8454e+23  -2.3595e+23
------------------------------------------------------------------------
5.040e+04  5  4.386e+02   421
                                Solution         5.6769e-10   4.8955e-09
```

```
                                                3.3582e+11   3.8644e+11
                                  ---------------------------------------
                                  Sensitivity 1   -9.2603e+07  -1.0058e+08
                                                5.2067e+23   6.9664e+23
                                  ---------------------------------------
                                  Sensitivity 2   -2.8796e+07  -3.1171e+07
                                               -1.8214e+23  -2.4370e+23
-------------------------------------------------------------------------
5.760e+04  4  2.412e+02   435
                                  Solution         7.8795e-08   3.8016e-07
                                                3.3203e+11   3.9090e+11
                                  ---------------------------------------
                                  Sensitivity 1   -5.0500e+08  -2.4304e+09
                                                5.0825e+23   7.1205e+23
                                  ---------------------------------------
                                  Sensitivity 2    8.3512e+07   9.0527e+07
                                               -1.7780e+23  -2.4910e+23
-------------------------------------------------------------------------
6.480e+04  5  6.415e+02   451
                                  Solution         5.1990e-10   2.5692e-09
                                                3.3130e+11   3.9634e+11
                                  ---------------------------------------
                                  Sensitivity 1    1.1607e+07   6.1478e+07
                                                5.0442e+23   7.3273e+23
                                  ---------------------------------------
                                  Sensitivity 2   -1.8895e+07  -2.0261e+07
                                               -1.7646e+23  -2.5633e+23
-------------------------------------------------------------------------
7.200e+04  5  6.415e+02   462
                                  Solution        -5.3928e-11  -2.6841e-10
                                                3.3297e+11   4.0388e+11
                                  ---------------------------------------
                                  Sensitivity 1    1.1878e+06   6.0543e+06
                                                5.0783e+23   7.6382e+23
                                  ---------------------------------------
                                  Sensitivity 2   -7.4515e+05  -7.9928e+05
                                               -1.7765e+23  -2.6721e+23
-------------------------------------------------------------------------
7.920e+04  5  6.415e+02   473
                                  Solution        -5.6664e-13  -2.8119e-12
                                                3.3344e+11   4.1203e+11
                                  ---------------------------------------
                                  Sensitivity 1   -1.6520e+06  -8.6438e+06
                                                5.0730e+23   7.9960e+23
                                  ---------------------------------------
                                  Sensitivity 2    3.9882e+06   4.2797e+06
                                               -1.7747e+23  -2.7972e+23
-------------------------------------------------------------------------
8.640e+04  5  6.415e+02   485
                                  Solution        -4.0729e-15  -1.9951e-14
                                                3.3518e+11   4.1625e+11
                                  ---------------------------------------
                                  Sensitivity 1   -8.8716e+03  -4.6515e+04
                                                5.1171e+23   8.2142e+23
```

```
                                ----------------------------------------
                                Sensitivity 2    2.2251e+04   2.3887e+04
                                                -1.7901e+23  -2.8736e+23
------------------------------------------------------------------------

Final Statistics

nst     =   485

nfe     =  1109
netf    =    29    nsetups  =    83
nni     =   621    ncfn     =     0

nfSe    =  1226    nfeS     =  2452
netfs   =     0    nsetupsS =     0
nniS    =   612    ncfnS    =     0
```

# 3 Adjoint sensitivity analysis example problems

The next two sections describe in detail a serial example (`cvadx`) and a parallel one (`pvanx`). For details on the other examples, the reader is directed to the comments in their source files.

## 3.1 A serial dense example: `cvadx`

As a first example of using CVODES for adjoint sensitivity analysis we examine the chemical kinetics problem

$$\begin{aligned} \dot{y}_1 &= -p_1 y_1 + p_2 y_2 y_3 \\ \dot{y}_2 &= p_1 y_1 - p_2 y_2 y_3 - p_3 y_2^2 \\ \dot{y}_3 &= p_3 y_2^2 \\ y(t_0) &= y_0 \,, \end{aligned} \tag{11}$$

for which we want to compute the gradient with respect to $p$ of

$$G(p) = \int_{t_0}^{t_1} y_3 dt, \tag{12}$$

without having to compute the solution sensitivities $dy/dp$. Following the derivation in §3.3, and taking into account the fact that the initial values of (11) do not depend on the parameters $p$, by (3.18) this gradient is simply

$$\frac{dG}{dp} = \int_{t_0}^{t_1} \left( g_p + \lambda^T f_p \right) dt \,, \tag{13}$$

where $g(t, y, p) = y_3$, $f$ is the vector-valued function defining the right-hand side of (11), and $\lambda$ is the solution of the adjoint problem (3.17),

$$\begin{aligned} \dot{\lambda} &= -(f_y)^T \lambda - (g_y)^T \\ \lambda(t_1) &= 0 \,. \end{aligned} \tag{14}$$

In order to avoid saving intermediate $\lambda$ values just for the evaluation of the integral in (13), we extend the backward problem with the following $N_p$ quadrature equations

$$\begin{aligned} \dot{\xi} &= g_p^T + f_p^T \lambda \\ \xi(t_1) &= 0 \,, \end{aligned} \tag{15}$$

which yield $\xi(t_0) = -\int_{t_0}^{t_1} (g_p^T + f_p^T \lambda) dt$ and thus $dG/dp = -\xi^T(t_0)$. Similarly, the value of $G$ in (12) can be obtained as $G = -\zeta(t_0)$, where $\zeta$ is solution of the following quadrature equation:

$$\begin{aligned} \dot{\zeta} &= g \\ \zeta(t_1) &= 0 \,. \end{aligned} \tag{16}$$

The source code for this example is listed in App. D. The main program and the user-defined routines are described below, with emphasis on the aspects particular to adjoint sensitivity calculations.

The calling program includes the CVODES header files `cvodes.h` and `cvodea.h` for CVODES definitions and interface function prototypes, the header file `cvdense.h` for the CVDENSE linear solver module, the header file `nvector_serial.h` for the definition of the serial implementation of the NVECTOR module - NVECTOR_SERIAL, and the file `sundialsmath.h` for the definition of the `ABS` macro. This program also includes two user-defined accessor macros, `Ith` and `IJth` that are useful in writing the problem functions in a form closely matching their mathematical description, i.e. with components numbered from 1 instead of from 0. Following that, the program defines problem-specific constants and a user-defined data structure which will be used to pass the values of the parameters $p$ to various user routines. The constant `STEPS` defines the number of integration steps between two consecutive checkpoints. The program prologue ends with the prototypes of four user-supplied functions that are called by CVODES. The first two provide the right-hand side and dense Jacobian for the forward problem, and the last two provide the right-hand side and dense Jacobian for the backward problem.

The `main` function begins with type declarations and continues with the allocation and initialization of the user data structure which contains the values of the parameters $p$. Next, it allocates and initializes `y` with the initial conditions for the forward problem, allocates and initializes `q` for the quadrature used in computing the value $G$, and finally sets the scalar relative tolerance `reltolQ` and vector absolute tolerance `abstolQ` for the quadrature variable. No tolerances for the state variables are defined since `cvadx` uses its own function to compute the error weights for WRMS norm estimates of state solution vectors.

The call to `CVodeCreate` creates the main integrator memory block for the forward integration and specifies the `CV_BDF` integration method with `CV_NEWTON` iteration. The call to `CVodeMalloc` initializes the forward integration by specifying the initial conditions and that a function for error weights will be provided (`itol=CV_WF`. The next two calls specify the optional user data pointer and error weight calculation function. The linear solver is selected to be CVDENSE through the call to its initialization routine `CVDense`. The user provided Jacobian routine `Jac` and user data structure `data` are specified through a call to `CVDenseSetJacFn`.

The next code block initializes quadrature computations on the forward phase, by specifying the user data structure to be passed to the function `fQ`, including the quadrature variable in the error test, and setting the integration tolerances for the quadrature variable and finally allocating CVODES memory for quadrature integration (the call to `CVodeQuadMalloc` specifies the right-hand side of the quadrature equation and the initial values of the quadrature variable).

Allocation for the memory block of the combined forward-backward problem is acomplished through the call to `CVadjMalloc` which specifies `STEPS = 150`, the number of steps between two checkpoints.

The call to `CVodeF` requests the solution of the forward problem to `TOUT`. If successful, at the end of the integration, `CVodeF` will return the number of saved checkpoints in the argument `ncheck` (optionally, a list of the checkpoints can be printed by calling `CVadjGetCheckPointsList`).

The next segment of code deals with the setup of the backward problem. First, a serial vector `yB` of length `NEQ` is allocated and initalized with the value of $\lambda$ at the final time (0.0). A second serial vector `qB` of dimension `NP` is created and initialized to 0.0. This vector corresponds to the quadrature variables $\xi$ whose values at $t_0$ are the components of the gradient of $G$ with respect to the problem parameters $p$. Following that, the program sets the relative and absolute tolerances for the backward integration.

The CVODES memory for the integration of the backward integration is created and allocated by the calls to the interface routines `CVodeCreateB` amd `CVodeMallocB` which specify the `CV_BDF` integration method with `CV_NEWTON` iteration, among other things. The dense linear solver CVDENSE is then initialized by calling the `CVDenseB` interface routine and specifying a non-`NULL` Jacobian routine `JacB` and user data `data`.

The tolerances for the integration of quadrature variables, `reltolB` and `abstolQB`, are specified through `CVodeSetQuadTolerancesB`. The call to `CVodeSetQuadErrConB` indicates that $\xi$ should be included in the error test. Quadrature computation is initialized by calling `CVodeQuadMallocB` which specifies the right-hand side of the quadrature equations as `fQB`.

The actual solution of the backward problem is acomplished through the call to `CVodeB`. If successful, `CVodeB` returns the solution of the backward problem at time `T0` in the vector `yB`. The values of the quadrature variables at time `T0` are loaded in `qB` by calling the extraction routine `CVodeGetQuadB`. The values for $G$ and its gradient are printed next.

The main program continues with a call to `CVodeReInitB` and `CVodeQuadReInitB` to re-initialize the backward memory block for a new adjoint computation with a different final time (`TB2`), followed by a second call to `CVodeB` and, upon successful return, reporting of the new values for $G$ and its gradient.

The main program ends by freeing previously allocated memory by calling `CVodeFree` (for the CVODES memory for the forward problem), `CVadjFree` (for the memory allocated for the combined problem), and `N_VFree_Serial` (for the various vectors).

The user-supplied functions `f` and `Jac` for the right-hand side and Jacobian of the forward problem are straightforward expressions of its mathematical formulation (11). The function `ewt` is the same as the one for `cvdx.c`. The function `fQ` implements (16), while `fB`, `JacB`, and `fQB` are mere translations of the backward problem (14) and (15).

The output generated by `cvadx` is shown below.

cvadx sample output

```
 Adjoint Sensitivity Example for Chemical Kinetics
 -------------------------------------------------

ODE: dy1/dt = -p1*y1 + p2*y2*y3
     dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
     dy3/dt =  p3*(y2)^2

Find dG/dp for
     G = int_t0^tB0 g(t,p,y) dt
     g(t,p,y) = y3


Create and allocate CVODES memory for forward runs
Allocate global memory
Forward integration ... done. ncheck = 5   G:   3.9983e+07

Create and allocate CVODES memory for backward run
Integrate backwards
--------------------------------------------------------
tB0:           4.0000e+07
dG/dp:         7.6843e+05  -3.0691e+00   5.1150e-04
lambda(t0):    3.9967e+07   3.9967e+07   3.9967e+07
--------------------------------------------------------
```

```
Re-initialize CVODES memory for backward run
Integrate backwards
--------------------------------------------------------
tB0:          4.0000e+07
dG/dp:        1.7341e+02  -5.0590e-04   8.4320e-08
lambda(t0):   8.4190e+00   1.6097e+01   1.6097e+01
--------------------------------------------------------

Free memory
```

### 3.2 A parallel nonstiff example: pvanx

As an example of using the CVODES adjoint sensitivity module with the parallel vector module NVECTOR_PARALLEL, we describe a sample program that solves the following problem: consider the 1-D advection-diffusion equation

$$\begin{aligned}
&\frac{\partial u}{\partial t} = p_1 \frac{\partial^2 u}{\partial x^2} + p_2 \frac{\partial u}{\partial x} \\
&0 = x_0 \le x \le x_1 = 2 \\
&0 = t_0 \le t \le t_1 = 2.5\,,
\end{aligned} \tag{17}$$

with boundary conditions $u(t, x_0) = u(t, x_1) = 0$, $\forall t$ and initial condition $u(t_0, x) = u_0(x) = x(2-x)e^{2x}$. Also consider the function

$$g(t) = \int_{x_0}^{x_1} u(t, x) dx\,.$$

We wish to find, through adjoint sensitivity analysis, the gradient of $g(t_1)$ with respect to $p = [p_1; p_2]$ and the perturbation in $g(t_1)$ due to a perturbation $\delta u_0$ in $u_0$.

The approach we take in the program pvanx is to first derive an adjoint PDE which is then discretized in space and integrated backwards in time to yield the desired sensitivities. A straightforward extension to PDEs of the derivation given in §3.3 gives

$$\frac{dg}{dp}(t_1) = \int_{t_0}^{t_1} dt \int_{x_0}^{x_1} dx \mu \cdot \left[ \frac{\partial^2 u}{\partial x^2}; \frac{\partial u}{\partial x} \right] \tag{18}$$

and

$$\delta g|_{t_1} = \int_{x_0}^{x_1} \mu(t_0, x) \delta u_0(x) dx\,, \tag{19}$$

where $\mu$ is the solution of the adjoint PDE

$$\begin{aligned}
&\frac{\partial \mu}{\partial t} + p_1 \frac{\partial^2 \mu}{\partial x^2} - p_2 \frac{\partial \mu}{\partial x} = 0 \\
&\mu(t_1, x) = 1 \\
&\mu(t, x_0) = \mu(t, x_1) = 0\,.
\end{aligned} \tag{20}$$

Both the forward problem (17) and the backward problem (20) are discretized on a uniform spatial grid of size $M_x + 2$ with central differencing and with boundary values eliminated, leaving ODE systems of size $N = M_x$ each. As always, we deal with the time quadratures in (18) by introducing the additional equations

$$\begin{aligned}
\dot{\xi}_1 &= \int_{x_0}^{x_1} dx \mu \frac{\partial^2 u}{\partial x^2}\,, \quad \xi_1(t_1) = 0\,, \\
\dot{\xi}_2 &= \int_{x_0}^{x_1} dx \mu \frac{\partial u}{\partial x}\,, \quad \xi_2(t_1) = 0\,,
\end{aligned} \tag{21}$$

yielding

$$\frac{dg}{dp}(t_1) = [\xi_1(t_0); \xi_2(t_0)]$$

The space integrals in (19) and (21) are evaluated numerically, on the given spatial mesh, using the trapezoidal rule.

Note that $\mu(t_0, x^*)$ is nothing but the perturbation in $g(t_1)$ due to a perturbation $\delta u_0(x) = \delta(x - x^*)$ in the initial conditions. Therefore, $\mu(t_0, x)$ completely describes $\delta g(t_1)$ for any perturbation $\delta u_0$.

The source code for this example is listed in App. E. Both the forward and the backward problems are solved with the option for nonstiff systems, i.e. using the Adams method with functional iteration for the solution of the nonlinear systems. The overall structure of the `main` function is very similar to that of the code `cvadx` discussed previously with differences arising from the use of the parallel NVECTOR module. Unlike `cvadx`, the example `pvanx` illustrates computation of the additional quadrature variables by appending `NP` equations to the adjoint system. This approach can be a better alternative to using special treatment of the quadrature equations when their number is too small for parallel treatment.

Besides the parallelism implemented by CVODES at the NVECTOR level, `pvanx` uses MPI calls to parallelize the calculations of the right-hand side routines `f` and `fB` and of the spatial integrals involved. The forward problem has size `NEQ = MX`, while the backward problem has size `NB = NEQ + NP`, where `NP = 2` is the number of quadrature equations in (21). The use of the total number of available processes on two problems of different sizes deserves some comments, as this is typical in adjoint sensitivity analysis. Out of the total number of available processes, namely `nprocs`, the first `npes = nprocs - 1` processes are dedicated to the integration of the ODEs arising from the semi-discretization of the PDEs (17) and (20) and receive the same load on both the forward and backward integration phases. The last process is reserved for the integration of the quadrature equations (21), and is therefore inactive during the forward phases. Of course, for problems involving a much larger number of quadrature equations, more than one process could be reserved for their integration. An alternative would be to redistribute the `NB` backward problem variables over all available processes, without any relationship to the load distribution of the forward phase. However, the approach taken in `pvanx` has the advantage that the communication strategy adopted for the forward problem can be directly transfered to communication among the first `npes` processes during the backward integration phase.

We must also emphasize that, although inactive during the forward integration phase, the last process *must* participate in that phase with a *zero local array length*. This is because, during the backward integration phase, this process must have its own local copy of variables (such as `cvadj_mem`) that were set only during the forward phase.

Using `MX` $= 40$ on 4 processes, the gradient of $g(t_f)$ with respect to the two problem parameters is obtained as $dg/dp(t_f) = [-1.13856; -1.01023]$. The gradient of $g(t_f)$ with respect to the initial conditions is shown in Fig. 4. The gradient is plotted superimposed over the initial conditions. Sample output generated by `pvanx`, for `MX` $= 20$, is shown below.

pvanx sample output

```
(PE# 3) Number of check points: 6

g(tf) = 2.129919e-02

dgdp(tf)
  [ 1]: -1.129221e+00
  [ 2]: -1.008885e+00

mu(t0)
  [ 1]: 2.777306e-04
  [ 2]: 5.619708e-04
```

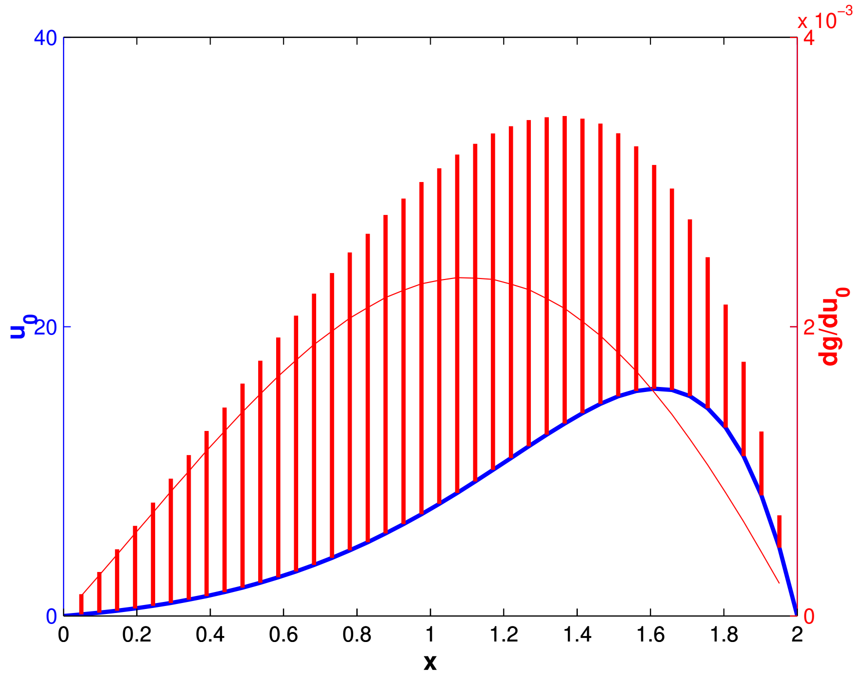


Figure 4: Results for the `pvanx` example problem. The gradient of $g(t_f)$ with respect to the initial conditions $u_0$ is shown superimposed over the values $u_0$.

```
[ 3]: 8.479539e-04
[ 4]: 1.126399e-03
[ 5]: 1.394128e-03
[ 6]: 1.639588e-03
[ 7]: 1.861653e-03
[ 8]: 2.047373e-03
[ 9]: 2.197987e-03
[10]: 2.300248e-03
[11]: 2.357877e-03
[12]: 2.358565e-03
[13]: 2.308409e-03
[14]: 2.197306e-03
[15]: 2.033385e-03
[16]: 1.809938e-03
[17]: 1.536549e-03
[18]: 1.210884e-03
[19]: 8.432127e-04
[20]: 4.362377e-04
```

### 3.3 An SPGMR parallel example using the CVBBDPRE module: pvakx

As a more elaborated adjoint sensitivity parallel example we describe next the pvakx code provided with CVODES. This example models an atmospheric release with an advection-diffusion PDE in 2-D or 3-D and computes the gradient with respect to source parameters of the space-time average of the squared norm of the concentration. Given a known velocity field $v(t, x)$, the transport equation for the cncentration $c(t, x)$ in a domain $\Omega$ is given by

$$\begin{aligned} \frac{\partial c}{\partial t} - k\delta c + v \cdot \nabla c + f = 0\,, \text{ in } (0,T) \times \Omega \\ \frac{\partial c}{\partial n} = g\,, \text{ on } (0,T) \times \partial\Omega \\ c = c_0(x)\,, \text{ in } \Omega \text{ at } t = 0\,, \end{aligned} \tag{22}$$

where $\Omega$ is a box in $\mathbb{R}^2$ or $\mathbb{R}^3$ and $n$ is the normal to the boundary of $\Omega$. We assume homogeneous boundary conditions ($g = 0$) and a zero initial concentration everywhere in $\Omega$ ($c_0(x) = 0$). The wind field has only a nonzero component in the $x$ direction given by a Poiseuille profile along the direction $y$.

Using adjoint sensitivity analysis, the gradient of

$$G(p) = \frac{1}{2} \int_0^T \int_\Omega \|c(t,x)\|^2 \, d\Omega \, dt \tag{23}$$

is obtained as

$$\frac{dG}{dp_i} = \int_t \int_\Omega \lambda(t,x)\delta(x_i) \, d\Omega \, dt = \int_t \lambda(t, x_i) \, dt\,, \tag{24}$$

where $x_i$ is the location of the source of intensity $p_i$ and $\lambda$ is solution of the adjoint PDE

$$\begin{aligned} -\frac{\partial \lambda}{\partial t} - k\Delta\lambda - v \cdot \lambda = c(t,x)\,, \text{ in } (T,0) \times \Omega \\ (k\nabla\lambda + v\lambda) \cdot n = 0\,, \text{ on } (0,T) \times \partial\Omega \\ \lambda = 0\,, \text{ in } \Omega \text{ at } t = T\,. \end{aligned} \tag{25}$$

The PDE (22) is semi-discretized in space with central finite differences, with the boundary conditions explicitely taken into account by using layers of ghost cells in every direction. If the direction $x^i$ of $\Omega$ is discretized into $m_i$ intervals, this leads to a system of ODEs of dimension $N = \prod_1^d (m_i + 1)$, with $d = 2$, or $d = 3$. The source term $f$ is parameterized as a piecewise constant function and yielding $N$ parameters in the problem. The nominal values of the source parameters corrspond to two Gaussian sources.

The adjoint PDE (25) is discretized to a system of ODEs in a similar fashion. The space integrals in (23) and (24) are simply approximated by their Riemann sums, while the time integrals are resolved by appending pure quadrature equations to the systems of ODEs.

The code for this example is listed in App. F. It uses BDF with the CVSPGMR linear solver and the CVBBDPRE preconditioner for both the forward and the backward integration phases. The value of $G$ is computed on the forward phase as a quadrature, while the components of the gradient $dG/dP$ are computed as quadratures during the backward integration phase. All quadrature variables are included in the corresponding error tests.

Communication between processes for the evaluation of the ODE right-hand sides involves passing the solution on the local boundaries (lines in 2-D, surfaces in 3-D) to the 4 (6 in 3-D) neighboring processes. This is implemented in the function `f_comm`, called in `f` and

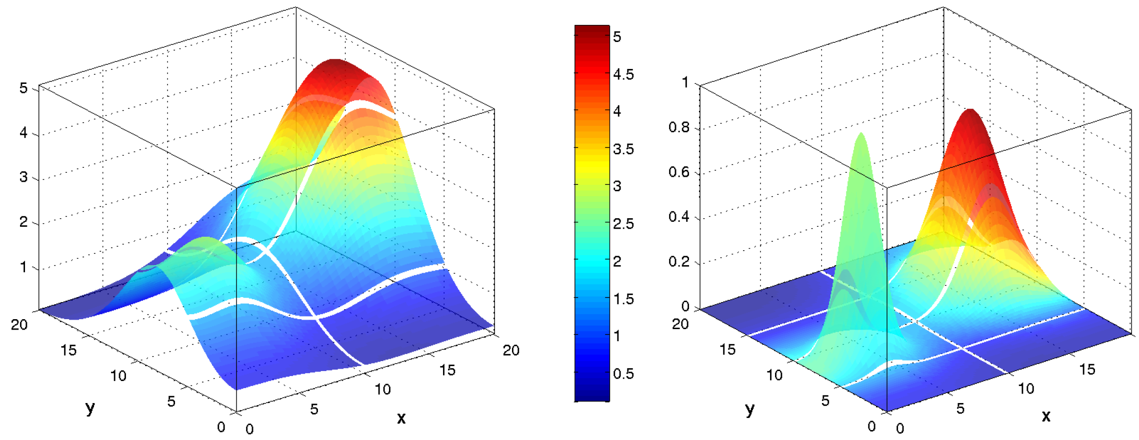


Figure 5: Results for the pvakx example problem in 2D. The gradient with respect to the source parameters is pictured on the left. On the right, the gradient was color coded and superimposed over the nominal value of the source parameters.

fB before evaluation of the local residual components. Since there is no additional communication required for the CVBBDPRE preconditioner, a NULL pointer is passed for gloc and glocB in the calls to CVBBSPrecAlloc and CVBBDPrecAllocB, respectivley.

For the sake of clarity, the pvakx example does not use the most memory-efficient implementation possible, as the local segment of the solution vectors (y on the forward phase and yB on the backward phase) and the data received from neighboring processes is loaded into a temporary array y_ext which is then used exclusively in computing the local components of the right-hand sides.

Note that if pvakx is given any command line argument, it will generate a series of MATLAB files which can be used to visualize the solution. Results for a 2-D simulation and adjoint sensitivity analysis with pvakx on a $80 \times 80$ grid and $2 \times 4 = 8$ processes are shown in Fig. 5. Results in 3-D [†], on a $80 \times 80 \times 40$ grid and $2 \times 4 \times 2 = 16$ processes are shown in Figs. 6 and 7. A sample output generated by pvakx for a 2D calculation is shown below.

pvakx sample output

```
Parallel Krylov adjoint sensitivity analysis example
2D Advection diffusion PDE with homogeneous Neumann B.C.
Computes gradient of G = int_t_Omega ( c_i^2 ) dt dOmega
with respect to the source values at each grid point.

Domain:
   0.000000 < x < 20.000000   mx = 20  npe_x = 2
   0.000000 < y < 20.000000   my = 40  npe_y = 2

Begin forward integration... done.   G = 3.723818e+03

Final Statistics..

lenrw    =    8746     leniw =    212
llrw     =    8656     lliw  =     80
```

[†] The name of executable for the 3-D version is pvakx3D.

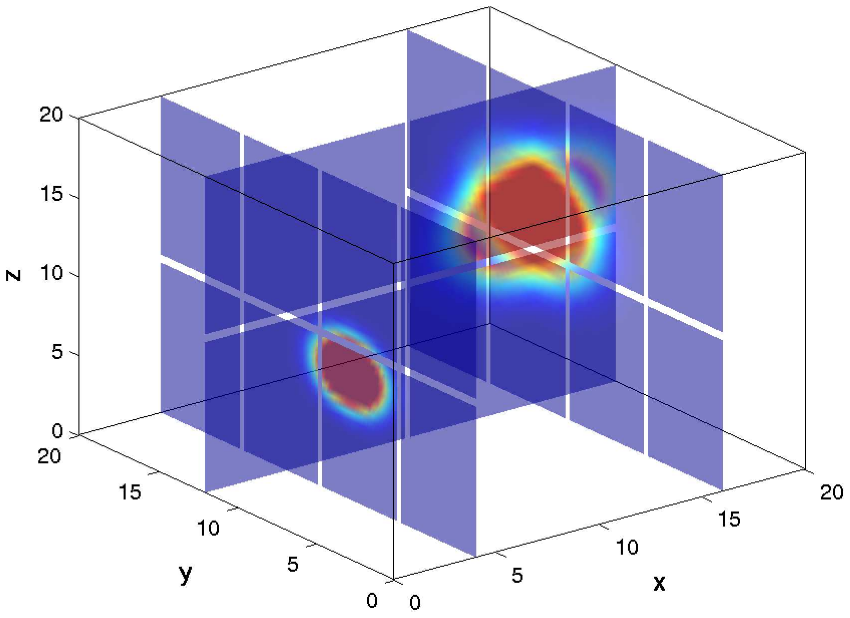


Figure 6: Results for the pvakx example problem in 3D. Nominal values of the source parameters.

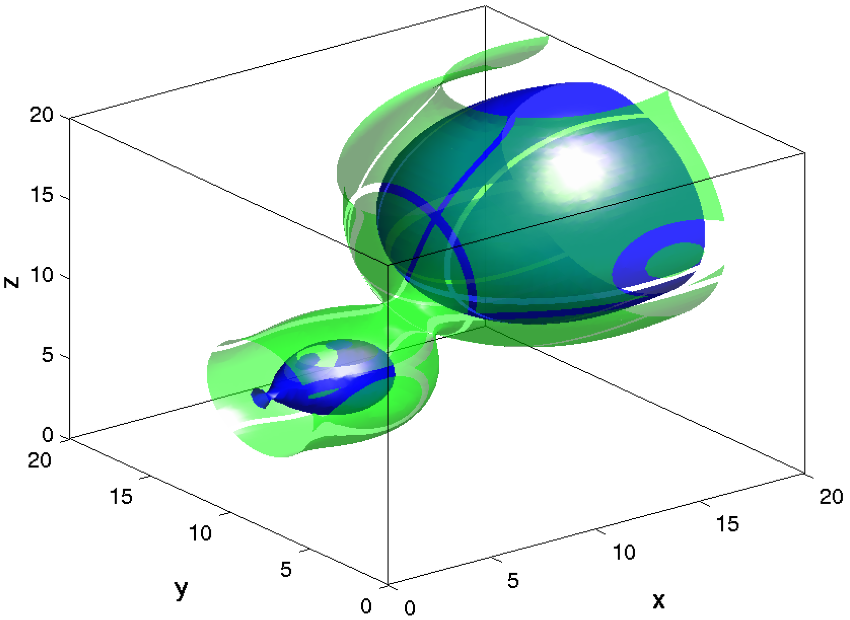


Figure 7: Results for the pvakx example problem in 3D. Two isosurfaces of the gradient with respect to the source parameters. They correspond to values of 0.25 (green) and 0.4 (blue).

```
nst     =    104
nfe     =    108     nfel  =    126
nni     =    105     nli   =    126
nsetups =     16     netf  =      0
npe     =      2     nps   =    215
ncfn    =      0     ncfl  =      0

Begin backward integration... done.

Final Statistics..

lenrw   =  17316     leniw =    212
llrw    =   8656     lliw  =     80
nst     =     78
nfe     =     90     nfel  =    138
nni     =     87     nli   =    138
nsetups =     17     netf  =      0
npe     =      2     nps   =    217
ncfn    =      0     ncfl  =      0
```

# 4 Parallel tests

The most preeminent advantage of CVODES over existing sensitivity solvers is the possibility of solving very large-scale problems on massively parallel computers. To illustrate this point we present speedup results for the integration and forward sensitivity analysis for an ODE system generated from the following 2-species diurnal kinetics advection-diffusion PDE system in 2 space dimensions. This work was reported in [3]. The PDE takes the form:

$$\frac{dc_i}{dt} = K_h \frac{d^2 c_i}{dx^2} + v \frac{dc_i}{dx} + K_v \frac{d^2 c_i}{dz^2} + R_i(c_1, c_2, t)\,, \quad \text{for } i = 1, 2\,,$$

where

$$\begin{aligned} R_1(c_1, c_2, t) &= -q_1 c_1 c_3 - q_2 c_1 c_2 + 2 q_3(t) c_3 + q_4(t) c_2, \\ R_2(c_1, c_2, t) &= q_1 c_1 c_3 - q_2 c_1 c_2 - q_4(t) c_2\,, \end{aligned}$$

$K_h$, $K_v$, $v$, $q_1$, $q_2$, and $c_3$ are constants, and $q_3(t)$ and $q_4(t)$ vary diurnally. The problem is posed on the square $0 \le x \le 20$, $30 \le z \le 50$ (all in km), with homogeneous Neumann boundary conditions, and for time t in $0 \le t \le 86400$ (1 day). The PDE system is treated by central differences on a uniform mesh, except for the advection term, which is treated with a biased 3-point difference formula. The initial profiles are proportional to a simple polynomial in $x$ and a hyperbolic tangent function in $z$.

The solution with CVODES is done with the BDF/GMRES method (i.e. using the CVSPGMR 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 preconditioner setup function.

The problem is solved by CVODES using $P$ processes, treated as a rectangular process grid of size $p_x \times p_z$. Each process is assigned a subgrid of size $n = n_x \times n_z$ of the $(x, z)$ mesh. Thus the actual mesh size is $N_x \times N_z = (p_x n_x) \times (p_z n_z)$, and the ODE system size is $N = 2 N_x N_z$. Parallel performance tests were performed on ASCI Frost, a 68-node, 16-way SMP system with POWER3 375 MHz processors and 16 GB of memory per node. We present timing results for the integration of only the state equations (column STATES), as well as for the computation of forward sensitivities with respect to the diffusion coefficients $K_h$ and $K_v$ using the staggered corrector method without and with error control on the sensitivity variables (columns STG and STG_FULL, respectively). Speedup results for a global problem size of $N = 2 N_x N_y = 2 \cdot 1600 \cdot 400 = 1280000$ shown in Fig. 8 and listed below.

| $P$ | STATES | STG | STG_FULL |
|---|---|---|---|
| 4 | 460.31 | 1414.53 | 2208.14 |
| 8 | 211.20 | 646.59 | 1064.94 |
| 16 | 97.16 | 320.78 | 417.95 |
| 32 | 42.78 | 137.51 | 210.84 |
| 64 | 19.50 | 63.34 | 83.24 |
| 128 | 13.78 | 42.71 | 55.17 |
| 256 | 9.87 | 31.33 | 47.95 |

We note that there was not enough memory to solve the problem (even without carrying sensitivities) using fewer processes.

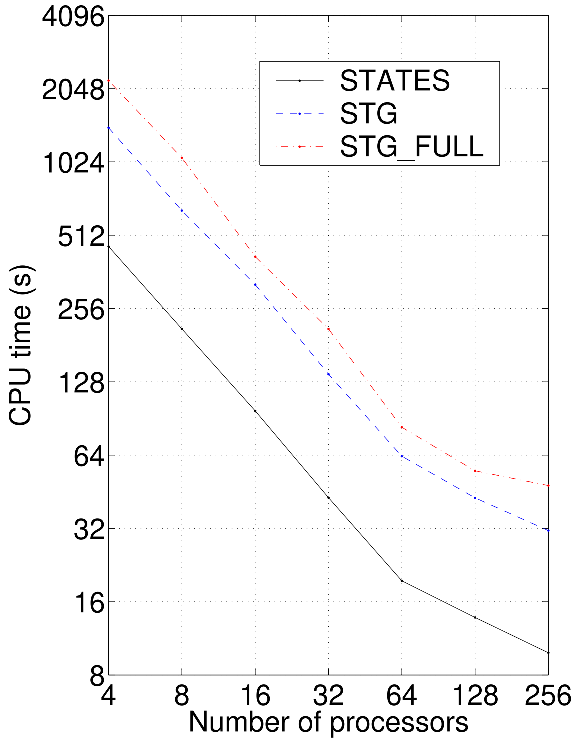


Figure 8: Speedup results for the integration of the state equations only (solid line and column 'STATES'), staggered sensitivity analysis without error control on the sensitivity variables (dashed line and column 'STG'), and staggered sensitivity analysis with full error control (dotted line and column 'STG_FULL')

The departure from the ideal line of slope $-1$ is explained by the interplay of several conflicting processes. On one hand, when increasing the number of processes, the preconditioner quality decreases, as it incorporates a smaller and smaller fraction of the Jacobian and the cost of interprocess communication increases. On the other hand, decreasing the number of processes leads to an increase in the cost of the preconditioner setup phase and to a larger local problem size which can lead to a point where a node starts memory paging to disk.

## References


[1] A. C. Hindmarsh and R. Serban. Example Programs for CVODE v2.2.0. Technical report, LLNL, 2004. UCRL-SM-208110.

[2] A. C. Hindmarsh and R. Serban. User Documentation for CVODES v2.1.0. Technical report, LLNL, 2004. UCRL-SM-208111.

[3] R. Serban and A. C. Hindmarsh. CVODES, an ODE solver with sensitivity analysis capabilities. Technical Report UCRL-TR-xxxxxx, LLNL, 2004.

# A Listing of cvfnx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.17.2.2 $
 * $Date: 2005/04/01 21:55:27 $
 * -----------------------------------------------------------------
 * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh, George D. Byrne,
 *                and Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * The following is a simple example problem, with the program for
 * its solution by CVODES. The problem is the semi-discrete form of
 * the advection-diffusion equation in 1-D:
 *   du/dt = q1 * d^2 u / dx^2 + q2 * du/dx
 * on the interval 0 <= x <= 2, and the time interval 0 <= t <= 5.
 * Homogeneous Dirichlet boundary conditions are posed, and the
 * initial condition is:
 *   u(x,y,t=0) = x(2-x)exp(2x).
 * The PDE is discretized on a uniform grid of size MX+2 with
 * central differencing, and with boundary values eliminated,
 * leaving an ODE system of size NEQ = MX.
 * This program solves the problem with the option for nonstiff
 * systems: ADAMS method and functional iteration.
 * It uses scalar relative and absolute tolerances.
 * Output is printed at t = .5, 1.0, ..., 5.
 * Run statistics (optional outputs) are printed at the end.
 *
 * Optionally, CVODES can compute sensitivities with respect to the
 * problem parameters q1 and q2.
 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
 * STAGGERED1) can be used and sensitivities may be included in the
 * error test or not (error control set on FULL or PARTIAL,
 * respectively).
 *
 * Execution:
 *
 * If no sensitivities are desired:
 *    % cvsnx -nosensi
 * If sensitivities are to be computed:
 *    % cvsnx -sensi sensi_meth err_con
 * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
 * {t, f}.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "sundialstypes.h"
#include "cvodes.h"
#include "nvector_serial.h"
```

```
/* Problem Constants */
#define XMAX  RCONST(2.0)   /* domain boundary           */
#define MX    10            /* mesh dimension            */
#define NEQ   MX            /* number of equations       */
#define ATOL  RCONST(1.e-5) /* scalar absolute tolerance */
#define T0    RCONST(0.0)   /* initial time              */
#define T1    RCONST(0.5)   /* first output time         */
#define DTOUT RCONST(0.5)   /* output time increment     */
#define NOUT  10            /* number of output times    */

#define NP    2
#define NS    2

#define ZERO  RCONST(0.0)

/* Type : UserData
   contains problem parameters, grid constants, work array. */

typedef struct {
  realtype *p;
  realtype dx;
} *UserData;

/* Functions Called by the CVODES Solver */

static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);

/* Private Helper Functions */

static void ProcessArgs(int argc, char *argv[],
                        booleantype *sensi, int *sensi_meth,
                        booleantype *err_con);
static void WrongArgs(char *name);
static void SetIC(N_Vector u, realtype dx);
static void PrintOutput(void *cvode_mem, realtype t, N_Vector u);
static void PrintOutputS(N_Vector *uS);
static void PrintFinalStats(void *cvode_mem, booleantype sensi);

static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  void *cvode_mem;
  UserData data;
  realtype dx, reltol, abstol, t, tout;
  N_Vector u;
  int iout, flag;
```

```
  realtype *pbar;
  int is, *plist;
  N_Vector *uS;
  booleantype sensi, err_con;
  int sensi_meth;

  cvode_mem = NULL;
  data = NULL;
  u = NULL;
  pbar = NULL;
  plist = NULL;
  uS = NULL;

  /* Process arguments */
  ProcessArgs(argc, argv, &sensi, &sensi_meth, &err_con);

  /* Set user data */
  data = (UserData) malloc(sizeof *data); /* Allocate data memory */
  if(check_flag((void *)data, "malloc", 2)) return(1);
  data->p = (realtype *) malloc(NP * sizeof(realtype));
  dx = data->dx = XMAX/((realtype)(MX+1));
  data->p[0] = RCONST(1.0);
  data->p[1] = RCONST(0.5);

  /* Allocate and set initial states */
  u = N_VNew_Serial(NEQ);
  if(check_flag((void *)u, "N_VNew_Serial", 0)) return(1);
  SetIC(u, dx);

  /* Set integration tolerances */
  reltol = ZERO;
  abstol = ATOL;

  /* Create CVODES object */
  cvode_mem = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
  if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

  flag = CVodeSetFdata(cvode_mem, data);
  if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);

  /* Allocate CVODES memory */
  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if(check_flag(&flag, "CVodeMalloc", 1)) return(1);

  printf("\n1-D advection-diffusion equation, mesh size =%3d\n", MX);

  /* Sensitivity-related settings */
  if(sensi) {

    plist = (int *) malloc(NS * sizeof(int));
    if(check_flag((void *)plist, "malloc", 2)) return(1);
    for(is=0; is<NS; is++) plist[is] = is+1;

```

```
    pbar  = (realtype *) malloc(NS * sizeof(realtype));
    if(check_flag((void *)pbar, "malloc", 2)) return(1);
    for(is=0; is<NS; is++) pbar[is] = data->p[plist[is]-1];

    uS = N_VNewVectorArray_Serial(NS, NEQ);
    if(check_flag((void *)uS, "N_VNew", 0)) return(1);
    for(is=0;is<NS;is++)
      N_VConst(ZERO, uS[is]);

    flag = CVodeSensMalloc(cvode_mem, NS, sensi_meth, uS);
    if(check_flag(&flag, "CVodeSensMalloc", 1)) return(1);

    flag = CVodeSetSensErrCon(cvode_mem, err_con);
    if(check_flag(&flag, "CVodeSetSensErrCon", 1)) return(1);

    flag = CVodeSetSensRho(cvode_mem, ZERO);
    if(check_flag(&flag, "CVodeSetSensRho", 1)) return(1);

    flag = CVodeSetSensParams(cvode_mem, data->p, pbar, plist);
    if(check_flag(&flag, "CVodeSetSensParams", 1)) return(1);

    printf("Sensitivity: YES ");
    if(sensi_meth == CV_SIMULTANEOUS)
      printf("( SIMULTANEOUS +");
    else
      if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
      else                           printf("( STAGGERED1 +");
    if(err_con) printf(" FULL ERROR CONTROL )");
    else        printf(" PARTIAL ERROR CONTROL )");

  } else {

    printf("Sensitivity: NO ");

  }

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

  printf("\n\n");
  printf("============================================================\n");
  printf("     T     Q       H      NST                    Max norm   \n");
  printf("============================================================\n");

  for (iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
    flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    if(check_flag(&flag, "CVode", 1)) break;
    PrintOutput(cvode_mem, t, u);
    if (sensi) {
      flag = CVodeGetSens(cvode_mem, t, uS);
      if(check_flag(&flag, "CVodeGetSens", 1)) break;
      PrintOutputS(uS);
    }
    printf("------------------------------------------------------------\n");
  }
```

```

  /* Print final statistics */
  PrintFinalStats(cvode_mem, sensi);

  /* Free memory */
  N_VDestroy_Serial(u);
  if (sensi) {
    N_VDestroyVectorArray_Serial(uS, NS);
    free(plist);
    free(pbar);
  }
  free(data);
  CVodeFree(cvode_mem);

  return(0);
}

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY CVODES
 *--------------------------------------------------------------------
 */

/*
 * f routine. Compute f(t,u).
 */

static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  realtype ui, ult, urt, hordc, horac, hdiff, hadv;
  realtype dx;
  realtype *udata, *dudata;
  int i;
  UserData data;

  udata = NV_DATA_S(u);
  dudata = NV_DATA_S(udot);

  /* Extract needed problem constants from data */
  data = (UserData) f_data;
  dx    = data->dx;
  hordc = data->p[0]/(dx*dx);
  horac = data->p[1]/(RCONST(2.0)*dx);

  /* Loop over all grid points. */
  for (i=0; i<NEQ; i++) {

    /* Extract u at x_i and two neighboring points */
    ui = udata[i];
    if(i!=0)
      ult = udata[i-1];
    else
      ult = ZERO;
    if(i!=NEQ-1)
```

```
      urt = udata[i+1];
    else
      urt = ZERO;

    /* Set diffusion and advection terms and load into udot */
    hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
    hadv = horac*(urt - ult);
    dudata[i] = hdiff + hadv;
  }
}

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * Process and verify arguments to cvfnx.
 */

static void ProcessArgs(int argc, char *argv[],
                        booleantype *sensi, int *sensi_meth, booleantype *err_con)
{
  *sensi = FALSE;
  *sensi_meth = -1;
  *err_con = FALSE;

  if (argc < 2) WrongArgs(argv[0]);

  if (strcmp(argv[1],"-nosensi") == 0)
    *sensi = FALSE;
  else if (strcmp(argv[1],"-sensi") == 0)
    *sensi = TRUE;
  else
    WrongArgs(argv[0]);

  if (*sensi) {

    if (argc != 4)
      WrongArgs(argv[0]);

    if (strcmp(argv[2],"sim") == 0)
      *sensi_meth = CV_SIMULTANEOUS;
    else if (strcmp(argv[2],"stg") == 0)
      *sensi_meth = CV_STAGGERED;
    else if (strcmp(argv[2],"stg1") == 0)
      *sensi_meth = CV_STAGGERED1;
    else
      WrongArgs(argv[0]);

    if (strcmp(argv[3],"t") == 0)
      *err_con = TRUE;
    else if (strcmp(argv[3],"f") == 0)
```

```
      *err_con = FALSE;
    else
      WrongArgs(argv[0]);
  }

}

static void WrongArgs(char *name)
{
    printf("\nUsage: %s [-nosensi] [-sensi sensi_meth err_con]\n",name);
    printf("         sensi_meth = sim, stg, or stg1\n");
    printf("         err_con    = t or f\n");

    exit(0);
}

/*
 * Set initial conditions in u vector.
 */

static void SetIC(N_Vector u, realtype dx)
{
  int i;
  realtype x;
  realtype *udata;

  /* Set pointer to data array and get local length of u. */
  udata = NV_DATA_S(u);

  /* Load initial profile into u vector */
  for (i=0; i<NEQ; i++) {
    x = (i+1)*dx;
    udata[i] = x*(XMAX - x)*exp(RCONST(2.0)*x);
  }
}

/*
 * Print current t, step count, order, stepsize, and max norm of solution
 */

static void PrintOutput(void *cvode_mem, realtype t, N_Vector u)
{
  long int nst;
  int qu, flag;
  realtype hu;

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetLastOrder(cvode_mem, &qu);
  check_flag(&flag, "CVodeGetLastOrder", 1);
  flag = CVodeGetLastStep(cvode_mem, &hu);
  check_flag(&flag, "CVodeGetLastStep", 1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
```

```
  printf("%8.3Le %2d  %8.3Le %5ld\n", t, qu, hu ,nst);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%8.3le %2d  %8.3le %5ld\n", t, qu, hu ,nst);
#else
  printf("%8.3e %2d  %8.3e %5ld\n", t, qu, hu ,nst);
#endif

  printf("                                Solution       ");

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le \n", N_VMaxNorm(u));
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le \n", N_VMaxNorm(u));
#else
  printf("%12.4e \n", N_VMaxNorm(u));
#endif
}

/*
 * Print max norm of sensitivities
 */

static void PrintOutputS(N_Vector *uS)
{
  printf("                                Sensitivity 1  ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le \n", N_VMaxNorm(uS[0]));
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le \n", N_VMaxNorm(uS[0]));
#else
  printf("%12.4e \n", N_VMaxNorm(uS[0]));
#endif

  printf("                                Sensitivity 2  ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le \n", N_VMaxNorm(uS[1]));
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le \n", N_VMaxNorm(uS[1]));
#else
  printf("%12.4e \n", N_VMaxNorm(uS[1]));
#endif
}


/*
 * Print some final statistics located in the CVODES memory
 */

static void PrintFinalStats(void *cvode_mem, booleantype sensi)
{
  long int nst;
  long int nfe, nsetups, nni, ncfn, netf;
  long int nfSe, nfeS, nsetupsS, nniS, ncfnS, netfS;
  int flag;
```

```
  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);

  if (sensi) {
    flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
    check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
    flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
    check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
    flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
    check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
    flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
    check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);
    flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
    check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1);
    flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
    check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1);
  }

  printf("\nFinal Statistics\n\n");
  printf("nst     = %5ld\n\n", nst);
  printf("nfe     = %5ld\n",   nfe);
  printf("netf    = %5ld    nsetups  = %5ld\n", netf, nsetups);
  printf("nni     = %5ld    ncfn     = %5ld\n", nni, ncfn);

  if(sensi) {
    printf("\n");
    printf("nfSe    = %5ld    nfeS     = %5ld\n", nfSe, nfeS);
    printf("netfs   = %5ld    nsetupsS = %5ld\n", netfS, nsetupsS);
    printf("nniS    = %5ld    ncfnS    = %5ld\n", nniS, ncfnS);
  }

}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

```

```
static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr,
            "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr,
            "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  return(0);
}
```

# B Listing of `cvfdx.c`

```c
/*
 * -----------------------------------------------------------------
 * $Revision: 1.21.2.4 $
 * $Date: 2005/04/07 15:58:52 $
 * -----------------------------------------------------------------
 * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh, and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * The following is a simple example problem, with the coding
 * needed for its solution by CVODES. The problem is from chemical
 * kinetics, and consists of the following three rate equations:
 *    dy1/dt = -p1*y1 + p2*y2*y3
 *    dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
 *    dy3/dt =  p3*(y2)^2
 * on the interval from t = 0.0 to t = 4.e10, with initial
 * conditions y1 = 1.0, y2 = y3 = 0. The reaction rates are: p1=0.04,
 * p2=1e4, and p3=3e7. The problem is stiff.
 * This program solves the problem with the BDF method, Newton
 * iteration with the CVODES dense linear solver, and a
 * user-supplied Jacobian routine.
 * It uses a scalar relative tolerance and a vector absolute
 * tolerance.
 * Output is printed in decades from t = .4 to t = 4.e10.
 * Run statistics (optional outputs) are printed at the end.
 *
 * Optionally, CVODES can compute sensitivities with respect to the
 * problem parameters p1, p2, and p3.
 * The sensitivity right hand side is given analytically through the
 * user routine fS (of type SensRhs1Fn).
 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
 * STAGGERED1) can be used and sensitivities may be included in the
 * error test or not (error control set on TRUE or FALSE,
 * respectively).
 *
 * Execution:
 *
 * If no sensitivities are desired:
 *    % cvsdx -nosensi
 * If sensitivities are to be computed:
 *    % cvsdx -sensi sensi_meth err_con
 * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
 * {t, f}.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sundialstypes.h"   /* def. of type realtype                       */
#include "cvodes.h"          /* prototypes for CVODES functions and constants */
```

```
#include "cvdense.h"         /* prototype for CVDENSE functions and constants */
#include "nvector_serial.h"  /* defs. of serial NVECTOR functions and macros  */
#include "dense.h"           /* defs. of type DenseMat, macro DENSE_ELEM      */
#include "sundialsmath.h"    /* definition of ABS                             */

/* Accessor macros */

#define Ith(v,i)    NV_Ith_S(v,i-1)       /* i-th vector component i=1..NEQ */
#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j=1..NEQ */

/* Problem Constants */

#define NEQ   3             /* number of equations  */
#define Y1    RCONST(1.0)   /* initial y components */
#define Y2    RCONST(0.0)
#define Y3    RCONST(0.0)
#define RTOL  RCONST(1e-4)  /* scalar relative tolerance */
#define ATOL1 RCONST(1e-8)  /* vector absolute tolerance components */
#define ATOL2 RCONST(1e-14)
#define ATOL3 RCONST(1e-6)
#define T0    RCONST(0.0)   /* initial time */
#define T1    RCONST(0.4)   /* first output time */
#define TMULT RCONST(10.0)  /* output time factor */
#define NOUT  12            /* number of output times */

#define NP    3             /* number of problem parameters */
#define NS    3             /* number of sensitivities computed */

#define ZERO  RCONST(0.0)

/* Type : UserData */

typedef struct {
  realtype p[3];           /* problem parameters */
} *UserData;

/* Prototypes of functions by CVODES */

static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);

static void Jac(long int N, DenseMat J, realtype t,
                N_Vector y, N_Vector fy, void *jac_data,
                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

static void fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
               int iS, N_Vector yS, N_Vector ySdot,
               void *fS_data, N_Vector tmp1, N_Vector tmp2);

static int ewt(N_Vector y, N_Vector w, void *e_data);

/* Prototypes of private functions */

static void ProcessArgs(int argc, char *argv[],
                        booleantype *sensi, int *sensi_meth,
```

```
                        booleantype *err_con);
static void WrongArgs(char *name);
static void PrintOutput(void *cvode_mem, realtype t, N_Vector u);
static void PrintOutputS(N_Vector *uS);
static void PrintFinalStats(void *cvode_mem, booleantype sensi);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  void *cvode_mem;
  UserData data;
  realtype t, tout;
  N_Vector y;
  int iout, flag;

  realtype pbar[NS];
  int is;
  N_Vector *yS;
  booleantype sensi, err_con;
  int sensi_meth;

  cvode_mem = NULL;
  data      = NULL;
  y         =  NULL;
  yS        = NULL;

  /* Process arguments */
  ProcessArgs(argc, argv, &sensi, &sensi_meth, &err_con);

  /* User data structure */
  data = (UserData) malloc(sizeof *data);
  if (check_flag((void *)data, "malloc", 2)) return(1);
  data->p[0] = RCONST(0.04);
  data->p[1] = RCONST(1.0e4);
  data->p[2] = RCONST(3.0e7);

  /* Initial conditions */
  y = N_VNew_Serial(NEQ);
  if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);

  Ith(y,1) = Y1;
  Ith(y,2) = Y2;
  Ith(y,3) = Y3;

  /* Create CVODES object */
  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  if (check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

```

```
  /* Allocate space for CVODES */
  flag = CVodeMalloc(cvode_mem, f, T0, y, CV_WF, 0.0, NULL);
  if (check_flag(&flag, "CVodeMalloc", 1)) return(1);

  /* Use private function to compute error weights */
  flag = CVodeSetEwtFn(cvode_mem, ewt, NULL);
  if (check_flag(&flag, "CVodeSetEwtFn", 1)) return(1);

  /* Attach user data */
  flag = CVodeSetFdata(cvode_mem, data);
  if (check_flag(&flag, "CVodeSetFdata", 1)) return(1);

  /* Attach linear solver */
  flag = CVDense(cvode_mem, NEQ);
  if (check_flag(&flag, "CVDense", 1)) return(1);

  flag = CVDenseSetJacFn(cvode_mem, Jac, data);
  if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);

  printf("\n3-species chemical kinetics problem\n");

  /* Sensitivity-related settings */
  if (sensi) {

    pbar[0] = data->p[0];
    pbar[1] = data->p[1];
    pbar[2] = data->p[2];

    yS = N_VNewVectorArray_Serial(NS, NEQ);
    if (check_flag((void *)yS, "N_VNewVectorArray_Serial", 0)) return(1);
    for (is=0;is<NS;is++) N_VConst(ZERO, yS[is]);

    flag = CVodeSensMalloc(cvode_mem, NS, sensi_meth, yS);
    if(check_flag(&flag, "CVodeSensMalloc", 1)) return(1);

    flag = CVodeSetSensRhs1Fn(cvode_mem, fS);
    if (check_flag(&flag, "CVodeSetSensRhs1Fn", 1)) return(1);
    flag = CVodeSetSensErrCon(cvode_mem, err_con);
    if (check_flag(&flag, "CVodeSetSensFdata", 1)) return(1);
    flag = CVodeSetSensFdata(cvode_mem, data);
    if (check_flag(&flag, "CVodeSetSensFdata", 1)) return(1);
    flag = CVodeSetSensParams(cvode_mem, NULL, pbar, NULL);
    if (check_flag(&flag, "CVodeSetSensParams", 1)) return(1);

    printf("Sensitivity: YES ");
    if(sensi_meth == CV_SIMULTANEOUS)
      printf("( SIMULTANEOUS +");
    else
      if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
      else                           printf("( STAGGERED1 +");
    if(err_con) printf(" FULL ERROR CONTROL )");
    else        printf(" PARTIAL ERROR CONTROL )");

  } else {
```

```c

    printf("Sensitivity: NO ");

  }

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

  printf("\n\n");
  printf("===================================================");
  printf("============================\n");
  printf("     T     Q       H      NST                    y1");
  printf("           y2           y3    \n");
  printf("===================================================");
  printf("============================\n");

  for (iout=1, tout=T1; iout <= NOUT; iout++, tout *= TMULT) {

    flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
    if (check_flag(&flag, "CVode", 1)) break;

    PrintOutput(cvode_mem, t, y);

    if (sensi) {
      flag = CVodeGetSens(cvode_mem, t, yS);
      if (check_flag(&flag, "CVodeGetSens", 1)) break;
      PrintOutputS(yS);
    }
    printf("-------------------------------------------------");
    printf("------------------------------\n");

  }

  /* Print final statistics */
  PrintFinalStats(cvode_mem, sensi);

  /* Free memory */

  N_VDestroy_Serial(y);                    /* Free y vector */
  if (sensi) {
    N_VDestroyVectorArray_Serial(yS, NS);  /* Free yS vector */
  }
  free(data);                              /* Free user data */
  CVodeFree(cvode_mem);                    /* Free CVODES memory */

  return(0);
}

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY CVODES
 *--------------------------------------------------------------------
 */

/*
```

```
 * f routine. Compute f(t,y).
 */

static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
{
  realtype y1, y2, y3, yd1, yd3;
  UserData data;
  realtype p1, p2, p3;

  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
  data = (UserData) f_data;
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
  yd3 = Ith(ydot,3) = p3*y2*y2;
        Ith(ydot,2) = -yd1 - yd3;
}


/*
 * Jacobian routine. Compute J(t,y).
 */

static void Jac(long int N, DenseMat J, realtype t,
                N_Vector y, N_Vector fy, void *jac_data,
                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  realtype y1, y2, y3;
  UserData data;
  realtype p1, p2, p3;

  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
  data = (UserData) jac_data;
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  IJth(J,1,1) = -p1;  IJth(J,1,2) = p2*y3;          IJth(J,1,3) = p2*y2;
  IJth(J,2,1) =  p1;  IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
                      IJth(J,3,2) = 2*p3*y2;
}

/*
 * fS routine. Compute sensitivity r.h.s.
 */

static void fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
               int iS, N_Vector yS, N_Vector ySdot,
               void *fS_data, N_Vector tmp1, N_Vector tmp2)
{
  UserData data;
  realtype p1, p2, p3;
  realtype y1, y2, y3;
  realtype s1, s2, s3;
  realtype sd1, sd2, sd3;

```

```
  data = (UserData) fS_data;
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  y1 = Ith(y,1);  y2 = Ith(y,2);  y3 = Ith(y,3);
  s1 = Ith(yS,1); s2 = Ith(yS,2); s3 = Ith(yS,3);

  sd1 = -p1*s1 + p2*y3*s2 + p2*y2*s3;
  sd3 = 2*p3*y2*s2;
  sd2 = -sd1-sd3;

  switch (iS) {
  case 0:
    sd1 += -y1;
    sd2 +=  y1;
    break;
  case 1:
    sd1 +=  y2*y3;
    sd2 += -y2*y3;
    break;
  case 2:
    sd2 += -y2*y2;
    sd3 +=  y2*y2;
    break;
  }

  Ith(ySdot,1) = sd1;
  Ith(ySdot,2) = sd2;
  Ith(ySdot,3) = sd3;
}

/*
 * EwtSet function. Computes the error weights at the current solution.
 */

static int ewt(N_Vector y, N_Vector w, void *e_data)
{
  int i;
  realtype yy, ww, rtol, atol[3];

  rtol    = RTOL;
  atol[0] = ATOL1;
  atol[1] = ATOL2;
  atol[2] = ATOL3;

  for (i=1; i<=3; i++) {
    yy = Ith(y,i);
    ww = rtol * ABS(yy) + atol[i-1];
    if (ww <= 0.0) return (-1);
    Ith(w,i) = 1.0/ww;
  }

  return(0);
}

```

```
/*
 *-----------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *-----------------------------------------------------------------
 */

/*
 * Process and verify arguments to cvfdx.
 */

static void ProcessArgs(int argc, char *argv[],
                        booleantype *sensi, int *sensi_meth, booleantype *err_con)
{
  *sensi = FALSE;
  *sensi_meth = -1;
  *err_con = FALSE;

  if (argc < 2) WrongArgs(argv[0]);

  if (strcmp(argv[1],"-nosensi") == 0)
    *sensi = FALSE;
  else if (strcmp(argv[1],"-sensi") == 0)
    *sensi = TRUE;
  else
    WrongArgs(argv[0]);

  if (*sensi) {

    if (argc != 4)
      WrongArgs(argv[0]);

    if (strcmp(argv[2],"sim") == 0)
      *sensi_meth = CV_SIMULTANEOUS;
    else if (strcmp(argv[2],"stg") == 0)
      *sensi_meth = CV_STAGGERED;
    else if (strcmp(argv[2],"stg1") == 0)
      *sensi_meth = CV_STAGGERED1;
    else
      WrongArgs(argv[0]);

    if (strcmp(argv[3],"t") == 0)
      *err_con = TRUE;
    else if (strcmp(argv[3],"f") == 0)
      *err_con = FALSE;
    else
      WrongArgs(argv[0]);
  }

}

static void WrongArgs(char *name)
{
    printf("\nUsage: %s [-nosensi] [-sensi sensi_meth err_con]\n",name);
    printf("         sensi_meth = sim, stg, or stg1\n");
```

```
    printf("         err_con    = t or f\n");

    exit(0);
}

/*
 * Print current t, step count, order, stepsize, and solution.
 */

static void PrintOutput(void *cvode_mem, realtype t, N_Vector u)
{
  long int nst;
  int qu, flag;
  realtype hu, *udata;

  udata = NV_DATA_S(u);

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetLastOrder(cvode_mem, &qu);
  check_flag(&flag, "CVodeGetLastOrder", 1);
  flag = CVodeGetLastStep(cvode_mem, &hu);
  check_flag(&flag, "CVodeGetLastStep", 1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%8.3Le %2d  %8.3Le %5ld\n", t, qu, hu, nst);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%8.3le %2d  %8.3le %5ld\n", t, qu, hu, nst);
#else
  printf("%8.3e %2d  %8.3e %5ld\n", t, qu, hu, nst);
#endif

  printf("                          Solution       ");

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le %12.4Le %12.4Le \n", udata[0], udata[1], udata[2]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le %12.4le %12.4le \n", udata[0], udata[1], udata[2]);
#else
  printf("%12.4e %12.4e %12.4e \n", udata[0], udata[1], udata[2]);
#endif

}

/*
 * Print sensitivities.
*/

static void PrintOutputS(N_Vector *uS)
{
  realtype *sdata;

  sdata = NV_DATA_S(uS[0]);
  printf("                          Sensitivity 1  ");
```

```
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le %12.4Le %12.4Le \n", sdata[0], sdata[1], sdata[2]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le %12.4le %12.4le \n", sdata[0], sdata[1], sdata[2]);
#else
  printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
#endif

  sdata = NV_DATA_S(uS[1]);
  printf("                          Sensitivity 2  ");

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le %12.4Le %12.4Le \n", sdata[0], sdata[1], sdata[2]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le %12.4le %12.4le \n", sdata[0], sdata[1], sdata[2]);
#else
  printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
#endif

  sdata = NV_DATA_S(uS[2]);
  printf("                          Sensitivity 3  ");

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%12.4Le %12.4Le %12.4Le \n", sdata[0], sdata[1], sdata[2]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%12.4le %12.4le %12.4le \n", sdata[0], sdata[1], sdata[2]);
#else
  printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
#endif
}

/*
 * Print some final statistics from the CVODES memory.
 */

static void PrintFinalStats(void *cvode_mem, booleantype sensi)
{
  long int nst;
  long int nfe, nsetups, nni, ncfn, netf;
  long int nfSe, nfeS, nsetupsS, nniS, ncfnS, netfS;
  long int njeD, nfeD;
  int flag;

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
```

```c
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);

  if (sensi) {
    flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
    check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
    flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
    check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
    flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
    check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
    flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
    check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);
    flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
    check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1);
    flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
    check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1);
  }

  flag = CVDenseGetNumJacEvals(cvode_mem, &njeD);
  check_flag(&flag, "CVDenseGetNumJacEvals", 1);
  flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeD);
  check_flag(&flag, "CVDenseGetNumRhsEvals", 1);

  printf("\nFinal Statistics\n\n");
  printf("nst     = %5ld\n\n", nst);
  printf("nfe     = %5ld\n",   nfe);
  printf("netf    = %5ld    nsetups  = %5ld\n", netf, nsetups);
  printf("nni     = %5ld    ncfn     = %5ld\n", nni, ncfn);

  if(sensi) {
    printf("\n");
    printf("nfSe    = %5ld    nfeS     = %5ld\n", nfSe, nfeS);
    printf("netfs   = %5ld    nsetupsS = %5ld\n", netfS, nsetupsS);
    printf("nniS    = %5ld    ncfnS    = %5ld\n", nniS, ncfnS);
  }

  printf("\n");
  printf("njeD    = %5ld    nfeD     = %5ld\n", njeD, nfeD);

}

/*
 * Check function return value.
 *    opt == 0 means SUNDIALS function allocates memory so check if
 *             returned NULL pointer
 *    opt == 1 means SUNDIALS function returns a flag so check if
 *             flag >= 0
 *    opt == 2 means function allocates memory so check if returned
 *             NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;
```

```

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr,
            "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr,
            "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  return(0);
}
```

## C Listing of `pvfkx.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.20.2.3 $
 * $Date: 2005/04/06 23:34:05 $
 * -----------------------------------------------------------------
 * Programmer(s): S. D. Cohen, A. C. Hindmarsh, Radu Serban,
 *                and M. R. Wittman @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * 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 CVODES 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 neq = 2*MX*MY.
 *
 * The solution with CVODES is done with the BDF/GMRES method (i.e.
 * using the CVSPGMR 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 values are printed at
 * selected output times, and all performance counters are printed
 * on completion.
 *
 * Optionally, CVODES can compute sensitivities with respect to the
 * problem parameters q1 and q2.
 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
 * STAGGERED1) can be used and sensitivities may be included in the
 * error test or not (error control set on FULL or PARTIAL,
 * respectively).
 *
 * Execution:
 *
```

```
 * Note: This version uses MPI for user routines, and the CVODES
 *       solver. In what follows, N is the number of processors,
 *       N = NPEX*NPEY (see constants below) and it is assumed that
 *       the MPI script mpirun is used to run a paralles
 *       application.
 * If no sensitivities are desired:
 *    % mpirun -np N pvfkx -nosensi
 * If sensitivities are to be computed:
 *    % mpirun -np N pvfkx -sensi sensi_meth err_con
 * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
 * {t, f}.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "sundialstypes.h"    /* def. of realtype                             */
#include "cvodes.h"           /* main CVODES header file                      */
#include "iterative.h"        /* types of preconditioning                     */
#include "cvspgmr.h"          /* defs. for CVSPGMR functions and constants    */
#include "smalldense.h"       /* generic DENSE solver used in preconditioning */
#include "nvector_parallel.h" /* defs of paralel NVECTOR functions and macros */
#include "sundialsmath.h"     /* contains SQR macro                           */
#include "mpi.h"


/* Problem Constants */

#define NVARS     2              /* number of species                    */
#define C1_SCALE  RCONST(1.0e6)  /* coefficients in initial profiles     */
#define C2_SCALE  RCONST(1.0e12)

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

#define XMIN      RCONST(0.0)    /* grid boundaries in x                 */
#define XMAX      RCONST(20.0)
#define YMIN      RCONST(30.0)   /* grid boundaries in y                 */
#define YMAX      RCONST(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             */
```

```
/* CVodeMalloc Constants */

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

/* Sensitivity constants */
#define NP        8              /* number of problem parameters           */
#define NS        2              /* number of sensitivities                */

#define ZERO      RCONST(0.0)


/* 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 realtype **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])

/* Types : UserData and PreconData
   contain problem parameters, problem constants, preconditioner blocks,
   pivot arrays, grid constants, and processor indices */

typedef struct {
  realtype *p;
  realtype q4, om, dx, dy, hdco, haco, vdco;
  realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
  long int my_pe, isubx, isuby, nvmxsub, nvmxsub2;
  MPI_Comm comm;
} *UserData;

typedef struct {
  void *f_data;
  realtype **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
  long int *pivot[MXSUB][MYSUB];
} *PreconData;


/* Functions Called by the CVODES Solver */

static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);

static int Precond(realtype tn, N_Vector u, N_Vector fu,
                   booleantype jok, booleantype *jcurPtr,
                   realtype gamma, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

```
static int PSolve(realtype tn, N_Vector u, N_Vector fu,
                  N_Vector r, N_Vector z,
                  realtype gamma, realtype delta,
                  int lr, void *P_data, N_Vector vtemp);

/* Private Helper Functions */

static void ProcessArgs(int argc, char *argv[], int my_pe,
                        booleantype *sensi, int *sensi_meth, booleantype *err_con);
static void WrongArgs(int my_pe, char *name);

static PreconData AllocPreconData(UserData data);
static void FreePreconData(PreconData pdata);
static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
static void SetInitialProfiles(N_Vector u, UserData data);

static void BSend(MPI_Comm comm, int my_pe, long int isubx,
                  long int isuby, long int dsizex,
                  long int dsizey, realtype udata[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[], int my_pe,
                      long int isubx, long int isuby,
                      long int dsizex, long int dsizey,
                      realtype uext[], realtype buffer[]);
static void BRecvWait(MPI_Request request[], long int isubx, long int isuby,
                      long int dsizex, realtype uext[], realtype buffer[]);
static void ucomm(realtype t, N_Vector u, UserData data);
static void fcalc(realtype t, realtype udata[], realtype dudata[], UserData data);

static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
                        realtype t, N_Vector u);
static void PrintOutputS(int my_pe, MPI_Comm comm, N_Vector *uS);
static void PrintFinalStats(void *cvode_mem, booleantype sensi);
static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  realtype abstol, reltol, t, tout;
  N_Vector u;
  UserData data;
  PreconData predata;
  void *cvode_mem;
  int iout, flag, my_pe, npes;
  long int neq, local_N;
  MPI_Comm comm;

  realtype *pbar;
  int is, *plist;
```

```
  N_Vector *uS;
  booleantype sensi, err_con;
  int sensi_meth;

  u = NULL;
  data = NULL;
  predata = NULL;
  cvode_mem = NULL;
  pbar = NULL;
  plist = NULL;
  uS = NULL;

  /* Set problem size neq */
  neq = NVARS*MX*MY;

  /* 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)
      fprintf(stderr,
              "\nMPI_ERROR(0): npes = %d is not equal to NPEX*NPEY = %d\n\n",
              npes, NPEX*NPEY);
    MPI_Finalize();
    return(1);
  }

  /* Process arguments */
  ProcessArgs(argc, argv, my_pe, &sensi, &sensi_meth, &err_con);

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

  /* Allocate and load user data block; allocate preconditioner block */
  data = (UserData) malloc(sizeof *data);
  data->p = NULL;
  if (check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
  data->p = (realtype *) malloc(NP*sizeof(realtype));
  if (check_flag((void *)data->p, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
  InitUserData(my_pe, comm, data);
  predata = AllocPreconData (data);
  if (check_flag((void *)predata, "AllocPreconData", 2, my_pe)) MPI_Abort(comm, 1);

  /* Allocate u, and set initial values and tolerances */
  u = N_VNew_Parallel(comm, local_N, neq);
  if (check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  SetInitialProfiles(u, data);
  abstol = ATOL; reltol = RTOL;

  /* Create CVODES object, set optional input, allocate memory */
  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
```

```
  if (check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);

  flag = CVodeSetFdata(cvode_mem, data);
  if (check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);

  flag = CVodeSetMaxNumSteps(cvode_mem, 2000);
  if (check_flag(&flag, "CVodeSetMaxNumSteps", 1, my_pe)) MPI_Abort(comm, 1);

  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if (check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);

  /* Attach linear solver CVSPGMR */
  flag = CVSpgmr(cvode_mem, PREC_LEFT, 0);
  if (check_flag(&flag, "CVSpgmr", 1, my_pe)) MPI_Abort(comm, 1);

  flag = CVSpgmrSetPreconditioner(cvode_mem, Precond, PSolve, predata);
  if (check_flag(&flag, "CVSpgmrSetPreconditioner", 1, my_pe)) MPI_Abort(comm, 1);

  if(my_pe == 0)
    printf("\n2-species diurnal advection-diffusion problem\n");

  /* Sensitivity-related settings */
  if( sensi) {

    plist = (int *) malloc(NS * sizeof(int));
    if (check_flag((void *)plist, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
    for (is=0; is<NS; is++) plist[is] = is+1;

    pbar = (realtype *) malloc(NS*sizeof(realtype));
    if (check_flag((void *)pbar, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
    for (is=0; is<NS; is++) pbar[is] = data->p[plist[is]-1];

    uS = N_VNewVectorArray_Parallel(NS, comm, local_N, neq);
    if (check_flag((void *)uS, "N_VNewVectorArray_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
    for (is = 0; is < NS; is++)
      N_VConst(ZERO,uS[is]);

    flag = CVodeSensMalloc(cvode_mem, NS, sensi_meth, uS);
    if (check_flag(&flag, "CVodeSensMalloc", 1, my_pe)) MPI_Abort(comm, 1);

    flag = CVodeSetSensErrCon(cvode_mem, err_con);
    if (check_flag(&flag, "CVodeSetSensErrCon", 1, my_pe)) MPI_Abort(comm, 1);

    flag = CVodeSetSensRho(cvode_mem, ZERO);
    if (check_flag(&flag, "CVodeSetSensRho", 1, my_pe)) MPI_Abort(comm, 1);

    flag = CVodeSetSensParams(cvode_mem, data->p, pbar, plist);
    if (check_flag(&flag, "CVodeSetSensParams", 1, my_pe)) MPI_Abort(comm, 1);

    if(my_pe == 0) {
      printf("Sensitivity: YES ");
      if(sensi_meth == CV_SIMULTANEOUS)
        printf("( SIMULTANEOUS +");
      else
```

```
        if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
        else                           printf("( STAGGERED1 +");
      if(err_con) printf(" FULL ERROR CONTROL )");
      else        printf(" PARTIAL ERROR CONTROL )");
    }

  } else {

    if(my_pe == 0) printf("Sensitivity: NO ");

  }

  if (my_pe == 0) {
    printf("\n\n");
    printf("=========================================================================\n");
    printf("     T     Q       H      NST                     Bottom left  Top right \n");
    printf("=========================================================================\n");
  }

  /* 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, CV_NORMAL);
    if (check_flag(&flag, "CVode", 1, my_pe)) break;
    PrintOutput(cvode_mem, my_pe, comm, t, u);
    if (sensi) {
      flag = CVodeGetSens(cvode_mem, t, uS);
      if (check_flag(&flag, "CVodeGetSens", 1, my_pe)) break;
      PrintOutputS(my_pe, comm, uS);
    }
    if (my_pe == 0)
      printf("-------------------------------------------------------------------------\n");
  }

  /* Print final statistics */
  if (my_pe == 0) PrintFinalStats(cvode_mem, sensi);

  /* Free memory */
  N_VDestroy_Parallel(u);
  if (sensi) {
    N_VDestroyVectorArray_Parallel(uS, NS);
    free(plist);
    free(pbar);
  }
  free(data->p);
  free(data);
  FreePreconData(predata);
  CVodeFree(cvode_mem);

  MPI_Finalize();

  return(0);
}

/*
```

```
 *----------------------------------------------------------------------
 * FUNCTIONS CALLED BY CVODES
 *----------------------------------------------------------------------
 */

/*
 * 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(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  realtype *udata, *dudata;
  UserData data;

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

  /* Call ucomm to do inter-processor communicaiton */
  ucomm (t, u, data);

  /* Call fcalc to calculate all right-hand sides */
  fcalc (t, udata, dudata, data);
}

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

static int Precond(realtype tn, N_Vector u, N_Vector fu,
                   booleantype jok, booleantype *jcurPtr,
                   realtype gamma, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
  realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
  realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
  int ier;
  long int nvmxsub, *(*pivot)[MYSUB], offset;
  int lx, ly, jx, jy, isubx, isuby;
  realtype *udata, **a, **j;
  PreconData predata;
  UserData data;
  realtype 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 = NV_DATA_P(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 = data->vdco;
    hordco  = data->hdco;

    /* 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 - RCONST(0.5))*dely;
      yup = ydn + dely;
      cydn = verdco*exp(RCONST(0.2)*ydn);
      cyup = verdco*exp(RCONST(0.2)*yup);
      diag = -(cydn + cyup + RCONST(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(realtype tn, N_Vector u, N_Vector fu,
                  N_Vector r, N_Vector z,
                  realtype gamma, realtype delta,
                  int lr, void *P_data, N_Vector vtemp)
{
  realtype **(*P)[MYSUB];
  long int nvmxsub, *(*pivot)[MYSUB];
  int lx, ly;
  realtype *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(RCONST(1.0), r, z);

  nvmxsub = data->nvmxsub;
  zdata = NV_DATA_P(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);
}

/*
 *-----------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *-----------------------------------------------------------------
 */

/*
 * Process and verify arguments to pvfkx.
 */

static void ProcessArgs(int argc, char *argv[], int my_pe,
                        booleantype *sensi, int *sensi_meth, booleantype *err_con)
{
  *sensi = FALSE;
  *sensi_meth = -1;
  *err_con = FALSE;

  if (argc < 2) WrongArgs(my_pe, argv[0]);

  if (strcmp(argv[1],"-nosensi") == 0)
    *sensi = FALSE;
  else if (strcmp(argv[1],"-sensi") == 0)
    *sensi = TRUE;
  else
    WrongArgs(my_pe, argv[0]);

  if (*sensi) {

    if (argc != 4)
      WrongArgs(my_pe, argv[0]);

    if (strcmp(argv[2],"sim") == 0)
      *sensi_meth = CV_SIMULTANEOUS;
    else if (strcmp(argv[2],"stg") == 0)
      *sensi_meth = CV_STAGGERED;
    else if (strcmp(argv[2],"stg1") == 0)
      *sensi_meth = CV_STAGGERED1;
    else
      WrongArgs(my_pe, argv[0]);

    if (strcmp(argv[3],"t") == 0)
      *err_con = TRUE;
    else if (strcmp(argv[3],"f") == 0)
      *err_con = FALSE;
    else
      WrongArgs(my_pe, argv[0]);
  }

}
```

```
static void WrongArgs(int my_pe, char *name)
{
  if (my_pe == 0) {
    printf("\nUsage: %s [-nosensi] [-sensi sensi_meth err_con]\n",name);
    printf("         sensi_meth = sim, stg, or stg1\n");
    printf("         err_con    = t or f\n");
  }
  MPI_Finalize();
  exit(0);
}


/*
 * Allocate memory for data structure of type PreconData.
 */

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);
}

/*
 * Free preconditioner 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 user data.
 */

static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
{
  long int isubx, isuby;
  realtype KH, VEL, KV0;

  /* Set problem parameters */
  data->p[0]  = RCONST(1.63e-16);      /* Q1  coeffs. q1, q2, c3             */
  data->p[1]  = RCONST(4.66e-16);      /* Q2                                 */
  data->p[2]  = RCONST(3.7e16);        /* C3                                 */
  data->p[3]  = RCONST(22.62);         /* A3  coeff. in expression for q3(t) */
  data->p[4]  = RCONST(7.601);         /* A4  coeff. in expression for q4(t) */
  KH  = data->p[5]  = RCONST(4.0e-6);  /* KH  horizontal diffusivity Kh      */
  VEL = data->p[6]  = RCONST(0.001);   /* VEL advection velocity V           */
  KV0 = data->p[7]  = RCONST(1.0e-8);  /* KV0 coeff. in Kv(z)                */

  /* Set problem constants */
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/((realtype)(MX-1));
  data->dy = (YMAX-YMIN)/((realtype)(MY-1));
  data->hdco = KH/SQR(data->dx);
  data->haco = VEL/(RCONST(2.0)*data->dx);
  data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;

  /* 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);
}

/*
 * Set initial conditions in u.
 */

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

  /* Set pointer to data array in vector u */
  udata = NV_DATA_P(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 = RCONST(0.5)*(XMIN + XMAX);
  ymid = RCONST(0.5)*(YMIN + YMAX);
  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    y = YMIN + jy*dy;
    cy = SQR(RCONST(0.1)*(y - ymid));
    cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      x = XMIN + jx*dx;
      cx = SQR(RCONST(0.1)*(x - xmid));
      cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
      udata[offset  ] = C1_SCALE*cx*cy;
      udata[offset+1] = C2_SCALE*cx*cy;
      offset = offset + 2;
    }
  }
}

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

static void BSend(MPI_Comm comm, int my_pe, long int isubx,
                  long int isuby, long int dsizex, long int dsizey,
                  realtype udata[])
{
  int i, ly;
  long int offsetu, offsetbuf;
  realtype 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 realtype 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[], int my_pe,
                      long int isubx, long int isuby,
                      long int dsizex, long int dsizey,
                      realtype uext[], realtype buffer[])
{
  long int offsetue;

  /* Have bufleft and bufright use the same buffer */
  realtype *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]);
```

```c
  }

  /* 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 realtype 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[], long int isubx, long int isuby,
                      long int dsizex, realtype uext[], realtype buffer[])
{
  int i, ly;
  long int dsizex2, offsetue, offsetbuf;
  realtype *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(realtype t, N_Vector u, UserData data)
{
  realtype *udata, *uext, buffer[2*NVARS*MYSUB];
  MPI_Comm comm;
  int my_pe;
  long int isubx, isuby, nvmxsub, nvmysub;
  MPI_Request request[4];

  udata = NV_DATA_P(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(realtype t, realtype udata[], realtype dudata[], UserData data)
{
  realtype *uext;
  realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
  realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
```

```
  realtype q4coef, dely, verdco, hordco, horaco;
  int i, lx, ly, jx, jy;
  long int isubx, isuby, nvmxsub, nvmxsub2, offsetu, offsetue;
  realtype Q1, Q2, C3, A3, A4, KH, VEL, KV0;

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

  /* 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];

  /* 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 = data->vdco;
  hordco = data->hdco;
  horaco = data->haco;

  /* 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 > ZERO) {
    q3 = exp(-A3/s);
    q4coef = exp(-A4/s);
  } else {
    q3 = ZERO;
    q4coef = ZERO;
  }
  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(RCONST(0.2)*ydn);
    cyup = verdco*exp(RCONST(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 + RCONST(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;
    }
  }

}

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

static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
                        realtype t, N_Vector u)
{
  long int nst;
  int qu, flag;
  realtype hu, *udata, tempu[2];
  long int npelast, i0, i1;
  MPI_Status status;

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

  /* Send c 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 c 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);

    flag = CVodeGetNumSteps(cvode_mem, &nst);
    check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
    flag = CVodeGetLastOrder(cvode_mem, &qu);
    check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
    flag = CVodeGetLastStep(cvode_mem, &hu);
    check_flag(&flag, "CVodeGetLastStep", 1, my_pe);

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%8.3Le %2d  %8.3Le %5ld\n", t,qu,hu,nst);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%8.3le %2d  %8.3le %5ld\n", t,qu,hu,nst);
#else
    printf("%8.3e %2d  %8.3e %5ld\n", t,qu,hu,nst);
#endif

    printf("                                Solution       ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%12.4Le %12.4Le \n", udata[0], tempu[0]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%12.4le %12.4le \n", udata[0], tempu[0]);
#else
    printf("%12.4e %12.4e \n", udata[0], tempu[0]);
#endif

    printf("                                               ");

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%12.4Le %12.4Le \n", udata[1], tempu[1]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%12.4le %12.4le \n", udata[1], tempu[1]);
#else
    printf("%12.4e %12.4e \n", udata[1], tempu[1]);
#endif

  }

}

/*
 * Print sampled sensitivity values.
 */

static void PrintOutputS(int my_pe, MPI_Comm comm, N_Vector *uS)
{
  realtype *sdata, temps[2];
  long int npelast, i0, i1;
  MPI_Status status;

  npelast = NPEX*NPEY - 1;

  sdata = NV_DATA_P(uS[0]);
```

```

  /* Send s1 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(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
      temps[0] = sdata[i0];
      temps[1] = sdata[i1];
    }
  }

  /* On PE 0, receive s1 at top right, then print sampled sensitivity values */
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    printf("                                ----------------------------------------\n");
    printf("                                Sensitivity 1  ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%12.4Le %12.4Le \n", sdata[0], temps[0]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%12.4le %12.4le \n", sdata[0], temps[0]);
#else
    printf("%12.4e %12.4e \n", sdata[0], temps[0]);
#endif
    printf("                                               ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%12.4Le %12.4Le \n", sdata[1], temps[1]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%12.4le %12.4le \n", sdata[1], temps[1]);
#else
    printf("%12.4e %12.4e \n", sdata[1], temps[1]);
#endif
  }

  sdata = NV_DATA_P(uS[1]);

  /* Send s2 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(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
      temps[0] = sdata[i0];
      temps[1] = sdata[i1];
    }
  }

  /* On PE 0, receive s2 at top right, then print sampled sensitivity values */
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
```

```
    printf("                                -----------------------------------------\n");
    printf("                                Sensitivity 2  ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%12.4Le %12.4Le \n", sdata[0], temps[0]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%12.4le %12.4le \n", sdata[0], temps[0]);
#else
    printf("%12.4e %12.4e \n", sdata[0], temps[0]);
#endif
    printf("                                               ");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%12.4Le %12.4Le \n", sdata[1], temps[1]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%12.4le %12.4le \n", sdata[1], temps[1]);
#else
    printf("%12.4e %12.4e \n", sdata[1], temps[1]);
#endif
  }
}

/*
 * Print final statistics from the CVODES memory.
 */

static void PrintFinalStats(void *cvode_mem, booleantype sensi)
{
  long int nst;
  long int nfe, nsetups, nni, ncfn, netf;
  long int nfSe, nfeS, nsetupsS, nniS, ncfnS, netfS;
  int flag;

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1, 0);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);

  if (sensi) {
    flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
    check_flag(&flag, "CVodeGetNumSensRhsEvals", 1, 0);
    flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
    check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1, 0);
    flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
    check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1, 0);
    flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
    check_flag(&flag, "CVodeGetNumSensErrTestFails", 1, 0);
    flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
```

```
    check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1, 0);
    flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
    check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1, 0);
  }

  printf("\nFinal Statistics\n\n");
  printf("nst     = %5ld\n\n", nst);
  printf("nfe     = %5ld\n",   nfe);
  printf("netf    = %5ld    nsetups  = %5ld\n", netf, nsetups);
  printf("nni     = %5ld    ncfn     = %5ld\n", nni, ncfn);

  if(sensi) {
    printf("\n");
    printf("nfSe    = %5ld    nfeS     = %5ld\n", nfSe, nfeS);
    printf("netfs   = %5ld    nsetupsS = %5ld\n", netfS, nsetupsS);
    printf("nniS    = %5ld    ncfnS    = %5ld\n", nniS, ncfnS);
  }

}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr,
            "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
              id, funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr,
            "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
```

```
              id, funcname);
    return(1); }

  return(0);
}
```

# D listing of cvadx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.18.2.7 $
 * $Date: 2005/04/28 20:06:31 $
 * -----------------------------------------------------------------
 * Programmer(s): Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Copyright (c) 2002, The Regents of the University of California.
 * Produced at the Lawrence Livermore National Laboratory.
 * All rights reserved.
 * For details, see sundials/cvodes/LICENSE.
 * -----------------------------------------------------------------
 * Adjoint sensitivity example problem.
 * The following is a simple example problem, with the coding
 * needed for its solution by CVODES. The problem is from chemical
 * kinetics, and consists of the following three rate equations.
 *    dy1/dt = -p1*y1 + p2*y2*y3
 *    dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
 *    dy3/dt =  p3*(y2)^2
 * on the interval from t = 0.0 to t = 4.e10, with initial
 * conditions: y1 = 1.0, y2 = y3 = 0. The reaction rates are:
 * p1=0.04, p2=1e4, and p3=3e7. The problem is stiff.
 * This program solves the problem with the BDF method, Newton
 * iteration with the CVODE dense linear solver, and a user-supplied
 * Jacobian routine.
 * It uses a scalar relative tolerance and a vector absolute
 * tolerance.
 * Output is printed in decades from t = .4 to t = 4.e10.
 * Run statistics (optional outputs) are printed at the end.
 *
 * Optionally, CVODES can compute sensitivities with respect to
 * the problem parameters p1, p2, and p3 of the following quantity:
 *   G = int_t0^t1 g(t,p,y) dt
 * where
 *   g(t,p,y) = y3
 *
 * The gradient dG/dp is obtained as:
 *   dG/dp = int_t0^t1 (g_p - lambda^T f_p ) dt - lambda^T(t0)*y0_p
 *         = - xi^T(t0) - lambda^T(t0)*y0_p
 * where lambda and xi are solutions of:
 *   d(lambda)/dt = - (f_y)^T * lambda - (g_y)^T
 *   lambda(t1) = 0
 * and
 *   d(xi)/dt = - (f_p)^T * lambda + (g_p)^T
 *   xi(t1) = 0
 *
 * During the backward integration, CVODES also evaluates G as
 *   G = - phi(t0)
 * where
 *   d(phi)/dt = g(t,y,p)
 *   phi(t1) = 0
 * -----------------------------------------------------------------
```

```
 */

#include <stdio.h>
#include <stdlib.h>
#include "cvodes.h"
#include "cvodea.h"
#include "cvdense.h"
#include "nvector_serial.h"
#include "sundialstypes.h"
#include "sundialsmath.h"

/* Accessor macros */

#define Ith(v,i)    NV_Ith_S(v,i-1)       /* i-th vector component i= 1..NEQ */
#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j = 1..NEQ */

/* Problem Constants */

#define NEQ      3             /* number of equations                  */

#define RTOL     RCONST(1e-6)  /* scalar relative tolerance            */

#define ATOL1    RCONST(1e-8)  /* vector absolute tolerance components */
#define ATOL2    RCONST(1e-14)
#define ATOL3    RCONST(1e-6)

#define ATOLl    RCONST(1e-5)  /* absolute tolerance for adjoint vars. */
#define ATOLq    RCONST(1e-6)  /* absolute tolerance for quadratures   */

#define T0       RCONST(0.0)   /* initial time                         */
#define TOUT     RCONST(4e7)   /* final time                           */

#define TB1      RCONST(4e7)   /* starting point for adjoint problem   */
#define TB2      RCONST(50.0)  /* starting point for adjoint problem   */

#define STEPS    150           /* number of steps between check points */

#define NP       3             /* number of problem parameters         */

#define ZERO     RCONST(0.0)


/* Type : UserData */

typedef struct {
  realtype p[3];
} *UserData;

/* Prototypes of user-supplied functions */

static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
static void Jac(long int N, DenseMat J, realtype t,
                N_Vector y, N_Vector fy, void *jac_data,
                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

```
static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data);
static int ewt(N_Vector y, N_Vector w, void *e_data);

static void fB(realtype t, N_Vector y,
               N_Vector yB, N_Vector yBdot, void *f_dataB);
static void JacB(long int NB, DenseMat JB, realtype t,
                 N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
                 N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B);
static void fQB(realtype t, N_Vector y, N_Vector yB,
                N_Vector qBdot, void *fQ_dataB);


/* Prototypes of private functions */

static void PrintOutput(N_Vector yB, N_Vector qB);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  UserData data;

  void *cvadj_mem;
  void *cvode_mem;

  realtype reltolQ, abstolQ;
  N_Vector y, q;

  realtype reltolB, abstolB, abstolQB;
  N_Vector yB, qB;

  realtype time;
  int flag, ncheck;

  data = NULL;
  cvadj_mem = cvode_mem = NULL;
  y = yB = qB = NULL;

  /* Print problem description */
  printf("\n\n Adjoint Sensitivity Example for Chemical Kinetics\n");
  printf(" -------------------------------------------------\n\n");
  printf("ODE: dy1/dt = -p1*y1 + p2*y2*y3\n");
  printf("     dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2\n");
  printf("     dy3/dt =  p3*(y2)^2\n\n");
  printf("Find dG/dp for\n");
  printf("     G = int_t0^tB0 g(t,p,y) dt\n");
  printf("     g(t,p,y) = y3\n\n\n");


```

```
/* User data structure */
data = (UserData) malloc(sizeof *data);
if (check_flag((void *)data, "malloc", 2)) return(1);
data->p[0] = RCONST(0.04);
data->p[1] = RCONST(1.0e4);
data->p[2] = RCONST(3.0e7);

/* Initialize y */
y = N_VNew_Serial(NEQ);
if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
Ith(y,1) = RCONST(1.0);
Ith(y,2) = ZERO;
Ith(y,3) = ZERO;

/* Initialize q */
q = N_VNew_Serial(1);
if (check_flag((void *)q, "N_VNew_Serial", 0)) return(1);
Ith(q,1) = ZERO;

/* Set the scalar realtive and absolute tolerances reltolQ and abstolQ */
reltolQ = RTOL;
abstolQ = ATOLq;

/* Create and allocate CVODES memory for forward run */
printf("Create and allocate CVODES memory for forward runs\n");

cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
if (check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

flag = CVodeMalloc(cvode_mem, f, T0, y, CV_WF, 0.0, NULL);
if (check_flag(&flag, "CVodeMalloc", 1)) return(1);

flag = CVodeSetEwtFn(cvode_mem, ewt, NULL);
if (check_flag(&flag, "CVodeSetEwtFn", 1)) return(1);

flag = CVodeSetFdata(cvode_mem, data);
if (check_flag(&flag, "CVodeSetFdata", 1)) return(1);

flag = CVDense(cvode_mem, NEQ);
if (check_flag(&flag, "CVDense", 1)) return(1);

flag = CVDenseSetJacFn(cvode_mem, Jac, data);
if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);

flag = CVodeQuadMalloc(cvode_mem, fQ, q);
if (check_flag(&flag, "CVodeQuadMalloc", 1)) return(1);

flag = CVodeSetQuadFdata(cvode_mem, data);
if (check_flag(&flag, "CVodeSetQuadFdata", 1)) return(1);

flag = CVodeSetQuadErrCon(cvode_mem, TRUE, CV_SS, reltolQ, &abstolQ);
if (check_flag(&flag, "CVodeSetQuadErrCon", 1)) return(1);

/* Allocate global memory */
```

```
  printf("Allocate global memory\n");

  cvadj_mem = CVadjMalloc(cvode_mem, STEPS);
  if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0)) return(1);

  /* Perform forward run */
  printf("Forward integration ... ");

  flag = CVodeF(cvadj_mem, TOUT, y, &time, CV_NORMAL, &ncheck);
  if (check_flag(&flag, "CVodeF", 1)) return(1);

  flag = CVodeGetQuad(cvode_mem, TOUT, q);
  if (check_flag(&flag, "CVodeGetQuad", 1)) return(1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("done. ncheck = %d   G: %12.4Le \n",ncheck, Ith(q,1));
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("done. ncheck = %d   G: %12.4le \n",ncheck, Ith(q,1));
#else
  printf("done. ncheck = %d   G: %12.4e \n",ncheck, Ith(q,1));
#endif

  /* Initialize yB */
  yB = N_VNew_Serial(NEQ);
  if (check_flag((void *)yB, "N_VNew_Serial", 0)) return(1);
  Ith(yB,1) = ZERO;
  Ith(yB,2) = ZERO;
  Ith(yB,3) = ZERO;

  /* Initialize qB */
  qB = N_VNew_Serial(NP);
  if (check_flag((void *)qB, "N_VNew", 0)) return(1);
  Ith(qB,1) = ZERO;
  Ith(qB,2) = ZERO;
  Ith(qB,3) = ZERO;

  /* Set the scalar relative tolerance reltolB */
  reltolB = RTOL;

  /* Set the scalar absolute tolerance abstolB */
  abstolB = ATOLl;

  /* Set the scalar absolute tolerance abstolQB */
  abstolQB = ATOLq;

  /* Create and allocate CVODES memory for backward run */
  printf("\nCreate and allocate CVODES memory for backward run\n");

  flag = CVodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
  if (check_flag(&flag, "CVodeCreateB", 1)) return(1);

  flag = CVodeMallocB(cvadj_mem, fB, TB1, yB, CV_SS, reltolB, &abstolB);
  if (check_flag(&flag, "CVodeMallocB", 1)) return(1);
```

```
  flag = CVodeSetFdataB(cvadj_mem, data);
  if (check_flag(&flag, "CVodeSetFdataB", 1)) return(1);

  flag = CVDenseB(cvadj_mem, NEQ);
  if (check_flag(&flag, "CVDenseB", 1)) return(1);

  flag = CVDenseSetJacFnB(cvadj_mem, JacB, data);
  if (check_flag(&flag, "CVDenseSetJacFnB", 1)) return(1);

  flag = CVodeQuadMallocB(cvadj_mem, fQB, qB);
  if (check_flag(&flag, "CVodeQuadMallocB", 1)) return(1);

  flag = CVodeSetQuadFdataB(cvadj_mem, data);
  if (check_flag(&flag, "CVodeSetQuadFdataB", 1)) return(1);

  flag = CVodeSetQuadErrConB(cvadj_mem, TRUE, CV_SS, reltolB, &abstolQB);
  if (check_flag(&flag, "CVodeSetQuadErrConB", 1)) return(1);

  /* Backward Integration */
  printf("Integrate backwards\n");

  flag = CVodeB(cvadj_mem, T0, yB, &time, CV_NORMAL);
  if (check_flag(&flag, "CVodeB", 1)) return(1);

  flag = CVodeGetQuadB(cvadj_mem, qB);
  if (check_flag(&flag, "CVodeGetQuadB", 1)) return(1);

  PrintOutput(yB, qB);

  /* Reinitialize backward phase (new tB0) */
  Ith(yB,1) = ZERO;
  Ith(yB,2) = ZERO;
  Ith(yB,3) = ZERO;

  Ith(qB,1) = ZERO;
  Ith(qB,2) = ZERO;
  Ith(qB,3) = ZERO;

  printf("Re-initialize CVODES memory for backward run\n");

  flag = CVodeReInitB(cvadj_mem, fB, TB2, yB, CV_SS, reltolB, &abstolB);
  if (check_flag(&flag, "CVodeReInitB", 1)) return(1);

  flag = CVodeQuadReInitB(cvadj_mem, fQB, qB);
  if (check_flag(&flag, "CVodeQuadReInitB", 1)) return(1);

  /* Backward Integration */
  printf("Integrate backwards\n");

  flag = CVodeB(cvadj_mem, T0, yB, &time, CV_NORMAL);
  if (check_flag(&flag, "CVodeB", 1)) return(1);

  flag = CVodeGetQuadB(cvadj_mem, qB);
  if (check_flag(&flag, "CVodeGetQuadB", 1)) return(1);
```

```
  PrintOutput(yB, qB);

  /* Free memory */
  printf("Free memory\n\n");

  CVodeFree(cvode_mem);
  N_VDestroy_Serial(y);
  N_VDestroy_Serial(q);
  N_VDestroy_Serial(yB);
  N_VDestroy_Serial(qB);
  CVadjFree(cvadj_mem);
  free(data);

  return(0);

}

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY CVODES
 *--------------------------------------------------------------------
 */

/*
 * f routine. Compute f(t,y).
*/

static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
{
  realtype y1, y2, y3, yd1, yd3;
  UserData data;
  realtype p1, p2, p3;

  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
  data = (UserData) f_data;
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
  yd3 = Ith(ydot,3) = p3*y2*y2;
        Ith(ydot,2) = -yd1 - yd3;
}

/*
 * Jacobian routine. Compute J(t,y).
*/

static void Jac(long int N, DenseMat J, realtype t,
                N_Vector y, N_Vector fy, void *jac_data,
                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  realtype y1, y2, y3;
  UserData data;
  realtype p1, p2, p3;
```

```
  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
  data = (UserData) jac_data;
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  IJth(J,1,1) = -p1;  IJth(J,1,2) = p2*y3;          IJth(J,1,3) = p2*y2;
  IJth(J,2,1) =  p1;  IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
                      IJth(J,3,2) = 2*p3*y2;
}

/*
 * fQ routine. Compute fQ(t,y).
*/

static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
{
  Ith(qdot,1) = Ith(y,3);
}

/*
 * EwtSet function. Computes the error weights at the current solution.
 */

static int ewt(N_Vector y, N_Vector w, void *e_data)
{
  int i;
  realtype yy, ww, rtol, atol[3];

  rtol    = RTOL;
  atol[0] = ATOL1;
  atol[1] = ATOL2;
  atol[2] = ATOL3;

  for (i=1; i<=3; i++) {
    yy = Ith(y,i);
    ww = rtol * ABS(yy) + atol[i-1];
    if (ww <= 0.0) return (-1);
    Ith(w,i) = 1.0/ww;
  }

  return(0);
}

/*
 * fB routine. Compute fB(t,y,yB).
*/

static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot, void *f_dataB)
{
  UserData data;
  realtype y1, y2, y3;
  realtype p1, p2, p3;
  realtype l1, l2, l3;
  realtype l21, l32, y23;
```

```

  data = (UserData) f_dataB;

  /* The p vector */
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  /* The y vector */
  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

  /* The lambda vector */
  l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);

  /* Temporary variables */
  l21 = l2-l1;
  l32 = l3-l2;
  y23 = y2*y3;

  /* Load yBdot */
  Ith(yBdot,1) = - p1*l21;
  Ith(yBdot,2) = p2*y3*l21 - RCONST(2.0)*p3*y2*l32;
  Ith(yBdot,3) = p2*y2*l21 - RCONST(1.0);
}

/*
 * JacB routine. Compute JB(t,y,yB).
*/

static void JacB(long int NB, DenseMat JB, realtype t,
                 N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
                 N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B)
{
  UserData data;
  realtype y1, y2, y3;
  realtype p1, p2, p3;

  data = (UserData) jac_dataB;

  /* The p vector */
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  /* The y vector */
  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

  /* Load JB */
  IJth(JB,1,1) = p1;     IJth(JB,1,2) = -p1;
  IJth(JB,2,1) = -p2*y3; IJth(JB,2,2) = p2*y3+2.0*p3*y2; IJth(JB,2,3) = RCONST(-2.0)*p3*y2;
  IJth(JB,3,1) = -p2*y2; IJth(JB,3,2) = p2*y2;
}

/*
 * fQB routine. Compute integrand for quadratures
*/

static void fQB(realtype t, N_Vector y, N_Vector yB,
```

```
                N_Vector qBdot, void *fQ_dataB)
{
  UserData data;
  realtype y1, y2, y3;
  realtype p1, p2, p3;
  realtype l1, l2, l3;
  realtype l21, l32, y23;

  data = (UserData) fQ_dataB;

  /* The p vector */
  p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];

  /* The y vector */
  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

  /* The lambda vector */
  l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);

  /* Temporary variables */
  l21 = l2-l1;
  l32 = l3-l2;
  y23 = y2*y3;

  Ith(qBdot,1) = y1*l21;
  Ith(qBdot,2) = - y23*l21;
  Ith(qBdot,3) = y2*y2*l32;
}

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * Print results after backward integration
 */

static void PrintOutput(N_Vector yB, N_Vector qB)
{
  printf("--------------------------------------------------------\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("tB0:        %12.4Le\n",TB1);
  printf("dG/dp:      %12.4Le %12.4Le %12.4Le\n",
         -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
  printf("lambda(t0): %12.4Le %12.4Le %12.4Le\n",
         Ith(yB,1), Ith(yB,2), Ith(yB,3));
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("tB0:        %12.4le\n",TB1);
  printf("dG/dp:      %12.4le %12.4le %12.4le\n",
         -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
  printf("lambda(t0): %12.4le %12.4le %12.4le\n",
         Ith(yB,1), Ith(yB,2), Ith(yB,3));
```

```
#else
  printf("tB0:        %12.4e\n",TB1);
  printf("dG/dp:      %12.4e %12.4e %12.4e\n",
         -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
  printf("lambda(t0): %12.4e %12.4e %12.4e\n",
         Ith(yB,1), Ith(yB,2), Ith(yB,3));
#endif
  printf("--------------------------------------------------------\n\n");
}

/*
 * Check function return value.
 *    opt == 0 means SUNDIALS function allocates memory so check if
 *             returned NULL pointer
 *    opt == 1 means SUNDIALS function returns a flag so check if
 *             flag >= 0
 *    opt == 2 means function allocates memory so check if returned
 *             NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  return(0);
}
```

# E Listing of pvanx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.17.2.3 $
 * $Date: 2005/04/28 20:06:28 $
 * -----------------------------------------------------------------
 * Programmer(s): Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * The following is a simple example problem, with the program for
 * its solution by CVODE. The problem is the semi-discrete form of
 * the advection-diffusion equation in 1-D:
 *   du/dt = p1 * d^2u / dx^2 + p2 * du / dx
 * on the interval 0 <= x <= 2, and the time interval 0 <= t <= 5.
 * Homogeneous Dirichlet boundary conditions are posed, and the
 * initial condition is:
 *   u(x,t=0) = x(2-x)exp(2x).
 * The nominal values of the two parameters are: p1=1.0, p2=0.5
 * The PDE is discretized on a uniform grid of size MX+2 with
 * central differencing, and with boundary values eliminated,
 * leaving an ODE system of size NEQ = MX.
 * This program solves the problem with the option for nonstiff
 * systems: ADAMS method and functional iteration.
 * It uses scalar relative and absolute tolerances.
 *
 * In addition to the solution, sensitivities with respect to p1
 * and p2 as well as with respect to initial conditions are
 * computed for the quantity:
 *    g(t, u, p) = int_x u(x,t) at t = 5
 * These sensitivities are obtained by solving the adjoint system:
 *    dv/dt = -p1 * d^2 v / dx^2 + p2 * dv / dx
 * with homogeneous Ditrichlet boundary conditions and the final
 * condition:
 *    v(x,t=5) = 1.0
 * Then, v(x, t=0) represents the sensitivity of g(5) with respect
 * to u(x, t=0) and the gradient of g(5) with respect to p1, p2 is
 *    (dg/dp)^T = [  int_t int_x (v * d^2u / dx^2) dx dt ]
 *                [  int_t int_x (v * du / dx) dx dt     ]
 *
 * This version uses MPI for user routines.
 * Execute with Number of Processors = N,  with 1 <= N <= MX.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#include "cvodes.h"
#include "cvodea.h"
#include "nvector_parallel.h"
#include "sundialstypes.h"
```

```
/* Problem Constants */

#define XMAX  RCONST(2.0)   /* domain boundary            */
#define MX    20            /* mesh dimension             */
#define NEQ   MX            /* number of equations        */
#define ATOL  RCONST(1.e-5) /* scalar absolute tolerance  */
#define T0    RCONST(0.0)   /* initial time               */
#define TOUT  RCONST(2.5)   /* output time increment      */

/* Adjoint Problem Constants */

#define NP    2            /* number of parameters       */
#define STEPS 200          /* steps between check points */

#define ZERO RCONST(0.0)
#define ONE  RCONST(1.0)
#define TWO  RCONST(2.0)

/* Type : UserData */

typedef struct {
  realtype p[2];            /* model parameters                         */
  realtype dx;              /* spatial discretization grid              */
  realtype hdcoef, hacoef;  /* diffusion and advection coefficients     */
  long int local_N;
  long int npes, my_pe;     /* total number of processes and current ID */
  long int nperpe, nrem;
  MPI_Comm comm;            /* MPI communicator                         */
  realtype *z1, *z2;        /* work space                               */
} *UserData;

/* Prototypes of user-supplied funcitons */

static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
static void fB(realtype t, N_Vector u,
               N_Vector uB, N_Vector uBdot, void *f_dataB);

/* Prototypes of private functions */

static void SetIC(N_Vector u, realtype dx, long int my_length, long int my_base);
static void SetICback(N_Vector uB, long int my_base);
static realtype Xintgr(realtype *z, long int l, realtype dx);
static realtype Compute_g(N_Vector u, UserData data);
static void PrintOutput(realtype g_val, N_Vector uB, UserData data);
static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */
```

```
int main(int argc, char *argv[])
{
  UserData data;

  void *cvadj_mem;
  void *cvode_mem;

  N_Vector u;
  realtype reltol, abstol;

  N_Vector uB;

  realtype dx, t, g_val;
  int flag, my_pe, nprocs, npes, ncheck;
  long int local_N=0, nperpe, nrem, my_base=0;

  MPI_Comm comm;

  data = NULL;
  cvadj_mem = cvode_mem = NULL;
  u = uB = NULL;

  /*------------------------------------------------------
    Initialize MPI and get total number of pe's, and my_pe
    ------------------------------------------------------*/
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &nprocs);
  MPI_Comm_rank(comm, &my_pe);

  npes = nprocs - 1; /* pe's dedicated to PDE integration */

  if ( npes <= 0 ) {
    if (my_pe == npes)
      fprintf(stderr, "\nMPI_ERROR(%d): number of processes must be >= 2\n\n",
              my_pe);
    MPI_Finalize();
    return(1);
  }

  /*-----------------------
    Set local vector length
    -----------------------*/
  nperpe = NEQ/npes;
  nrem = NEQ - npes*nperpe;
  if (my_pe < npes) {

    /* PDE vars. distributed to this proccess */
    local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
    my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;

  } else {

    /* Make last process inactive for forward phase */
```

```
    local_N = 0;

  }

  /*------------------------------------
    Allocate and load user data structure
    ------------------------------------*/
  data = (UserData) malloc(sizeof *data);
  if (check_flag((void *)data , "malloc", 2, my_pe)) MPI_Abort(comm, 1);
  data->p[0] = ONE;
  data->p[1] = RCONST(0.5);
  dx = data->dx = XMAX/((realtype)(MX+1));
  data->hdcoef = data->p[0]/(dx*dx);
  data->hacoef = data->p[1]/(TWO*dx);
  data->comm = comm;
  data->npes = npes;
  data->my_pe = my_pe;
  data->nperpe = nperpe;
  data->nrem = nrem;
  data->local_N = local_N;

  /*-------------------------
    Forward integration phase
    -------------------------*/

  /* Set relative and absolute tolerances for forward phase */
  reltol = ZERO;
  abstol = ATOL;

  /* Allocate and initialize forward variables */
  u = N_VNew_Parallel(comm, local_N, NEQ);
  if (check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  SetIC(u, dx, local_N, my_base);

  /* Allocate CVODES memory for forward integration */
  cvode_mem = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
  if (check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);

  flag = CVodeSetFdata(cvode_mem, data);
  if (check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);

  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if (check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);

  /* Allocate combined forward/backward memory */
  cvadj_mem = CVadjMalloc(cvode_mem, STEPS);
  if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0, my_pe)) MPI_Abort(comm, 1);

  /* Integrate to TOUT and collect check point information */
  flag = CVodeF(cvadj_mem, TOUT, u, &t, CV_NORMAL, &ncheck);
  if (check_flag(&flag, "CVodeF", 1, my_pe)) MPI_Abort(comm, 1);

  if(my_pe == npes)
    printf("(PE# %d) Number of check points: %d\n",my_pe, ncheck);
```

```
  /*---------------------------
    Compute and value of g(t_f)
    ---------------------------*/
  g_val = Compute_g(u, data);

  /*--------------------------
    Backward integration phase
    --------------------------*/

  if (my_pe == npes) {

    /* Activate last process for integration of the quadrature equations */
    local_N = NP;

  } else {

    /* Allocate work space */
    data->z1 = (realtype *)malloc(local_N*sizeof(realtype));
    if (check_flag((void *)data->z1, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
    data->z2 = (realtype *)malloc(local_N*sizeof(realtype));
    if (check_flag((void *)data->z2, "malloc", 2, my_pe)) MPI_Abort(comm, 1);

  }

  /* Allocate and initialize backward variables */
  uB = N_VNew_Parallel(comm, local_N, NEQ+NP);
  if (check_flag((void *)uB, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  SetICback(uB, my_base);

  /* Allocate CVODES memory for the backward integration */
  flag = CVodeCreateB(cvadj_mem, CV_ADAMS, CV_FUNCTIONAL);
  if (check_flag(&flag, "CVodeCreateB", 1, my_pe)) MPI_Abort(comm, 1);
  flag = CVodeSetFdataB(cvadj_mem, data);
  if (check_flag(&flag, "CVodeSetFdataB", 1, my_pe)) MPI_Abort(comm, 1);
  flag = CVodeMallocB(cvadj_mem, fB, TOUT, uB, CV_SS, reltol, &abstol);
  if (check_flag(&flag, "CVodeMallocB", 1, my_pe)) MPI_Abort(comm, 1);

  /* Integrate to T0 */
  flag = CVodeB(cvadj_mem, T0, uB, &t, CV_NORMAL);
  if (check_flag(&flag, "CVodeB", 1, my_pe)) MPI_Abort(comm, 1);

  /* Print results (adjoint states and quadrature variables) */
  PrintOutput(g_val, uB, data);


  /* Free memory */
  N_VDestroy_Parallel(u);
  N_VDestroy_Parallel(uB);
  CVodeFree(cvode_mem);
  CVadjFree(cvadj_mem);
  if (my_pe != npes) {
    free(data->z1);
    free(data->z2);
```

```
  }
  free(data);

  MPI_Finalize();

  return(0);
}

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY CVODES
 *--------------------------------------------------------------------
 */

/*
 * f routine. Compute f(t,u) for forward phase.
 */

static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  realtype uLeft, uRight, ui, ult, urt;
  realtype hordc, horac, hdiff, hadv;
  realtype *udata, *dudata;
  long int i, my_length;
  int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
  UserData data;
  MPI_Status status;
  MPI_Comm comm;

  /* Extract MPI info. from data */
  data = (UserData) f_data;
  comm = data->comm;
  npes = data->npes;
  my_pe = data->my_pe;

  /* If this process is inactive, return now */
  if (my_pe == npes) return;

  /* Extract problem constants from data */
  hordc = data->hdcoef;
  horac = data->hacoef;

  /* Find related processes */
  my_pe_m1 = my_pe - 1;
  my_pe_p1 = my_pe + 1;
  last_pe = npes - 1;

  /* Obtain local arrays */
  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);
  my_length = NV_LOCLENGTH_P(u);
  my_last = my_length - 1;

  /* Pass needed data to processes before and after current process. */
```

```
   if (my_pe != 0)
     MPI_Send(&udata[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
   if (my_pe != last_pe)
     MPI_Send(&udata[my_length-1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);

  /* Receive needed data from processes before and after current process. */
   if (my_pe != 0)
     MPI_Recv(&uLeft, 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
   else uLeft = ZERO;
   if (my_pe != last_pe)
     MPI_Recv(&uRight, 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
              &status);
   else uRight = ZERO;

  /* Loop over all grid points in current process. */
  for (i=0; i<my_length; i++) {

    /* Extract u at x_i and two neighboring points */
    ui = udata[i];
    ult = (i==0) ? uLeft: udata[i-1];
    urt = (i==my_length-1) ? uRight : udata[i+1];

    /* Set diffusion and advection terms and load into udot */
    hdiff = hordc*(ult - TWO*ui + urt);
    hadv = horac*(urt - ult);
    dudata[i] = hdiff + hadv;
  }
}

/*
 * fB routine. Compute right hand side of backward problem
 */

static void fB(realtype t, N_Vector u,
               N_Vector uB, N_Vector uBdot, void *f_dataB)
{
  realtype *uBdata, *duBdata, *udata;
  realtype uBLeft, uBRight, uBi, uBlt, uBrt;
  realtype uLeft, uRight, ui, ult, urt;
  realtype dx, hordc, horac, hdiff, hadv;
  realtype *z1, *z2, intgr1, intgr2;
  long int i, my_length;
  int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
  UserData data;
  realtype data_in[2], data_out[2];
  MPI_Status status;
  MPI_Comm comm;

  /* Extract MPI info. from data */
  data = (UserData) f_dataB;
  comm = data->comm;
  npes = data->npes;
  my_pe = data->my_pe;

```

```
  if (my_pe == npes) { /* This process performs the quadratures */

    /* Obtain local arrays */
    duBdata = NV_DATA_P(uBdot);
    my_length = NV_LOCLENGTH_P(uB);

    /* Loop over all other processes and load right hand side of quadrature eqs. */
    duBdata[0] = ZERO;
    duBdata[1] = ZERO;
    for (i=0; i<npes; i++) {
      MPI_Recv(&intgr1, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
      duBdata[0] += intgr1;
      MPI_Recv(&intgr2, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
      duBdata[1] += intgr2;
    }

  } else { /* This process integrates part of the PDE */

    /* Extract problem constants and work arrays from data */
    dx    = data->dx;
    hordc = data->hdcoef;
    horac = data->hacoef;
    z1    = data->z1;
    z2    = data->z2;

    /* Obtain local arrays */
    uBdata = NV_DATA_P(uB);
    duBdata = NV_DATA_P(uBdot);
    udata = NV_DATA_P(u);
    my_length = NV_LOCLENGTH_P(uB);

    /* Compute related parameters. */
    my_pe_m1 = my_pe - 1;
    my_pe_p1 = my_pe + 1;
    last_pe  = npes - 1;
    my_last  = my_length - 1;

    /* Pass needed data to processes before and after current process. */
    if (my_pe != 0) {
      data_out[0] = udata[0];
      data_out[1] = uBdata[0];

      MPI_Send(data_out, 2, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
    }
    if (my_pe != last_pe) {
      data_out[0] = udata[my_length-1];
      data_out[1] = uBdata[my_length-1];

      MPI_Send(data_out, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
    }

    /* Receive needed data from processes before and after current process. */
    if (my_pe != 0) {
      MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
```

```
      uLeft = data_in[0];
      uBLeft = data_in[1];
    } else {
      uLeft = ZERO;
      uBLeft = ZERO;
    }
    if (my_pe != last_pe) {
      MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm, &status);

      uRight = data_in[0];
      uBRight = data_in[1];
    } else {
      uRight = ZERO;
      uBRight = ZERO;
    }

    /* Loop over all grid points in current process. */
    for (i=0; i<my_length; i++) {

      /* Extract uB at x_i and two neighboring points */
      uBi = uBdata[i];
      uBlt = (i==0) ? uBLeft: uBdata[i-1];
      uBrt = (i==my_length-1) ? uBRight : uBdata[i+1];

      /* Set diffusion and advection terms and load into udot */
      hdiff = hordc*(uBlt - TWO*uBi + uBrt);
      hadv = horac*(uBrt - uBlt);
      duBdata[i] = - hdiff + hadv;

      /* Extract u at x_i and two neighboring points */
      ui = udata[i];
      ult = (i==0) ? uLeft: udata[i-1];
      urt = (i==my_length-1) ? uRight : udata[i+1];

      /* Load integrands of the two space integrals */
      z1[i] = uBdata[i]*(ult - TWO*ui + urt)/(dx*dx);
      z2[i] = uBdata[i]*(urt - ult)/(TWO*dx);
    }

    /* Compute local integrals */
    intgr1 = Xintgr(z1, my_length, dx);
    intgr2 = Xintgr(z2, my_length, dx);

    /* Send local integrals to 'quadrature' process */
    MPI_Send(&intgr1, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
    MPI_Send(&intgr2, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);

  }

}

/*
 *-----------------------------------------------------------------
```

```
 * PRIVATE FUNCTIONS
 *-----------------------------------------------------------------
 */

/*
 * Set initial conditions in u vector
 */

static void SetIC(N_Vector u, realtype dx, long int my_length, long int my_base)
{
  int i;
  long int iglobal;
  realtype x;
  realtype *udata;

  /* Set pointer to data array and get local length of u */
  udata = NV_DATA_P(u);
  my_length = NV_LOCLENGTH_P(u);

  /* Load initial profile into u vector */
  for (i=1; i<=my_length; i++) {
    iglobal = my_base + i;
    x = iglobal*dx;
    udata[i-1] = x*(XMAX - x)*exp(TWO*x);
  }
}

/*
 * Set final conditions in uB vector
 */

static void SetICback(N_Vector uB, long int my_base)
{
  int i;
  realtype *uBdata;
  long int my_length;

  /* Set pointer to data array and get local length of uB */
  uBdata = NV_DATA_P(uB);
  my_length = NV_LOCLENGTH_P(uB);

  /* Set adjoint states to 1.0 and quadrature variables to 0.0 */
  if (my_base == -1) for (i=0; i<my_length; i++) uBdata[i] = ZERO;
  else               for (i=0; i<my_length; i++) uBdata[i] = ONE;
}

/*
 * Compute local value of the space integral int_x z(x) dx
 */

static realtype Xintgr(realtype *z, long int l, realtype dx)
{
  realtype my_intgr;
  long int i;
```

```

  my_intgr = RCONST(0.5)*(z[0] + z[l-1]);
  for (i = 1; i < l-1; i++)
    my_intgr += z[i];
  my_intgr *= dx;

  return(my_intgr);
}

/*
 * Compute value of g(u)
 */

static realtype Compute_g(N_Vector u, UserData data)
{
  realtype intgr, my_intgr, dx, *udata;
  long int my_length;
  int npes, my_pe, i;
  MPI_Status status;
  MPI_Comm comm;

  /* Extract MPI info. from data */
  comm = data->comm;
  npes = data->npes;
  my_pe = data->my_pe;

  dx = data->dx;

  if (my_pe == npes) {  /* Loop over all other processes and sum */
    intgr = ZERO;
    for (i=0; i<npes; i++) {
      MPI_Recv(&my_intgr, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
      intgr += my_intgr;
    }
    return(intgr);
  } else {              /* Compute local portion of the integral */
    udata = NV_DATA_P(u);
    my_length = NV_LOCLENGTH_P(u);
    my_intgr = Xintgr(udata, my_length, dx);
    MPI_Send(&my_intgr, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
    return(my_intgr);
  }
}

/*
 * Print output after backward integration
 */

static void PrintOutput(realtype g_val, N_Vector uB, UserData data)
{
  MPI_Comm comm;
  MPI_Status status;
  int npes, my_pe;
  long int i, Ni, indx, local_N, nperpe, nrem;
```

```
  realtype *uBdata;
  realtype *mu;

  comm = data->comm;
  npes = data->npes;
  my_pe = data->my_pe;
  local_N = data->local_N;
  nperpe = data->nperpe;
  nrem = data->nrem;

  uBdata = NV_DATA_P(uB);

  if (my_pe == npes) {

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("\ng(tf) = %8Le\n\n", g_val);
    printf("dgdp(tf)\n  [ 1]: %8Le\n  [ 2]: %8Le\n\n", -uBdata[0], -uBdata[1]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("\ng(tf) = %8le\n\n", g_val);
    printf("dgdp(tf)\n  [ 1]: %8le\n  [ 2]: %8le\n\n", -uBdata[0], -uBdata[1]);
#else
    printf("\ng(tf) = %8e\n\n", g_val);
    printf("dgdp(tf)\n  [ 1]: %8e\n  [ 2]: %8e\n\n", -uBdata[0], -uBdata[1]);
#endif

    mu = (realtype *)malloc(NEQ*sizeof(realtype));
    if (check_flag((void *)mu, "malloc", 2, my_pe)) MPI_Abort(comm, 1);

    indx = 0;
    for ( i = 0; i < npes; i++) {
      Ni = ( i < nrem ) ? nperpe+1 : nperpe;
      MPI_Recv(&mu[indx], Ni, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
      indx += Ni;
    }

    printf("mu(t0)\n");

#if defined(SUNDIALS_EXTENDED_PRECISION)
    for (i=0; i<NEQ; i++)
      printf("  [%2ld]: %8Le\n", i+1, mu[i]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    for (i=0; i<NEQ; i++)
      printf("  [%2ld]: %8le\n", i+1, mu[i]);
#else
    for (i=0; i<NEQ; i++)
      printf("  [%2ld]: %8e\n", i+1, mu[i]);
#endif

    free(mu);

  } else {

    MPI_Send(uBdata, local_N, PVEC_REAL_MPI_TYPE, npes, 0, comm);

```

```
  }

}

/*
 * Check function return value.
 *    opt == 0 means SUNDIALS function allocates memory so check if
 *             returned NULL pointer
 *    opt == 1 means SUNDIALS function returns a flag so check if
 *             flag >= 0
 *    opt == 2 means function allocates memory so check if returned
 *             NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
              id, funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1); }

  return(0);
}
```

# F Listing of pvakx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.11.2.1 $
 * $Date: 2005/04/01 21:55:24 $
 * -----------------------------------------------------------------
 * Programmer(s): Lukas Jager and Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Parallel Krylov adjoint sensitivity example problem.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include "mpi.h"
#include "cvodes.h"
#include "cvodea.h"
#include "cvspgmr.h"
#include "cvbbdpre.h"
#include "nvector_parallel.h"
#include "sundialstypes.h"
#include "sundialsmath.h"

/*
 *------------------------------------------------------------------
 * Constants
 *------------------------------------------------------------------
 */

#ifdef USE3D
#define DIM 3
#else
#define DIM 2
#endif

/* Domain definition */

#define XMIN RCONST(0.0)
#define XMAX RCONST(20.0)
#define MX   20    /* no. of divisions in x dir. */
#define NPX  2     /* no. of procs. in x dir.    */

#define YMIN RCONST(0.0)
#define YMAX RCONST(20.0)
#define MY   40    /* no. of divisions in y dir. */
#define NPY  2     /* no. of procs. in y dir.    */

#ifdef USE3D
#define ZMIN RCONST(0.0)
#define ZMAX RCONST(20.0)
#define MZ   20    /* no. of divisions in z dir. */
```

```
#define NPZ  1     /* no. of procs. in z dir.    */
#endif

/* Parameters for source Gaussians */

#define G1_AMPL   RCONST(1.0)
#define G1_SIGMA  RCONST(1.7)
#define G1_X      RCONST(4.0)
#define G1_Y      RCONST(8.0)
#ifdef USE3D
#define G1_Z      RCONST(8.0)
#endif

#define G2_AMPL   RCONST(0.8)
#define G2_SIGMA  RCONST(3.0)
#define G2_X      RCONST(16.0)
#define G2_Y      RCONST(12.0)
#ifdef USE3D
#define G2_Z      RCONST(12.0)
#endif

#define G_MIN     RCONST(1.0e-5)

/* Diffusion coeff., max. velocity, domain width in y dir. */

#define DIFF_COEF RCONST(1.0)
#define V_MAX     RCONST(1.0)
#define L         (YMAX-YMIN)/RCONST(2.0)
#define V_COEFF   V_MAX/L/L

/* Initial and final times */

#define ti    RCONST(0.0)
#define tf    RCONST(10.0)

/* Integration tolerances */

#define RTOL    RCONST(1.0e-8) /* states */
#define ATOL    RCONST(1.0e-6)

#define RTOL_Q  RCONST(1.0e-8) /* forward quadrature */
#define ATOL_Q  RCONST(1.0e-6)

#define RTOL_B  RCONST(1.0e-8) /* adjoint variables */
#define ATOL_B  RCONST(1.0e-6)

#define RTOL_QB RCONST(1.0e-8) /* backward quadratures */
#define ATOL_QB RCONST(1.0e-6)

/* Steps between check points */

#define STEPS 200

#define ZERO RCONST(0.0)
```

```
#define ONE  RCONST(1.0)
#define TWO  RCONST(2.0)

/*
 *-----------------------------------------------------------------
 * Macros
 *-----------------------------------------------------------------
 */

#define FOR_DIM for(dim=0; dim<DIM; dim++)

/* IJth:     (i[0],i[1],i[2])-th vector component                       */
/* IJth_ext: (i[0],i[1],i[2])-th vector component in the extended array */

#ifdef USE3D
#define IJth(y,i)     ( y[(i[0])+(l_m[0]*((i[1])+(i[2])*l_m[1]))] )
#define IJth_ext(y,i) ( y[(i[0]+1)+((l_m[0]+2)*((i[1]+1)+(i[2]+1)*(l_m[1]+2)))] )
#else
#define IJth(y,i)     (y[i[0]+(i[1])*l_m[0]])
#define IJth_ext(y,i) (y[ (i[0]+1) + (i[1]+1) * (l_m[0]+2)])
#endif

/*
 *-----------------------------------------------------------------
 * Type definition: ProblemData
 *-----------------------------------------------------------------
 */

typedef struct {
  /* Domain */
  realtype xmin[DIM];  /* "left" boundaries */
  realtype xmax[DIM];  /* "right" boundaries */
  int m[DIM];          /* number of grid points */
  realtype dx[DIM];    /* grid spacing */
  realtype dOmega;     /* differential volume */

  /* Parallel stuff */
  MPI_Comm comm;       /* MPI communicator */
  int myId;            /* process id */
  int npes;            /* total number of processes */
  int num_procs[DIM];  /* number of processes in each direction */
  int nbr_left[DIM];   /* MPI ID of "left" neighbor */
  int nbr_right[DIM];  /* MPI ID of "right" neighbor */
  int m_start[DIM];    /* "left" index in the global domain */
  int l_m[DIM];        /* number of local grid points */
  realtype *y_ext;     /* extended data array */
  realtype *buf_send;  /* Send buffer */
  realtype *buf_recv;  /* Receive buffer */
  int buf_size;        /* Buffer size */

  /* Source */
  N_Vector p;          /* Source parameters */

} *ProblemData;
```

```

/*
 *-------------------------------------------------------------------
 * Interface functions to CVODES
 *-------------------------------------------------------------------
 */

static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
static void f_local(long int Nlocal, realtype t, N_Vector y,
                    N_Vector ydot, void *f_data);

static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data);


static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
                      void *f_dataB);
static void fB_local(long int NlocalB, realtype t,
                     N_Vector y, N_Vector yB, N_Vector yBdot,
                     void *f_dataB);

static void fQB(realtype t, N_Vector y, N_Vector yB,
                N_Vector qBdot, void *fQ_dataB);

/*
 *-------------------------------------------------------------------
 * Private functions
 *-------------------------------------------------------------------
 */

static void SetData(ProblemData d, MPI_Comm comm, int npes, int myId,
                    long int *neq, long int *l_neq);
static void SetSource(ProblemData d);
static void f_comm( long int Nlocal, realtype t, N_Vector y, void *f_data);
static void Load_yext(realtype *src, ProblemData d);
static void PrintHeader();
static void PrintFinalStats(void *cvode_mem);
static void OutputGradient(int myId, N_Vector qB, ProblemData d);

/*
 *-------------------------------------------------------------------
 * Main program
 *-------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  ProblemData d;

  MPI_Comm comm;
  int npes, npes_needed;
  int myId;

  long int neq, l_neq;

```

```
  void *cvode_mem;
  N_Vector y, q;
  realtype abstol, reltol, abstolQ, reltolQ;
  void *bbdp_data;
  int mudq, mldq, mukeep, mlkeep;

  void *cvadj_mem;
  void *cvode_memB;
  N_Vector yB, qB;
  realtype abstolB, reltolB, abstolQB, reltolQB;
  int mudqB, mldqB, mukeepB, mlkeepB;

  realtype tret, *qdata, G;

  int ncheckpnt, flag;

  booleantype output;

  /* Initialize MPI and set Ids */
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm, &myId);

  /* Check number of processes */
  npes_needed = NPX * NPY;
#ifdef USE3D
  npes_needed *= NPZ;
#endif
  MPI_Comm_size(comm, &npes);
  if (npes_needed != npes) {
    if (myId == 0)
      fprintf(stderr,"I need %d processes but I only got %d\n",
              npes_needed, npes);
    MPI_Abort(comm, EXIT_FAILURE);
  }

  /* Test if matlab output is requested */
  if (argc > 1) output = TRUE;
  else          output = FALSE;

  /* Allocate and set problem data structure */
  d = (ProblemData) malloc(sizeof *d);
  SetData(d, comm, npes, myId, &neq, &l_neq);

  if (myId == 0) PrintHeader();

  /*--------------------------
    Forward integration phase
    --------------------------*/

  /* Allocate space for y and set it with the I.C. */
  y = N_VNew_Parallel(comm, l_neq, neq);
  N_VConst(ZERO, y);

```

```
  /* Allocate and initialize qB (local contributin to cost) */
  q = N_VNew_Parallel(comm, 1, npes);
  N_VConst(ZERO, q);

  /* Create CVODES object, attach user data, and allocate space */
  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  flag = CVodeSetFdata(cvode_mem, d);
  abstol = ATOL;
  reltol = RTOL;
  flag = CVodeMalloc(cvode_mem, f, ti, y, CV_SS, reltol, &abstol);

  /* Attach preconditioner and linear solver modules */
  mudq = mldq = d->l_m[0]+1;
  mukeep = mlkeep = 2;
  bbdp_data = (void *) CVBBDPrecAlloc(cvode_mem, l_neq, mudq, mldq,
                                      mukeep, mlkeep, ZERO,
                                      f_local, NULL);
  flag = CVBBDSpgmr(cvode_mem, PREC_LEFT, 0, bbdp_data);

  /* Initialize quadrature calculations */
  abstolQ = ATOL_Q;
  reltolQ = RTOL_Q;
  flag = CVodeQuadMalloc(cvode_mem, fQ, q);
  flag = CVodeSetQuadFdata(cvode_mem, d);
  flag = CVodeSetQuadErrCon(cvode_mem, TRUE, CV_SS, reltolQ, &abstolQ);

  /* Allocate space for the adjoint calculation */
  cvadj_mem = CVadjMalloc(cvode_mem, STEPS);

  /* Integrate forward in time while storing check points */
  if (myId == 0) printf("Begin forward integration... ");
  flag = CVodeF(cvadj_mem, tf, y, &tret, CV_NORMAL, &ncheckpnt);
  if (myId == 0) printf("done. ");

   /* Extract quadratures */
  flag = CVodeGetQuad(cvode_mem, tf, q);
  qdata = NV_DATA_P(q);
  MPI_Allreduce(&qdata[0], &G, 1, PVEC_REAL_MPI_TYPE, MPI_SUM, comm);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  if (myId == 0) printf("  G = %Le\n",G);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  if (myId == 0) printf("  G = %le\n",G);
#else
  if (myId == 0) printf("  G = %e\n",G);
#endif

  /* Print statistics for forward run */
  if (myId == 0) PrintFinalStats(cvode_mem);

  /*--------------------------
    Backward integration phase
    --------------------------*/

  /* Allocate and initialize yB */
```

```
  yB = N_VNew_Parallel(comm, l_neq, neq);
  N_VConst(ZERO, yB);

  /* Allocate and initialize qB (gradient) */
  qB = N_VNew_Parallel(comm, l_neq, neq);
  N_VConst(ZERO, qB);

  /* Create and allocate backward CVODE memory */
  flag = CVodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
  flag = CVodeSetFdataB(cvadj_mem, d);
  abstolB = ATOL_B;
  reltolB = RTOL_B;
  flag = CVodeMallocB(cvadj_mem, fB, tf, yB, CV_SS, reltolB, &abstolB);

  /* Attach preconditioner and linear solver modules */
  mudqB = mldqB = d->l_m[0]+1;
  mukeepB = mlkeepB = 2;
  flag = CVBBDPrecAllocB(cvadj_mem, l_neq, mudqB, mldqB,
                         mukeepB, mlkeepB, ZERO, fB_local, NULL);
  flag = CVBBDSpgmrB(cvadj_mem, PREC_LEFT, 0);

  /* Initialize quadrature calculations */
  abstolQB = ATOL_QB;
  reltolQB = RTOL_QB;
  flag = CVodeQuadMallocB(cvadj_mem, fQB, qB);
  flag = CVodeSetQuadFdataB(cvadj_mem, d);
  flag = CVodeSetQuadErrConB(cvadj_mem, TRUE, CV_SS, reltolQB, &abstolQB);

  /* Integrate backwards */
  if (myId == 0) printf("Begin backward integration... ");
  flag = CVodeB(cvadj_mem, ti, yB, &tret, CV_NORMAL);
  if (myId == 0) printf("done.\n");

  /* Print statistics for backward run */
  if (myId == 0) {
    cvode_memB = CVadjGetCVodeBmem(cvadj_mem);
    PrintFinalStats(cvode_memB);
  }

   /* Extract quadratures */
  flag = CVodeGetQuadB(cvadj_mem, qB);

  /* Process 0 collects the gradient components and prints them */
  if (output) {
    OutputGradient(myId, qB, d);
    if (myId == 0) printf("Wrote matlab file 'grad.m'.\n");
  }

  /* Free memory */
  N_VDestroy_Parallel(y);
  N_VDestroy_Parallel(q);
  N_VDestroy_Parallel(qB);
  N_VDestroy_Parallel(yB);

```

```
  CVBBDPrecFree(bbdp_data);
  CVadjFree(cvadj_mem);
  CVodeFree(cvode_mem);

  MPI_Finalize();

  return(0);
}

/*
 *------------------------------------------------------------------
 * SetData:
 * Allocate space for the ProblemData structure.
 * Set fields in the ProblemData structure.
 * Return local and global problem dimensions.
 *
 * SetSource:
 * Instantiates the source parameters for a combination of two
 * Gaussian sources.
 *------------------------------------------------------------------
 */

static void SetData(ProblemData d, MPI_Comm comm, int npes, int myId,
                    long int *neq, long int *l_neq)
{
  int n[DIM], nd[DIM];
  int dim, size;

  /* Set MPI communicator, id, and total number of processes */

  d->comm = comm;
  d->myId = myId;
  d->npes = npes;

  /* Set domain boundaries */

  d->xmin[0] = XMIN;
  d->xmax[0] = XMAX;
  d->m[0]    = MX;

  d->xmin[1] = YMIN;
  d->xmax[1] = YMAX;
  d->m[1]    = MY;

#ifdef USE3D
  d->xmin[2] = ZMIN;
  d->xmax[2] = ZMAX;
  d->m[2]    = MZ;
#endif

  /* Calculate grid spacing and differential volume */

  d->dOmega = ONE;
  FOR_DIM {
```

```
    d->dx[dim] = (d->xmax[dim] - d->xmin[dim]) / d->m[dim];
    d->m[dim] +=1;
    d->dOmega *= d->dx[dim];
  }

  /* Set partitioning */

  d->num_procs[0] = NPX;
  n[0] = NPX;
  nd[0] = d->m[0] / NPX;

  d->num_procs[1] = NPY;
  n[1] = NPY;
  nd[1] = d->m[1] / NPY;

#ifdef USE3D
  d->num_procs[2] = NPZ;
  n[2] = NPZ;
  nd[2] = d->m[2] / NPZ;
#endif

  /* Compute the neighbors */

  d->nbr_left[0]  = (myId%n[0]) == 0                ? myId : myId-1;
  d->nbr_right[0] = (myId%n[0]) == n[0]-1           ? myId : myId+1;

  d->nbr_left[1]  = (myId/n[0])%n[1] == 0           ? myId : myId-n[0];
  d->nbr_right[1] = (myId/n[0])%n[1] == n[1]-1      ? myId : myId+n[0];

#ifdef USE3D
  d->nbr_left[2]  = (myId/n[0]/n[1])%n[2] == 0      ? myId : myId-n[0]*n[1];
  d->nbr_right[2] = (myId/n[0]/n[1])%n[2] == n[2]-1 ? myId : myId+n[0]*n[1];
#endif

  /* Compute the local subdomains
     m_start: left border in global index space
     l_m:     length of the subdomain */

  d->m_start[0] = (myId%n[0])*nd[0];
  d->l_m[0]     = d->nbr_right[0] == myId ? d->m[0] - d->m_start[0] : nd[0];

  d->m_start[1] = ((myId/n[0])%n[1])*nd[1];
  d->l_m[1]     = d->nbr_right[1] == myId ? d->m[1] - d->m_start[1] : nd[1];

#ifdef USE3D
  d->m_start[2] = (myId/n[0]/n[1])*nd[2];
  d->l_m[2]     = d->nbr_right[2] == myId ? d->m[2] - d->m_start[2] : nd[2];
#endif

  /* Allocate memory for the y_ext array
     (local solution + data from neighbors) */

  size = 1;
  FOR_DIM size *= d->l_m[dim]+2;
```

```
  d->y_ext = (realtype *) malloc( size*sizeof(realtype));

  /* Initialize Buffer field.
     Size of buffer is checked when needed */

  d->buf_send = NULL;
  d->buf_recv = NULL;
  d->buf_size = 0;

  /* Allocate space for the source parameters */

  *neq = 1; *l_neq = 1;
  FOR_DIM {*neq *= d->m[dim];  *l_neq *= d->l_m[dim];}
  d->p = N_VNew_Parallel(comm, *l_neq, *neq);

  /* Initialize the parameters for a source with Gaussian profile */

  SetSource(d);

}

static void SetSource(ProblemData d)
{
  int *l_m, *m_start;
  realtype *xmin, *xmax, *dx;
  realtype x[DIM], g, *pdata;
  int i[DIM];

  l_m  = d->l_m;
  m_start = d->m_start;
  xmin = d->xmin;
  xmax = d->xmax;
  dx = d->dx;


  pdata = NV_DATA_P(d->p);

  for(i[0]=0; i[0]<l_m[0]; i[0]++) {
    x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
    for(i[1]=0; i[1]<l_m[1]; i[1]++) {
      x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
#ifdef USE3D
      for(i[2]=0; i[2]<l_m[2]; i[2]++) {
        x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];

        g = G1_AMPL
          * exp( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
          * exp( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) )
          * exp( -SQR(G1_Z-x[2])/SQR(G1_SIGMA) );

        g += G2_AMPL
          * exp( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )
          * exp( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) )
          * exp( -SQR(G2_Z-x[2])/SQR(G2_SIGMA) );
```

```
        if( g < G_MIN ) g = ZERO;

        IJth(pdata, i) = g;
      }
#else
      g = G1_AMPL
        * exp( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
        * exp( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) );

      g += G2_AMPL
        * exp( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )
        * exp( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) );

      if( g < G_MIN ) g = ZERO;

      IJth(pdata, i) = g;
#endif
    }
  }
}

/*
 *------------------------------------------------------------------
 * f_comm:
 * Function for inter-process communication
 * Used both for the forward and backward phase.
 *------------------------------------------------------------------
 */

static void f_comm(long int N_local, realtype t, N_Vector y, void *f_data)
{
  int id, n[DIM], proc_cond[DIM], nbr[DIM][2];
  ProblemData d;
  realtype *yextdata, *ydata;
  int l_m[DIM], dim;
  int c, i[DIM], l[DIM-1];
  realtype *buf_send, *buf_recv;
  MPI_Status stat;
  MPI_Comm comm;
  int dir, size = 1, small = INT_MAX;

  d  = (ProblemData) f_data;
  comm = d->comm;
  id = d->myId;

  /* extract data from domain*/
  FOR_DIM {
    n[dim] = d->num_procs[dim];
    l_m[dim] = d->l_m[dim];
  }
  yextdata = d->y_ext;
  ydata    = NV_DATA_P(y);

```

```
  /* Calculate required buffer size */
  FOR_DIM {
    size *= l_m[dim];
    if( l_m[dim] < small) small = l_m[dim];
  }
  size /= small;

  /* Adjust buffer size if necessary */
  if( d->buf_size < size ) {
    d->buf_send = (realtype*) realloc( d->buf_send, size * sizeof(realtype));
    d->buf_recv = (realtype*) realloc( d->buf_recv, size * sizeof(realtype));
    d->buf_size = size;
  }

  buf_send = d->buf_send;
  buf_recv = d->buf_recv;

  /* Compute the communication pattern; who sends first? */
  /* if proc_cond==1 , process sends first in this dimension */
  proc_cond[0] = (id%n[0])%2;
  proc_cond[1] = ((id/n[0])%n[1])%2;
#ifdef USE3D
  proc_cond[2] = (id/n[0]/n[1])%2;
#endif

  /* Compute the actual communication pattern */
  /* nbr[dim][0] is first proc to communicate with in dimension dim */
  /* nbr[dim][1] the second one */
  FOR_DIM {
    nbr[dim][proc_cond[dim]]  = d->nbr_left[dim];
    nbr[dim][!proc_cond[dim]] = d->nbr_right[dim];
  }

  /* Communication: loop over dimension and direction (left/right) */
  FOR_DIM {

    for (dir=0; dir<=1; dir++) {

      /* If subdomain at boundary, no communication in this direction */

      if (id != nbr[dim][dir]) {
        c=0;
        /* Compute the index of the boundary (right or left) */
        i[dim] = (dir ^ proc_cond[dim]) ? (l_m[dim]-1) : 0;
        /* Loop over all other dimensions and copy data into buf_send */
        l[0]=(dim+1)%DIM;
#ifdef USE3D
        l[1]=(dim+2)%DIM;
        for(i[l[1]]=0; i[l[1]]<l_m[l[1]]; i[l[1]]++)
#endif
          for(i[l[0]]=0; i[l[0]]<l_m[l[0]]; i[l[0]]++)
            buf_send[c++] = IJth(ydata, i);

        if ( proc_cond[dim] ) {
```

```
          /* Send buf_send and receive into buf_recv */
          MPI_Send(buf_send, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm);
          MPI_Recv(buf_recv, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm, &stat);
        } else {
          /* Receive into buf_recv and send buf_send*/
          MPI_Recv(buf_recv, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm, &stat);
          MPI_Send(buf_send, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm);
        }

        c=0;

        /* Compute the index of the boundary (right or left) in yextdata */
        i[dim] = (dir ^ proc_cond[dim]) ? l_m[dim] : -1;

        /* Loop over all other dimensions and copy data into yextdata */
#ifdef USE3D
        for(i[l[1]]=0; i[l[1]]<l_m[l[1]]; i[l[1]]++)
#endif
          for(i[l[0]]=0; i[l[0]]<l_m[l[0]]; i[l[0]]++)
            IJth_ext(yextdata, i) = buf_recv[c++];
      }
    } /* end loop over direction */
  } /* end loop over dimension */
}

/*
 *--------------------------------------------------------------------
 * f and f_local:
 * Forward phase ODE right-hand side
 *--------------------------------------------------------------------
 */

static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
{
  ProblemData d;
  int l_neq=1;
  int dim;

  d = (ProblemData) f_data;
  FOR_DIM l_neq *= d->l_m[dim];

  /* Do all inter-processor communication */
  f_comm(l_neq, t, y, f_data);

  /* Compute right-hand side locally */
  f_local(l_neq, t, y, ydot, f_data);
}

static void f_local(long int Nlocal, realtype t, N_Vector y,
                    N_Vector ydot, void *f_data)
{
  realtype *Ydata, *dydata, *pdata;
  realtype dx[DIM], c, v[DIM], cl[DIM], cr[DIM];
  realtype adv[DIM], diff[DIM];
```

```
  realtype xmin[DIM], xmax[DIM], x[DIM], x1;
  int i[DIM], l_m[DIM], m_start[DIM], nbr_left[DIM], nbr_right[DIM], id;
  ProblemData d;
  int dim;

  d = (ProblemData) f_data;

  /* Extract stuff from data structure */
  id = d->myId;
  FOR_DIM {
    xmin[dim]      = d->xmin[dim];
    xmax[dim]      = d->xmax[dim];
    l_m[dim]       = d->l_m[dim];
    m_start[dim]   = d->m_start[dim];
    dx[dim]        = d->dx[dim];
    nbr_left[dim]  = d->nbr_left[dim];
    nbr_right[dim] = d->nbr_right[dim];
  }

  /* Get pointers to vector data */
  dydata = NV_DATA_P(ydot);
  pdata  = NV_DATA_P(d->p);

  /* Copy local segment of y to y_ext */
  Load_yext(NV_DATA_P(y), d);
  Ydata = d->y_ext;

  /* Velocity components in x1 and x2 directions (Poiseuille profile) */
  v[1] = ZERO;
#ifdef USE3D
  v[2] = ZERO;
#endif

  /* Local domain is [xmin+(m_start+1)*dx, xmin+(m_start+1+l_m-1)*dx] */
#ifdef USE3D
  for(i[2]=0; i[2]<l_m[2]; i[2]++) {

    x[2] = xmin[2] + (m_start[2]+i[2])*dx[2];
#endif
    for(i[1]=0; i[1]<l_m[1]; i[1]++) {

      x[1] = xmin[1] + (m_start[1]+i[1])*dx[1];

      /* Velocity component in x0 direction (Poiseuille profile) */
      x1 = x[1] - xmin[1] - L;
      v[0] = V_COEFF * (L + x1) * (L - x1);

      for(i[0]=0; i[0]<l_m[0]; i[0]++) {

        x[0] = xmin[0] + (m_start[0]+i[0])*dx[0];

        c  = IJth_ext(Ydata, i);

        /* Source term*/
```

```
        IJth(dydata, i) = IJth(pdata, i);

        FOR_DIM {
          i[dim]+=1;
          cr[dim] = IJth_ext(Ydata, i);
          i[dim]-=2;
          cl[dim] = IJth_ext(Ydata, i);
          i[dim]+=1;

          /* Boundary conditions for the state variables */
          if( i[dim]==l_m[dim]-1 && nbr_right[dim]==id)
            cr[dim] = cl[dim];
          else if( i[dim]==0 && nbr_left[dim]==id )
            cl[dim] = cr[dim];

          adv[dim]  = v[dim] * (cr[dim]-cl[dim]) / (TWO*dx[dim]);
          diff[dim] = DIFF_COEF * (cr[dim]-TWO*c+cl[dim]) / SQR(dx[dim]);

          IJth(dydata, i) += (diff[dim] - adv[dim]);
        }
      }
    }
#ifdef USE3D
  }
#endif
}

/*
 *------------------------------------------------------------------
 * fQ:
 * Right-hand side of quadrature equations on forward integration.
 * The only quadrature on this phase computes the local contribution
 * to the function G.
 *------------------------------------------------------------------
 */

static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
{
  ProblemData d;
  realtype *dqdata;

  d = (ProblemData) fQ_data;

  dqdata = NV_DATA_P(qdot);

  dqdata[0] = N_VDotProd_Parallel(y,y);
  dqdata[0] *= RCONST(0.5) * (d->dOmega);
}

/*
 *------------------------------------------------------------------
 * fB and fB_local:
 * Backward phase ODE right-hand side (the discretized adjoint PDE)
 *------------------------------------------------------------------
```

```
 */

static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
               void *f_dataB)
{
  ProblemData d;
  int l_neq=1;
  int dim;

  d = (ProblemData) f_dataB;
  FOR_DIM l_neq *= d->l_m[dim];

  /* Do all inter-processor communication */
  f_comm(l_neq, t, yB, f_dataB);

  /* Compute right-hand side locally */
  fB_local(l_neq, t, y, yB, yBdot, f_dataB);
}

static void fB_local(long int NlocalB, realtype t,
                     N_Vector y, N_Vector yB, N_Vector dyB,
                     void *f_dataB)
{
  realtype *YBdata, *dyBdata, *ydata;
  realtype dx[DIM], c, v[DIM], cl[DIM], cr[DIM];
  realtype adv[DIM], diff[DIM];
  realtype xmin[DIM], xmax[DIM], x[DIM], x1;
  int i[DIM], l_m[DIM], m_start[DIM], nbr_left[DIM], nbr_right[DIM], id;
  ProblemData d;
  int dim;

  d = (ProblemData) f_dataB;

  /* Extract stuff from data structure */
  id = d->myId;
  FOR_DIM {
    xmin[dim]      = d->xmin[dim];
    xmax[dim]      = d->xmax[dim];
    l_m[dim]       = d->l_m[dim];
    m_start[dim]   = d->m_start[dim];
    dx[dim]        = d->dx[dim];
    nbr_left[dim]  = d->nbr_left[dim];
    nbr_right[dim] = d->nbr_right[dim];
  }

  dyBdata = NV_DATA_P(dyB);
  ydata   = NV_DATA_P(y);

  /* Copy local segment of yB to y_ext */
  Load_yext(NV_DATA_P(yB), d);
  YBdata = d->y_ext;

  /* Velocity components in x1 and x2 directions (Poiseuille profile) */
  v[1] = ZERO;
```

```
#ifdef USE3D
  v[2] = ZERO;
#endif

  /* local domain is [xmin+(m_start)*dx, xmin+(m_start+l_m-1)*dx] */
#ifdef USE3D
  for(i[2]=0; i[2]<l_m[2]; i[2]++) {

    x[2] = xmin[2] + (m_start[2]+i[2])*dx[2];
#endif

    for(i[1]=0; i[1]<l_m[1]; i[1]++) {

      x[1] = xmin[1] + (m_start[1]+i[1])*dx[1];

      /* Velocity component in x0 direction (Poiseuille profile) */
      x1 = x[1] - xmin[1] - L;
      v[0] = V_COEFF * (L + x1) * (L - x1);

      for(i[0]=0; i[0]<l_m[0]; i[0]++) {

        x[0] = xmin[0] + (m_start[0]+i[0])*dx[0];

        c  = IJth_ext(YBdata, i);

        /* Source term for adjoint PDE */
        IJth(dyBdata, i) = -IJth(ydata, i);

        FOR_DIM {

          i[dim]+=1;
          cr[dim] = IJth_ext(YBdata, i);
          i[dim]-=2;
          cl[dim] = IJth_ext(YBdata, i);
          i[dim]+=1;

          /* Boundary conditions for the adjoint variables */
          if( i[dim]==l_m[dim]-1 && nbr_right[dim]==id)
            cr[dim] = cl[dim]-(TWO*dx[dim]*v[dim]/DIFF_COEF)*c;
          else if( i[dim]==0 && nbr_left[dim]==id )
              cl[dim] = cr[dim]+(TWO*dx[dim]*v[dim]/DIFF_COEF)*c;

          adv[dim]  = v[dim] * (cr[dim]-cl[dim]) / (TWO*dx[dim]);
          diff[dim] = DIFF_COEF * (cr[dim]-TWO*c+cl[dim]) / SQR(dx[dim]);

          IJth(dyBdata, i) -= (diff[dim] + adv[dim]);
        }
      }
    }
#ifdef USE3D
  }
#endif
}
```

```
/*
 *--------------------------------------------------------------------
 * fQB:
 * Right-hand side of quadrature equations on backward integration
 * The i-th component of the gradient is nothing but int_t yB_i dt
 *--------------------------------------------------------------------
 */

static void fQB(realtype t, N_Vector y, N_Vector yB, N_Vector qBdot,
                void *fQ_dataB)
{
  ProblemData d;

  d = (ProblemData) fQ_dataB;

  N_VScale_Parallel(-(d->dOmega), yB, qBdot);
}

/*
 *--------------------------------------------------------------------
 * Load_yext:
 * copies data from src (y or yB) into y_ext, which already contains
 * data from neighboring processes.
 *--------------------------------------------------------------------
 */

static void Load_yext(realtype *src, ProblemData d)
{
  int i[DIM], l_m[DIM], dim;

  FOR_DIM l_m[dim] = d->l_m[dim];

  /* copy local segment */
#ifdef USE3D
  for  (i[2]=0; i[2]<l_m[2]; i[2]++)
#endif
    for(i[1]=0; i[1]<l_m[1]; i[1]++)
      for(i[0]=0; i[0]<l_m[0]; i[0]++)
        IJth_ext(d->y_ext, i) = IJth(src, i);
}

/*
 *--------------------------------------------------------------------
 * PrintHeader:
 * Print first lins of output (problem description)
 *--------------------------------------------------------------------
 */

static void PrintHeader()
{
    printf("\nParallel Krylov adjoint sensitivity analysis example\n");
    printf("%1dD Advection diffusion PDE with homogeneous Neumann B.C.\n",DIM);
    printf("Computes gradient of G = int_t_Omega ( c_i^2 ) dt dOmega\n");
    printf("with respect to the source values at each grid point.\n\n");
```

```

    printf("Domain:\n");

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("   %Lf < x < %Lf   mx = %d  npe_x = %d \n",XMIN,XMAX,MX,NPX);
    printf("   %Lf < y < %Lf   my = %d  npe_y = %d \n",YMIN,YMAX,MY,NPY);
#else
    printf("   %f < x < %f   mx = %d  npe_x = %d \n",XMIN,XMAX,MX,NPX);
    printf("   %f < y < %f   my = %d  npe_y = %d \n",YMIN,YMAX,MY,NPY);
#endif

#ifdef USE3D
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("   %Lf < z < %Lf   mz = %d  npe_z = %d \n",ZMIN,ZMAX,MZ,NPZ);
#else
    printf("   %f < z < %f   mz = %d  npe_z = %d \n",ZMIN,ZMAX,MZ,NPZ);
#endif
#endif

    printf("\n");
  }

/*
 *------------------------------------------------------------------
 * PrintFinalStats:
 * Print final statistics contained in cvode_mem
 *------------------------------------------------------------------
 */

static void PrintFinalStats(void *cvode_mem)
{
  long int lenrw, leniw ;
  long int lenrwSPGMR, leniwSPGMR;
  long int nst, nfe, nsetups, nni, ncfn, netf;
  long int nli, npe, nps, ncfl, nfeSPGMR;
  int flag;

  flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
  flag = CVodeGetNumSteps(cvode_mem, &nst);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);

  flag = CVSpgmrGetWorkSpace(cvode_mem, &lenrwSPGMR, &leniwSPGMR);
  flag = CVSpgmrGetNumLinIters(cvode_mem, &nli);
  flag = CVSpgmrGetNumPrecEvals(cvode_mem, &npe);
  flag = CVSpgmrGetNumPrecSolves(cvode_mem, &nps);
  flag = CVSpgmrGetNumConvFails(cvode_mem, &ncfl);
  flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfeSPGMR);

  printf("\nFinal Statistics.. \n\n");
  printf("lenrw   = %6ld     leniw = %6ld\n", lenrw, leniw);
```

```
  printf("llrw    = %6ld     lliw  = %6ld\n", lenrwSPGMR, leniwSPGMR);
  printf("nst     = %6ld\n"                   , nst);
  printf("nfe     = %6ld     nfel  = %6ld\n"  , nfe, nfeSPGMR);
  printf("nni     = %6ld     nli   = %6ld\n"  , nni, nli);
  printf("nsetups = %6ld     netf  = %6ld\n"  , nsetups, netf);
  printf("npe     = %6ld     nps   = %6ld\n"  , npe, nps);
  printf("ncfn    = %6ld     ncfl  = %6ld\n\n", ncfn, ncfl);
}

/*
 *------------------------------------------------------------------
 * OutputGradient:
 * Generate matlab m files for visualization
 * One file gradXXXX.m from each process + a driver grad.m
 *------------------------------------------------------------------
 */

static void OutputGradient(int myId, N_Vector qB, ProblemData d)
{
  FILE *fid;
  char filename[20];
  int *l_m, *m_start, i[DIM],ip;
  realtype *xmin, *xmax, *dx;
  realtype x[DIM], *pdata, p, *qBdata, g;

  sprintf(filename,"grad%03d.m",myId);
  fid = fopen(filename,"w");

  l_m  = d->l_m;
  m_start = d->m_start;
  xmin = d->xmin;
  xmax = d->xmax;
  dx = d->dx;

  qBdata = NV_DATA_P(qB);
  pdata  = NV_DATA_P(d->p);

  /* Write matlab files with solutions from each process */

  for(i[0]=0; i[0]<l_m[0]; i[0]++) {
    x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
    for(i[1]=0; i[1]<l_m[1]; i[1]++) {
      x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
#ifdef USE3D
      for(i[2]=0; i[2]<l_m[2]; i[2]++) {
        x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];
        g = IJth(qBdata, i);
        p = IJth(pdata, i);
#if defined(SUNDIALS_EXTENDED_PRECISION)
        fprintf(fid,"x%d(%d,1) = %Le; \n",  myId, i[0]+1,         x[0]);
        fprintf(fid,"y%d(%d,1) = %Le; \n",  myId, i[1]+1,         x[1]);
        fprintf(fid,"z%d(%d,1) = %Le; \n",  myId, i[2]+1,         x[2]);
        fprintf(fid,"p%d(%d,%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, i[2]+1, p);
        fprintf(fid,"g%d(%d,%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, i[2]+1, g);
```

```
#elif defined(SUNDIALS_DOUBLE_PRECISION)
        fprintf(fid,"x%d(%d,1) = %le; \n",  myId, i[0]+1,         x[0]);
        fprintf(fid,"y%d(%d,1) = %le; \n",  myId, i[1]+1,         x[1]);
        fprintf(fid,"z%d(%d,1) = %le; \n",  myId, i[2]+1,         x[2]);
        fprintf(fid,"p%d(%d,%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, i[2]+1, p);
        fprintf(fid,"g%d(%d,%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, i[2]+1, g);
#else
        fprintf(fid,"x%d(%d,1) = %e; \n",  myId, i[0]+1,         x[0]);
        fprintf(fid,"y%d(%d,1) = %e; \n",  myId, i[1]+1,         x[1]);
        fprintf(fid,"z%d(%d,1) = %e; \n",  myId, i[2]+1,         x[2]);
        fprintf(fid,"p%d(%d,%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, i[2]+1, p);
        fprintf(fid,"g%d(%d,%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, i[2]+1, g);
#endif
      }
#else
      g = IJth(qBdata, i);
      p = IJth(pdata, i);
#if defined(SUNDIALS_EXTENDED_PRECISION)
      fprintf(fid,"x%d(%d,1) = %Le; \n",  myId, i[0]+1,         x[0]);
      fprintf(fid,"y%d(%d,1) = %Le; \n",  myId, i[1]+1,         x[1]);
      fprintf(fid,"p%d(%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, p);
      fprintf(fid,"g%d(%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, g);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
      fprintf(fid,"x%d(%d,1) = %le; \n",  myId, i[0]+1,         x[0]);
      fprintf(fid,"y%d(%d,1) = %le; \n",  myId, i[1]+1,         x[1]);
      fprintf(fid,"p%d(%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, p);
      fprintf(fid,"g%d(%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, g);
#else
      fprintf(fid,"x%d(%d,1) = %e; \n",  myId, i[0]+1,         x[0]);
      fprintf(fid,"y%d(%d,1) = %e; \n",  myId, i[1]+1,         x[1]);
      fprintf(fid,"p%d(%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, p);
      fprintf(fid,"g%d(%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, g);
#endif
#endif
    }
  }
  fclose(fid);

  /* Write matlab driver */

  if (myId == 0) {

    fid = fopen("grad.m","w");

#ifdef USE3D
    fprintf(fid,"clear;\nfigure;\nhold on\n");
    fprintf(fid,"trans = 0.7;\n");
    fprintf(fid,"ecol  = 'none';\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
    fprintf(fid,"xp=[%Lf %Lf];\n",G1_X,G2_X);
    fprintf(fid,"yp=[%Lf %Lf];\n",G1_Y,G2_Y);
    fprintf(fid,"zp=[%Lf %Lf];\n",G1_Z,G2_Z);
#else
    fprintf(fid,"xp=[%f %f];\n",G1_X,G2_X);
```

```
    fprintf(fid,"yp=[%f %f];\n",G1_Y,G2_Y);
    fprintf(fid,"zp=[%f %f];\n",G1_Z,G2_Z);
#endif
    fprintf(fid,"ns = length(xp)*length(yp)*length(zp);\n");

    for (ip=0; ip<d->npes; ip++) {
      fprintf(fid,"\ngrad%03d;\n",ip);
      fprintf(fid,"[X,Y,Z]=meshgrid(x%d,y%d,z%d);\n",ip,ip,ip);
      fprintf(fid,"s%d=slice(X,Y,Z,g%d,xp,yp,zp);\n",ip,ip);
      fprintf(fid,"for i = 1:ns\n");
      fprintf(fid,"  set(s%d(i),'FaceAlpha',trans);\n",ip);
      fprintf(fid,"  set(s%d(i),'EdgeColor',ecol);\n",ip);
      fprintf(fid,"end\n");
    }

    fprintf(fid,"view(3)\n");
    fprintf(fid,"\nshading interp\naxis equal\n");
#else
    fprintf(fid,"clear;\nfigure;\n");
    fprintf(fid,"trans = 0.7;\n");
    fprintf(fid,"ecol  = 'none';\n");

    for (ip=0; ip<d->npes; ip++) {

      fprintf(fid,"\ngrad%03d;\n",ip);

      fprintf(fid,"\nsubplot(1,2,1)\n");
      fprintf(fid,"s=surf(x%d,y%d,g%d);\n",ip,ip,ip);
      fprintf(fid,"set(s,'FaceAlpha',trans);\n");
      fprintf(fid,"set(s,'EdgeColor',ecol);\n");
      fprintf(fid,"hold on\n");
      fprintf(fid,"axis tight\n");
      fprintf(fid,"box on\n");

      fprintf(fid,"\nsubplot(1,2,2)\n");
      fprintf(fid,"s=surf(x%d,y%d,p%d);\n",ip,ip,ip);
      fprintf(fid,"set(s,'CData',g%d);\n",ip);
      fprintf(fid,"set(s,'FaceAlpha',trans);\n");
      fprintf(fid,"set(s,'EdgeColor',ecol);\n");
      fprintf(fid,"hold on\n");
      fprintf(fid,"axis tight\n");
      fprintf(fid,"box on\n");

    }
#endif
    fclose(fid);
  }
}
```