

Example Programs for CVODES v2.4.0

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

November 6, 2006



UCRL-SM-208115

DISCLAIMER

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

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	1
2	Forward sensitivity analysis example problems	5
2.1	A serial nonstiff example: cvsfwdnonx	5
2.2	A serial dense example: cvsfwddenx	10
2.3	An SPGMR parallel example with user preconditioner: cvsfwdkryx_p	17
3	Adjoint sensitivity analysis example problems	24
3.1	A serial dense example: cvsadjdenx	24
3.2	A parallel nonstiff example: cvsadjnonx_p	28
3.3	An SPGMR parallel example using the CVBBDPRE module: cvsadjkryx_p	31
4	Parallel tests	35
	References	37
A	Listing of cvsfwdnonx.c	38
B	Listing of cvsfwddenx.c	47
C	Listing of cvsfwdkryx_p.c	58
D	listing of cvsadjdenx.c	81
E	Listing of cvsadjnonx_p.c	92
F	Listing of cvsadjkryx_p.c	104

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 CVODES distribution contains examples of the following types: serial and parallel examples of Initial Value Problem (IVP) integration, serial and parallel examples of forward sensitivity analysis (FSA), and serial and parallel examples of adjoint sensitivity analysis (ASA). These examples, listed in the table below, are briefly described next.

	Serial examples	Parallel examples
IVP	<code>cvsdex</code> <code>cvsdex_uw</code> <code>cvsbex</code> <code>cvsdirectdem</code> <code>cvskryx</code> <code>cvskryx_bp</code> <code>cvskrydem_lin</code> <code>cvskrydem_pre</code>	<code>cvsnex_p</code> <code>cvskryx_p</code> <code>cvskryx_bbd_p</code>
FSA	<code>cvsfwdex</code> <code>cvsfwdkryx</code> <code>cvsfwdnex</code>	<code>cvsfwdnex_p</code> <code>cvsfwdkryx_p</code>
ASA	<code>cvsadjex</code> <code>cvsadjbex</code> <code>cvsadjkryx_int</code> <code>cvsadjkryx_intb</code>	<code>cvsadjnex_p</code> <code>cvsadjkryx_p</code>

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

- `cvsdex` 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.
- `cvsdex_uw` is the same as `cvsdex` but demonstrates the user-supplied error weight function feature of CVODES.
- `cvsbex` 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.
- `cvskryx` 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.
- `cvskryx_bp` solves the same problem as `cvskryx`, with the BDF/GMRES method and a banded preconditioner, generated by difference quotients, using the module `CVBAND-PRE`.
The problem is solved twice: with preconditioning on the left, then on the right.
- `cvskrydem_lin` solves the same problem as `cvskryx`, with the BDF method, but with three Krylov linear solvers: `CVSPGMR`, `CVSPBCG`, and `CVSPTFQMR`.

- **cvsdirectdem** 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 system 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.

- **cvskrydem_pre** 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 preconditioner is applied on the left and on the right. In each case, both the modified and classical Gram-Schmidt options are tested.

- **cvsfwddenx** solves a 3-species chemical kinetics problem (from **cvsdex**).

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.

- **cvsfwdkryx** solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D space (from **cvskryx**).

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.

- **cvsfwdnonx** 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.

- **cvsadjdenx** solves a 3-species chemical kinetics problem (from **cvsdex**).

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.

- **cvlsadjbanx** solves a semi-discrete 2-D advection-diffusion equation (from **cvlsbanx**).

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.

- **cvlsadjkryx_int** solves a stiff ODE system that arises from a system of partial differential equations (from **cvlskrydem_pre**). 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.

- **cvlsadjkryx_intb** solves the same problem as **cvlsadjkryx_int**, 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/examples/cvodes/parallel` directory are the following six parallel examples (using the `NVECTOR_PARALLEL` module):

- **cvlsnonx_p** 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.

- **cvlskryx_p** is the parallel implementation of **cvlskryx**.
- **cvlskryx_bbd_p** solves the same problem as **cvlskryx_p**, with the BDF/GMRES method and a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the module `CVBBDPRE`.
- **cvlsfwdnonx_p** is the parallel version of **cvlsfwdnonx**.
- **cvlsfwdkryx_p** is the parallel version of **cvlsfwdkryx**.
- **cvlsadjnonx_p** solves a semi-discrete 1-D advection-diffusion equation (from **cvlsnonx_p**).

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.

- `cvlsadjkryx_p` 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 the examples for IVP integration; for those, the interested reader should consult the CVODE Examples document [1]. Any CVODE problem will work with CVODES with only two modifications: (1) the main program should include the header file `cvodes.h` instead of `cvode.h`, and (2) the loader command must reference *builddir/lib/libsundials_cvodes.lib* instead of *builddir/lib/libsundials_cvode.lib*.

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 `cvskryx_p` 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 CVODES examples, any of three sensitivity method options (`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 three sections give detailed descriptions of two serial examples (`cvsfwdnonx` and `cvsfwdnmx`), and a parallel one (`cvsfwdkryx_p`). For details on the other examples, the reader is directed to the comments in their source files.

2.1 A serial nonstiff example: `cvsfwdnonx`

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

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

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

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

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 $\Delta x = 2/(\text{MX} + 1)$, the resulting system of ODEs, $\dot{u} = f(t, u)$, can now be written:

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

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

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

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

and

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

The source file for this problem, `cvsfwdnonx.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 `cvsfwdnonx.c` file begins by including several header files, including the main CVODES header file, the `sundials_types.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 `NVNewSerial`) 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 `NVCloneVectorArraySerial` 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 Eqn. (3). The rest of the source file `cvsfwdnonx.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 `cvsfwdnonx` are shown in Fig. 1. The output generated by `cvsfwdnonx` when computing sensitivities with the `CV_SIMULTANEOUS` method and full error control (`cvsfwdnonx -sensi sim t`) is:

cvsfwdnonx 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
				Sensitivity 1	3.8668e+00

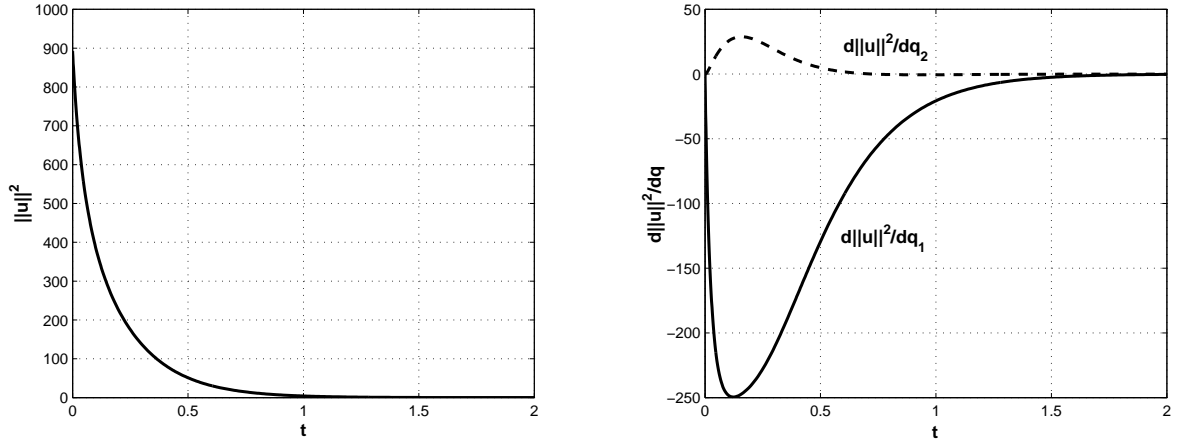


Figure 1: Results for the `cvsfwdnonx` example problem. The time evolution of the squared solution norm, $\|u\|^2$, is shown on the left. The figure on the right shows the evolution of the sensitivities of $\|u\|^2$ with respect to the two problem parameters.

				Sensitivity 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      =   1373
netf     =     1   nsetups   =     0
nni      =   1369   ncfn     =    117

nfSe     =   2746   nfeS     =   5492
netfs    =     0   nsetupsS  =     0
nniS     =     0   ncfnS     =     0

```

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

```

----- cvsfwdnonx 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	=	3158		
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: `cvsfwddenx`

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

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

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 `cvstdenx` 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 `cvstdenx`, 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 `cvsfwddenx` 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 right-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 `cvsfwddenx`, 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 called 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 `cvsfwddenx`.

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. `cvsfwddenx` 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 `cvsdexn.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` ($pbar_i = p_i$, which can typically be used for nonzero model parameters). Next, the program allocates memory for `yS`, by calling the `NVECTOR_SERIAL` function `N_VCloneVectorArray_Serial`, and initializes all sensitivity variables to 0.0.

The call to `CVodeSensMalloc` specifies the sensitivity solution method through the argument `sensi_meth` (read from the command line arguments) as one of `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 sensitivity variables tolerances. Neither `p` nor `plist` are required since the sensitivity right-hand sides are computed in a user-provided function (`fS`). As a consequence, 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 `cvsdexn.c` with the notable exception 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 `cvsdexn.c`. The user-supplied function `fS` computes the sensitivity right-hand side for the

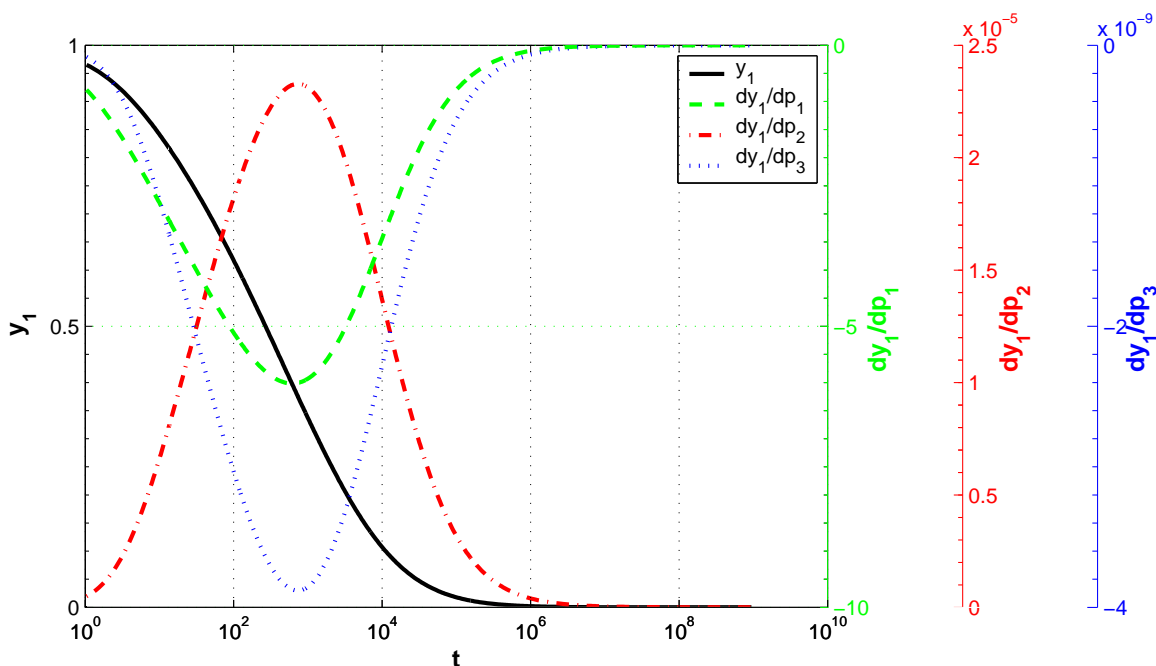


Figure 2: Results for the `cvsfwddenx` example problem: time evolution of y_1 and its sensitivities with respect to the three problem parameters.

iS-th sensitivity equation.

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

```
% cvsfwddenx -nosensi
```

if no sensitivity calculations are desired, or

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

cvsfwddenx 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	=	1222					
netf	=	29	nsetups	=	142		
nni	=	1218	ncfn	=	4		
nfSe	=	3666	nfeS	=	0		
netfs	=	0	nsetupsS	=	0		
nniS	=	0	ncfnS	=	0		
nje	=	24	nfeLS	=	0		

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

----- cvswfddenx 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	=	1380		
netf	=	25	nsetups	= 109
nni	=	797	ncfn	= 0

<code>nfSe</code>	<code>=</code>	<code>2829</code>	<code>nfeS</code>	<code>=</code>	<code>0</code>
<code>netfs</code>	<code>=</code>	<code>0</code>	<code>nsetupsS</code>	<code>=</code>	<code>3</code>
<code>nniS</code>	<code>=</code>	<code>942</code>	<code>ncfnS</code>	<code>=</code>	<code>0</code>
<code>nje</code>	<code>=</code>	<code>11</code>	<code>nfeLS</code>	<code>=</code>	<code>0</code>

2.3 An SPGMR parallel example with user preconditioner: cvsfwdkryx_p

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

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

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

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

The spatial domain is $0 \leq x \leq 20$, $30 \leq y \leq 50$ (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:

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

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

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

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 $\text{MXSUB} \times \text{MYSUB}$ 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 $\text{MX} \times \text{MY}$ with $\text{MX} = \text{NPEX} \times \text{MXSUB}$ and $\text{MY} = \text{NPEY} \times \text{MYSUB}$, and the size of the ODE system is $2 \cdot \text{MX} \cdot \text{MY}$.

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

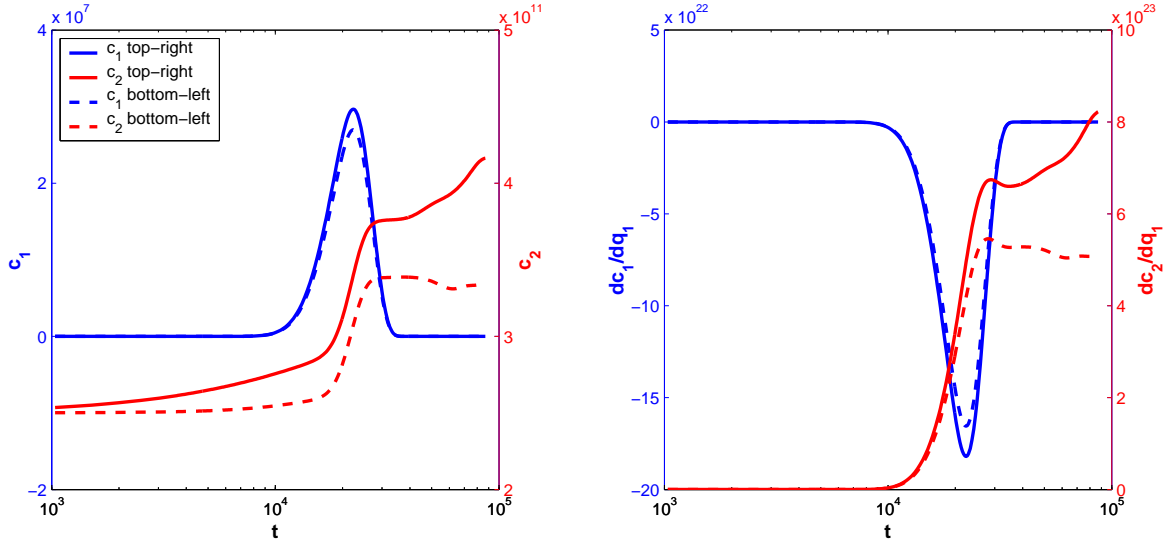


Figure 3: Results for the `cvsfwdkryx_p` example problem: time evolution of c_1 and c_2 at the bottom-left and top-right corners (left) and of their sensitivities with respect to q_1 .

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 `cvsfwddenx` described above with differences arising from the use of the parallel `NVECTOR` module - `NVECTOR_PARALLEL`. On the other hand, the user-supplied routines in `cvsfwdkryx_p`, `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 `cvskryx_p` 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 `cvsfwdkryx_p` 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 `cvsfwdkryx_p` are shown in Fig. 3. These results were generated on a $(2 \times 40) \times (2 \times 40)$ grid.

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

```
% mpirun -np nproc cvsfwdkryx\_p -nosensi
```

if no sensitivity calculations are desired, or

```
% mpirun -np nproc cvsfwdkryx\_p -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 following output is generated by `cvsfwdkryx_p` when computing sensitivities with the `CV_SIMULTANEOUS` method and full error control (`mpirun -np 4 cvsfwdkryx_p -sensi sim t`):

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

T	Q	H	NST		Bottom left	Top right
7.200e+03	3	3.544e+01	415			
				Solution	1.0468e+04 2.5267e+11	1.1185e+04 2.6998e+11
				Sensitivity 1	-6.4201e+19 7.1177e+19	-6.8598e+19 7.6556e+19
				Sensitivity 2	-4.3853e+14 -2.4407e+18	-5.0065e+14 -2.7842e+18
1.440e+04	3	4.903e+01	640			
				Solution	6.6590e+06 2.5819e+11	7.3008e+06 2.8329e+11
				Sensitivity 1	-4.0848e+22 5.9550e+22	-4.4785e+22 6.7173e+22
				Sensitivity 2	-4.5235e+17 -6.5419e+21	-5.4318e+17 -7.8316e+21
2.160e+04	2	2.948e+01	1198			
				Solution	2.6650e+07 2.9928e+11	2.9308e+07 3.3134e+11
				Sensitivity 1	-1.6346e+23 3.8203e+23	-1.7976e+23 4.4991e+23
				Sensitivity 2	-7.6601e+18 -7.6459e+22	-9.4433e+18 -9.4502e+22
2.880e+04	3	4.499e+01	1542			
				Solution	8.7021e+06 3.3804e+11	9.6501e+06 3.7510e+11
				Sensitivity 1	-5.3375e+22 5.4487e+23	-5.9188e+22 6.7430e+23
				Sensitivity 2	-4.8855e+18 -1.7194e+23	-6.1040e+18 -2.1518e+23
3.600e+04	3	1.107e+01	1675			
				Solution	1.4040e+04 3.3868e+11	1.5609e+04 3.7652e+11
				Sensitivity 1	-8.6141e+19 5.2718e+23	-9.5761e+19 6.6030e+23
				Sensitivity 2	-8.4328e+15 -1.8439e+23	-1.0549e+16 -2.3096e+23

4.320e+04	4	8.016e+01	3146			
				Solution	-2.5802e-07	-2.3679e-07
					3.3823e+11	3.8035e+11

				Sensitivity 1	-3.4238e+08	-3.8876e+08
					5.2753e+23	6.7448e+23

				Sensitivity 2	5.4404e+03	6.6262e+03
					-1.8454e+23	-2.3595e+23

5.040e+04	4	2.431e+02	3194			
				Solution	8.0708e-08	7.0502e-08
					3.3582e+11	3.8645e+11

				Sensitivity 1	-4.5550e+10	-3.9697e+10
					5.2067e+23	6.9664e+23

				Sensitivity 2	-5.5727e+08	-5.5845e+08
					-1.8214e+23	-2.4371e+23

5.760e+04	4	2.523e+02	3216			
				Solution	-1.8957e-11	-1.6762e-11
					3.3203e+11	3.9090e+11

				Sensitivity 1	-1.5940e+06	-1.4035e+06
					5.0825e+23	7.1205e+23

				Sensitivity 2	-2.2867e+02	-2.2722e+02
					-1.7780e+23	-2.4910e+23

6.480e+04	4	2.820e+02	3254			
				Solution	-7.8717e-10	-6.7017e-10
					3.3130e+11	3.9634e+11

				Sensitivity 1	-3.4373e+07	-2.1671e+07
					5.0442e+23	7.3274e+23

				Sensitivity 2	1.3198e+08	1.3575e+08
					-1.7646e+23	-2.5633e+23

7.200e+04	4	3.810e+02	3276			
				Solution	-4.4006e-09	-3.7463e-09
					3.3297e+11	4.0389e+11

				Sensitivity 1	4.5106e+09	3.8200e+09
					5.0783e+23	7.6382e+23

				Sensitivity 2	2.0264e+05	2.0791e+05
					-1.7765e+23	-2.6721e+23

7.920e+04	5	6.406e+02	3291			
				Solution	-1.8328e-11	-1.5666e-11
					3.3344e+11	4.1203e+11

				Sensitivity 1	1.3650e+07	1.1729e+07
					5.0730e+23	7.9960e+23

				Sensitivity 2	6.1764e+01	6.8476e+01
					-1.7747e+23	-2.7972e+23


```

-----
8.640e+04  5  6.406e+02  3302
Solution      -2.0206e-13  -1.7557e-13
              3.3518e+11  4.1625e+11
-----
Sensitivity 1   1.1323e+06  9.7319e+05
              5.1171e+23  8.2142e+23
-----
Sensitivity 2   7.6632e+00  8.2818e+00
              -1.7901e+23 -2.8736e+23
-----

Final Statistics

nst      = 3302

nfe      = 4387
netf     = 165   nsetups = 508
nni      = 4383  ncfn    = 7

nfSe     = 8774   nfeS    = 17548
netfs    = 0      nsetupsS = 0
nniS     = 0      ncfnS   = 0

```

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

```

----- cvswfdkryx_p 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	2.041e+02	306	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	8.953e+01	347	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	4.739e+02	410	Solution	3.0957e-07	3.4673e-07
					3.3823e+11	3.8035e+11
				Sensitivity 1	-3.4875e+11	-4.3058e+11
					5.2753e+23	6.7448e+23
				Sensitivity 2	-1.0432e+08	1.3672e+08
					-1.8454e+23	-2.3595e+23
5.040e+04	4	3.618e+02	427	Solution	-9.8547e-08	-1.1038e-07
					3.3582e+11	3.8644e+11
				Sensitivity 1	5.0030e+10	6.1933e+10
					5.2067e+23	6.9664e+23
				Sensitivity 2	-2.0381e+07	2.2220e+07
					-1.8214e+23	-2.4370e+23
5.760e+04	5	4.519e+02	440	Solution	4.7038e-07	5.2625e-07
					3.3203e+11	3.9090e+11
				Sensitivity 1	-2.4178e+09	-2.9454e+09
					5.0825e+23	7.1205e+23
				Sensitivity 2	1.3338e+09	-1.5676e+09
					-1.7780e+23	-2.4910e+23

6.480e+04	5	7.147e+02	450			
				Solution	-3.4380e-07	-3.8470e-07
					3.3130e+11	3.9634e+11

				Sensitivity 1	-6.7941e+08	-9.7442e+08
					5.0442e+23	7.3274e+23

				Sensitivity 2	-3.6754e+07	4.3312e+07
					-1.7646e+23	-2.5633e+23

7.200e+04	5	7.147e+02	461			
				Solution	5.1980e-09	5.8167e-09
					3.3297e+11	4.0388e+11

				Sensitivity 1	-4.0025e+09	-4.6921e+09
					5.0783e+23	7.6382e+23

				Sensitivity 2	8.7605e+07	-1.0328e+08
					-1.7765e+23	-2.6721e+23

7.920e+04	5	7.147e+02	471			
				Solution	1.0510e-10	1.1755e-10
					3.3344e+11	4.1203e+11

				Sensitivity 1	2.5316e+08	2.9711e+08
					5.0730e+23	7.9960e+23

				Sensitivity 2	8.1353e+05	-9.5897e+05
					-1.7747e+23	-2.7972e+23

8.640e+04	5	7.147e+02	481			
				Solution	-2.7924e-11	-3.1239e-11
					3.3518e+11	4.1625e+11

				Sensitivity 1	-1.1922e+07	-1.4044e+07
					5.1171e+23	8.2142e+23

				Sensitivity 2	-3.8361e+05	4.5224e+05
					-1.7901e+23	-2.8736e+23

Final Statistics						
nst	=	481				
nfe	=	1107				
netf	=	28	nsetups	=	81	
nni	=	622	ncfn	=	1	
nfSe	=	1216	nfeS	=	2432	
netfs	=	0	nsetupsS	=	0	
nniS	=	607	ncfnS	=	0	

3 Adjoint sensitivity analysis example problems

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

3.1 A serial dense example: `cvlsadjdenx`

As a first example of using CVODES for adjoint sensitivity analysis we examine the chemical kinetics problem (from `cvlsfwddenx`)

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

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

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

where $g(t, y, p) = y_3$, f is the vector-valued function defining the right-hand side of (11), and λ 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 λ 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 ζ 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` for CVODES definitions and interface function prototypes, the header file `cvodes_dense.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 `sundials_math.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 `cvlsadjdenx` 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 accomplished through the call to `CVadjMalloc` which specifies `STEPS = 150`, the number of steps between two checkpoints, and specifies cubic Hermite interpolation.

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 obtained by calling `CVadjGetCheckPointsInfo` and the checkpoint information printed).

The next segment of code deals with the setup of the backward problem. First, a serial vector `yB` of length `NEQ` is allocated and initialized with the value of λ 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 ξ 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` and `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 ξ 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 accomplished 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 `cvsdex.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 `cvسادجنخ` is shown below.

```

----- cvسادجنخ 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
Forward integration ... done ( nst = 810 )
-----
G:                3.9983e+07
-----

Create and allocate CVODES memory for backward run
Backward integration ... done ( nst = 230 )
-----
tB0:                4.0000e+07
dG/dp:              7.6842e+05   -3.0691e+00   5.1145e-04
lambda(t0):         3.9967e+07   3.9967e+07   3.9967e+07
-----

```

```

Re-initialize CVODES memory for backward run
Backward integration ... done ( nst = 197 )
-----
tB0:          5.0000e+01
dG/dp:        1.7341e+02  -5.0591e-04  8.4323e-08
lambda(t0):    8.4191e+00  1.6097e+01  1.6097e+01
-----

Free memory

```

3.2 A parallel nonstiff example: `cvsadjnonx_p`

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

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

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 δu_0 in u_0 .

The approach we take in the program `cvsadjnonx_p` 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 μ 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 δu_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 `cvlsadjdenx` discussed previously with differences arising from the use of the parallel `NVECTOR` module. Unlike `cvlsadjdenx`, the example `cvlsadjnonx.p` 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, `cvlsadjnonx.p` 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 `cvlsadjnonx.p` has the advantage that the communication strategy adopted for the forward problem can be directly transferred 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 proceses, 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 `cvlsadjnonx.p`, for `MX = 20`, is shown below.

`cvlsadjnonx.p sample output`

```

g(tf) = 2.129919e-02

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

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

```

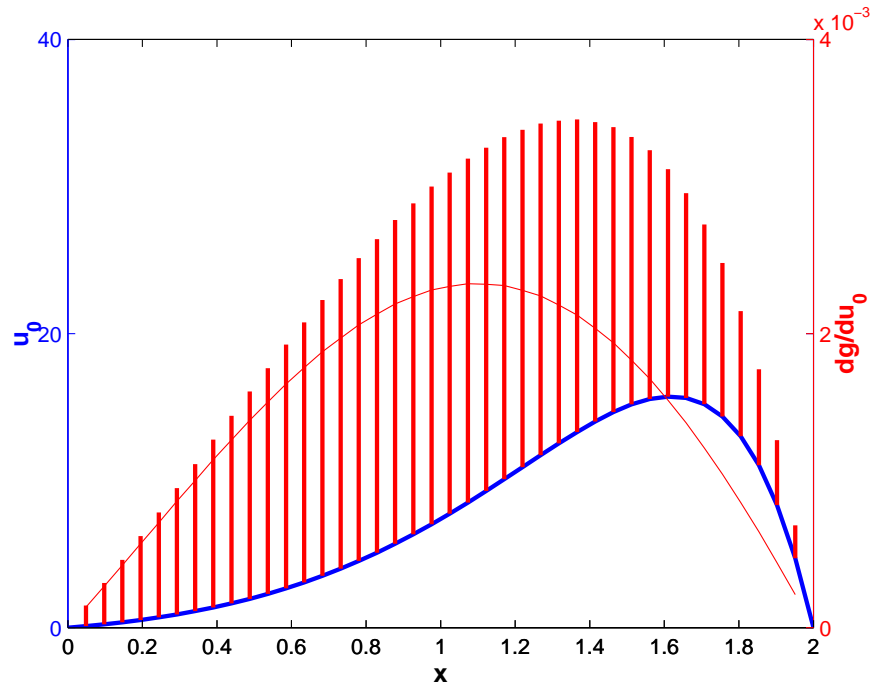


Figure 4: Results for the `cvsadjnonx_p` example problem. The gradient of $g(t_f)$ with respect to the initial conditions u_0 is shown superimposed over the values u_0 .

```
[ 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: cvsadjkryx_p

As a more elaborated adjoint sensitivity parallel example we describe next the `cvsadjkryx_p` 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 concentration $c(t, x)$ in a domain Ω is given by

$$\begin{aligned} \frac{\partial c}{\partial t} - k \nabla^2 c + v \cdot \nabla c + f &= 0, \text{ in } (0, T) \times \Omega \\ \frac{\partial c}{\partial n} &= g, \text{ on } (0, T) \times \partial\Omega \\ c &= c_0(x), \text{ in } \Omega \text{ at } t = 0, \end{aligned} \quad (22)$$

where Ω is a box in \mathbb{R}^2 or \mathbb{R}^3 and n is the normal to the boundary of Ω . We assume homogeneous boundary conditions ($g = 0$) and a zero initial concentration everywhere in Ω ($c_0(x) = 0$). 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

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

is obtained as

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

where x_i is the location of the source of intensity p_i and λ is solution of the adjoint PDE

$$\begin{aligned} -\frac{\partial \lambda}{\partial t} - k \nabla^2 \lambda - v \cdot \nabla \lambda &= c(t, x), \text{ in } (T, 0) \times \Omega \\ (k \nabla \lambda + v \lambda) \cdot n &= 0, \text{ on } (0, T) \times \partial\Omega \\ \lambda &= 0, \text{ in } \Omega \text{ at } t = T. \end{aligned} \quad (25)$$

The PDE (22) is semi-discretized in space with central finite differences, with the boundary conditions explicitly taken into account by using layers of ghost cells in every direction. If the direction x^i of Ω 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 correspond 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)

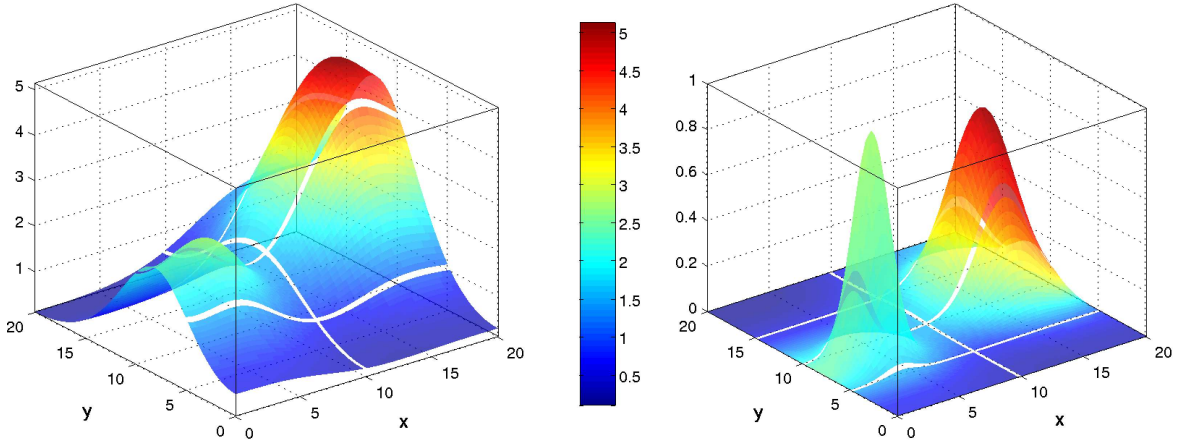


Figure 5: Results for the `cvsadjkryx_p` example problem in 2D. The gradient with respect to the source parameters is pictured on the left. On the right, the gradient was color coded and superimposed over the nominal value of the source parameters.

neighboring processes. This is implemented in the function `f_comm`, called in `f` and `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`, respectively.

For the sake of clarity, the `cvsadjkryx_p` 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 `cvsadjkryx_p` 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 `cvsadjkryx_p` on a 80×80 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 `cvsadjkryx_p` for a 2D calculation is shown below.

```

cvsadjkryx_p sample output

Parallel Krylov adjoint sensitivity analysis example
3D Advection diffusion PDE with homogeneous Neumann B.C.
Computes gradient of  $G = \int_{\Omega} c_i^2 dt d\Omega$ 
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
  0.000000 < z < 20.000000    mz = 20    npe_z = 1

Begin forward integration... done.    G = 8.232843e+03

Final Statistics..

```

[†]The name of executable for the 3-D version is `cvsadjkryx_p3D`.

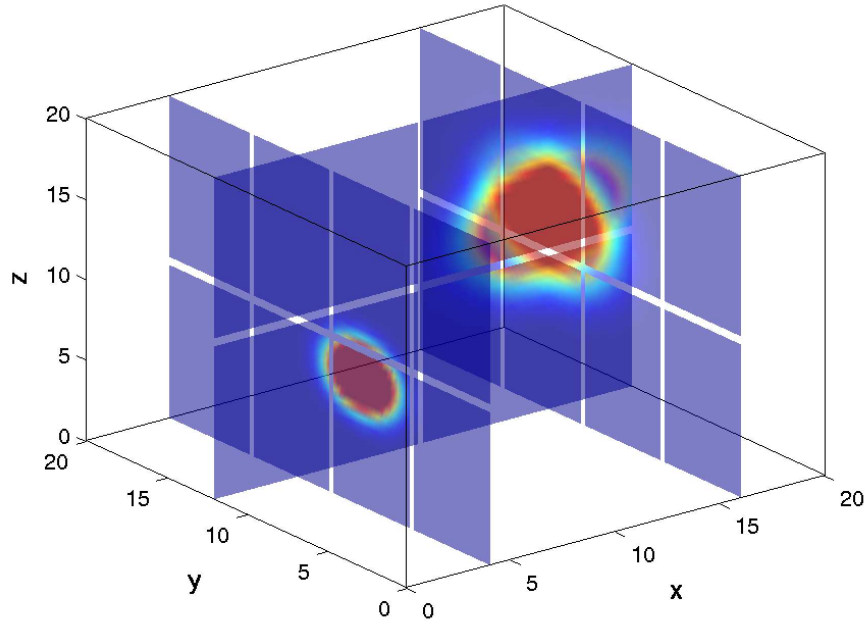


Figure 6: Results for the `cvsadjkryx.p` example problem in 3D. Nominal values of the source parameters.

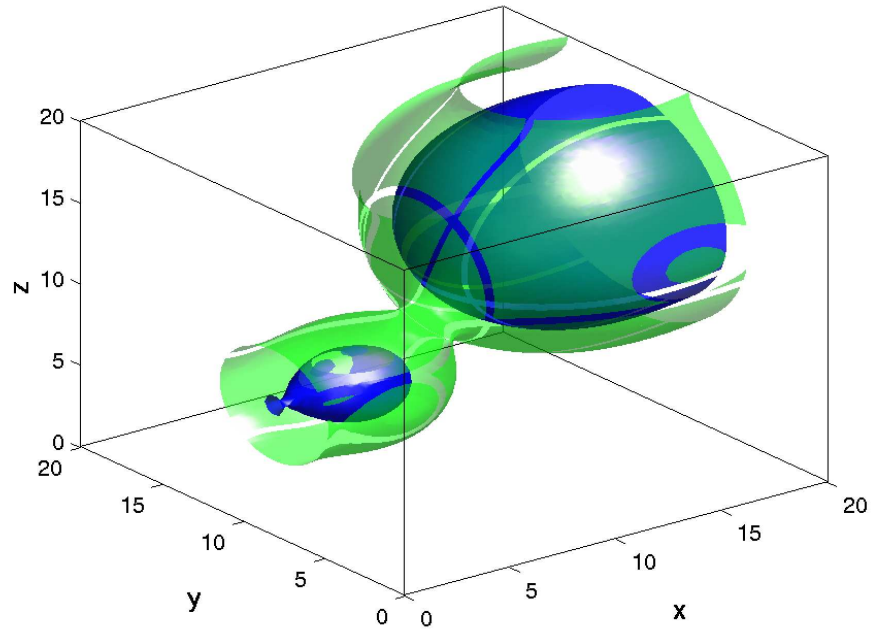


Figure 7: Results for the `cvsadjkryx.p` example problem in 3D. Two isosurfaces of the gradient with respect to the source parameters. They correspond to values of 0.25 (green) and 0.4 (blue).

lenrw	=	180946	leniw	=	212
llrw	=	180856	lliw	=	80
nst	=	118			
nfe	=	125	nfel	=	141
nni	=	121	nli	=	141
nsetups	=	17	netf	=	0
npe	=	2	nps	=	234
ncfn	=	0	ncfl	=	0

Begin backward integration... done.

Final Statistics..

lenrw	=	361716	leniw	=	212
llrw	=	180856	lliw	=	80
nst	=	70			
nfe	=	80	nfel	=	133
nni	=	76	nli	=	133
nsetups	=	15	netf	=	0
npe	=	2	nps	=	204
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

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

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 $0 \leq x \leq 20$, $30 \leq z \leq 50$ (all in km), with homogeneous Neumann boundary conditions, and for time t in $0 \leq t \leq 86400$ (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 $N = 2N_x N_z$. 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 $N = 2N_x N_y = 2 \cdot 1600 \cdot 400 = 1280000$ 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.

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

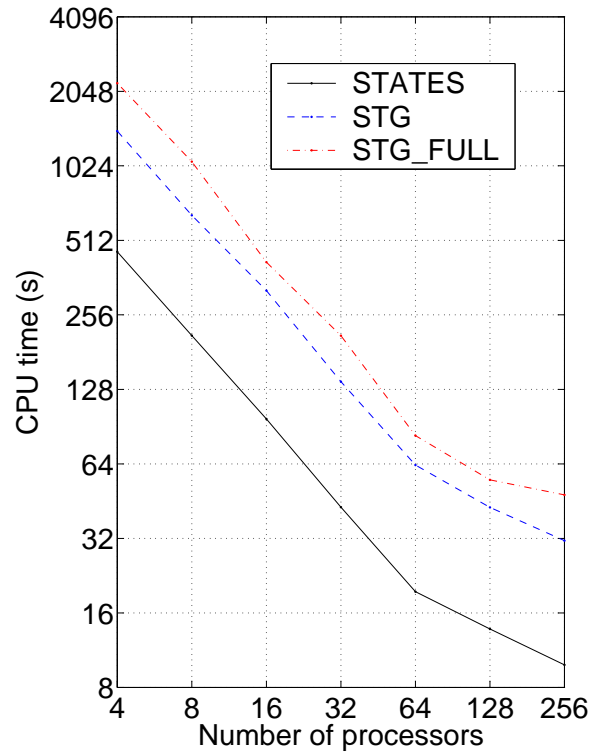


Figure 8: Speedup results for the integration of the state equations only (solid line and column 'STATES'), staggered sensitivity analysis without error control on the sensitivity variables (dashed line and column 'STG'), and staggered sensitivity analysis with full error control (dotted line and column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

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 v2.4.0. Technical report, LLNL, 2005. UCRL-SM-208110.
- [2] A. C. Hindmarsh and R. Serban. User Documentation for CVODES v2.3.0. Technical report, LLNL, 2005. UCRL-SM-208111.
- [3] R. Serban and A. C. Hindmarsh. CVODES, the sensitivity-enabled ode solver in SUNDIALS. In *Proceedings of the 5th International Conference on Multibody Systems, Nonlinear Dynamics and Control*, Long Beach, CA, 2005. ASME.

A Listing of cvsfdnonx.c

```

1  /*
2  * -----
3  * $Revision: 1.2 $
4  * $Date: 2006/07/20 16:59:31 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh, George D. Byrne,
7  *               and Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * The following is a simple example problem, with the program for
12 * its solution by CVODES. The problem is the semi-discrete form of
13 * the advection-diffusion equation in 1-D:
14 *   du/dt = q1 * d^2 u / dx^2 + q2 * du/dx
15 * on the interval 0 <= x <= 2, and the time interval 0 <= t <= 5.
16 * Homogeneous Dirichlet boundary conditions are posed, and the
17 * initial condition is:
18 *   u(x,y,t=0) = x(2-x)exp(2x).
19 * The PDE is discretized on a uniform grid of size MX+2 with
20 * central differencing, and with boundary values eliminated,
21 * leaving an ODE system of size NEQ = MX.
22 * This program solves the problem with the option for nonstiff
23 * systems: ADAMS method and functional iteration.
24 * It uses scalar relative and absolute tolerances.
25 * Output is printed at t = .5, 1.0, ..., 5.
26 * Run statistics (optional outputs) are printed at the end.
27 *
28 * Optionally, CVODES can compute sensitivities with respect to the
29 * problem parameters q1 and q2.
30 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
31 * STAGGERED1) can be used and sensitivities may be included in the
32 * error test or not (error control set on FULL or PARTIAL,
33 * respectively).
34 *
35 * Execution:
36 *
37 * If no sensitivities are desired:
38 *   % cvsfx -nosensi
39 * If sensitivities are to be computed:
40 *   % cvsfx -sensi sensi_meth err_con
41 * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
42 * {t, f}.
43 * -----
44 */
45
46 #include <stdio.h>
47 #include <stdlib.h>
48 #include <string.h>
49 #include <math.h>
50
51 #include <cvodes/cvodes.h>
52 #include <nvector/nvector_serial.h>
53 #include <sundials/sundials_types.h>
54 #include <sundials/sundials_math.h>
55
56 /* Problem Constants */
57 #define XMAX   RCONST(2.0)    /* domain boundary */

```

```

58 #define MX      10          /* mesh dimension          */
59 #define NEQ     MX          /* number of equations    */
60 #define ATOL    RCONST(1.e-5) /* scalar absolute tolerance */
61 #define T0      RCONST(0.0) /* initial time           */
62 #define T1      RCONST(0.5) /* first output time       */
63 #define DTOUT   RCONST(0.5) /* output time increment   */
64 #define NOUT    10          /* number of output times  */
65
66 #define NP      2
67 #define NS      2
68
69 #define ZERO    RCONST(0.0)
70
71 /* Type : UserData
72    contains problem parameters, grid constants, work array. */
73
74 typedef struct {
75     realtype *p;
76     realtype dx;
77 } *UserData;
78
79 /* Functions Called by the CVODES Solver */
80
81 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
82
83 /* Private Helper Functions */
84
85 static void ProcessArgs(int argc, char *argv[],
86                        booleantype *sensi, int *sensi_meth,
87                        booleantype *err_con);
88 static void WrongArgs(char *name);
89 static void SetIC(N_Vector u, realtype dx);
90 static void PrintOutput(void *cnode_mem, realtype t, N_Vector u);
91 static void PrintOutputS(N_Vector *uS);
92 static void PrintFinalStats(void *cnode_mem, booleantype sensi);
93
94 static int check_flag(void *flagvalue, char *funcname, int opt);
95
96 /*
97  *-----
98  * MAIN PROGRAM
99  *-----
100 */
101
102 int main(int argc, char *argv[])
103 {
104     void *cnode_mem;
105     UserData data;
106     realtype dx, reltol, abstol, t, tout;
107     N_Vector u;
108     int iout, flag;
109
110     realtype *pbar;
111     int is, *plist;
112     N_Vector *uS;
113     booleantype sensi, err_con;
114     int sensi_meth;
115
116     cnode_mem = NULL;

```

```

117     data = NULL;
118     u = NULL;
119     pbar = NULL;
120     plist = NULL;
121     uS = NULL;
122
123     /* Process arguments */
124     ProcessArgs(argc, argv, &sensi, &sensi_meth, &err_con);
125
126     /* Set user data */
127     data = (UserData) malloc(sizeof *data); /* Allocate data memory */
128     if(check_flag((void *)data, "malloc", 2)) return(1);
129     data->p = (realtype *) malloc(NP * sizeof(realtype));
130     dx = data->dx = XMAX/((realtype)(MX+1));
131     data->p[0] = RCONST(1.0);
132     data->p[1] = RCONST(0.5);
133
134     /* Allocate and set initial states */
135     u = N_VNew_Serial(NEQ);
136     if(check_flag((void *)u, "N_VNew_Serial", 0)) return(1);
137     SetIC(u, dx);
138
139     /* Set integration tolerances */
140     reltol = ZERO;
141     abstol = ATOL;
142
143     /* Create CVODES object */
144     ccode_mem = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
145     if(check_flag((void *)ccode_mem, "CVodeCreate", 0)) return(1);
146
147     flag = CVodeSetFdata(ccode_mem, data);
148     if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);
149
150     /* Allocate CVODES memory */
151     flag = CVodeMalloc(ccode_mem, f, T0, u, CV_SS, reltol, &abstol);
152     if(check_flag(&flag, "CVodeMalloc", 1)) return(1);
153
154     printf("\n1-D advection-diffusion equation, mesh size = %3d\n", MX);
155
156     /* Sensitivity-related settings */
157     if(sensi) {
158
159         plist = (int *) malloc(NS * sizeof(int));
160         if(check_flag((void *)plist, "malloc", 2)) return(1);
161         for(is=0; is<NS; is++) plist[is] = is;
162
163         pbar = (realtype *) malloc(NS * sizeof(realtype));
164         if(check_flag((void *)pbar, "malloc", 2)) return(1);
165         for(is=0; is<NS; is++) pbar[is] = data->p[plist[is]];
166
167         uS = N_VCloneVectorArray_Serial(NS, u);
168         if(check_flag((void *)uS, "N_VCloneVectorArray_Serial", 0)) return(1);
169         for(is=0; is<NS; is++)
170             N_VConst(ZERO, uS[is]);
171
172         flag = CVodeSensMalloc(ccode_mem, NS, sensi_meth, uS);
173         if(check_flag(&flag, "CVodeSensMalloc", 1)) return(1);
174
175         flag = CVodeSetSensErrCon(ccode_mem, err_con);

```



```

235  *-----
236  * FUNCTIONS CALLED BY CVOIDS
237  *-----
238  */
239
240 /*
241  * f routine. Compute f(t,u).
242  */
243
244 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
245 {
246     realtype ui, ult, urt, hordc, horac, hdiff, hadv;
247     realtype dx;
248     realtype *udata, *dudata;
249     int i;
250     UserData data;
251
252     udata = NV_DATA_S(u);
253     dudata = NV_DATA_S(udot);
254
255     /* Extract needed problem constants from data */
256     data = (UserData) f_data;
257     dx = data->dx;
258     hordc = data->p[0]/(dx*dx);
259     horac = data->p[1]/(RCONST(2.0)*dx);
260
261     /* Loop over all grid points. */
262     for (i=0; i<NEQ; i++) {
263
264         /* Extract u at x_i and two neighboring points */
265         ui = udata[i];
266         if(i!=0)
267             ult = udata[i-1];
268         else
269             ult = ZERO;
270         if(i!=NEQ-1)
271             urt = udata[i+1];
272         else
273             urt = ZERO;
274
275         /* Set diffusion and advection terms and load into udot */
276         hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
277         hadv = horac*(urt - ult);
278         dudata[i] = hdiff + hadv;
279     }
280
281     return(0);
282 }
283
284 /*
285  *-----
286  * PRIVATE FUNCTIONS
287  *-----
288  */
289
290 /*
291  * Process and verify arguments to cvsfdnonx.
292  */
293

```

```

294 static void ProcessArgs(int argc, char *argv[],
295                          booleantype *sensi, int *sensi_meth, booleantype *err_con)
296 {
297     *sensi = FALSE;
298     *sensi_meth = -1;
299     *err_con = FALSE;
300
301     if (argc < 2) WrongArgs(argv[0]);
302
303     if (strcmp(argv[1], "-nosensi") == 0)
304         *sensi = FALSE;
305     else if (strcmp(argv[1], "-sensi") == 0)
306         *sensi = TRUE;
307     else
308         WrongArgs(argv[0]);
309
310     if (*sensi) {
311
312         if (argc != 4)
313             WrongArgs(argv[0]);
314
315         if (strcmp(argv[2], "sim") == 0)
316             *sensi_meth = CV_SIMULTANEOUS;
317         else if (strcmp(argv[2], "stg") == 0)
318             *sensi_meth = CV_STAGGERED;
319         else if (strcmp(argv[2], "stg1") == 0)
320             *sensi_meth = CV_STAGGERED1;
321         else
322             WrongArgs(argv[0]);
323
324         if (strcmp(argv[3], "t") == 0)
325             *err_con = TRUE;
326         else if (strcmp(argv[3], "f") == 0)
327             *err_con = FALSE;
328         else
329             WrongArgs(argv[0]);
330     }
331 }
332
333 static void WrongArgs(char *name)
334 {
335     printf("\nUsage: %s [-nosensi] [-sensi_sensi_meth_err_con]\n", name);
336     printf("          sensi_meth = sim, stg, or stg1\n");
337     printf("          err_con = t or f\n");
338
339     exit(0);
340 }
341
342 /*
343  * Set initial conditions in u vector.
344  */
345
346 static void SetIC(N_Vector u, realtype dx)
347 {
348     int i;
349     realtype x;
350     realtype *udata;
351
352

```

```

353     /* Set pointer to data array and get local length of u. */
354     udata = NV_DATA_S(u);
355
356     /* Load initial profile into u vector */
357     for (i=0; i<NEQ; i++) {
358         x = (i+1)*dx;
359         udata[i] = x*(XMAX - x)*EXP(RCONST(2.0)*x);
360     }
361 }
362
363 /*
364  * Print current t, step count, order, stepsize, and max norm of solution
365  */
366
367 static void PrintOutput(void *cnode_mem, realtype t, N_Vector u)
368 {
369     long int nst;
370     int qu, flag;
371     realtype hu;
372
373     flag = CVodeGetNumSteps(cnode_mem, &nst);
374     check_flag(&flag, "CVodeGetNumSteps", 1);
375     flag = CVodeGetLastOrder(cnode_mem, &qu);
376     check_flag(&flag, "CVodeGetLastOrder", 1);
377     flag = CVodeGetLastStep(cnode_mem, &hu);
378     check_flag(&flag, "CVodeGetLastStep", 1);
379
380     #if defined(SUNDIALS_EXTENDED_PRECISION)
381         printf("%8.3Le_2d_8.3Le_5ld\n", t, qu, hu, nst);
382     #elif defined(SUNDIALS_DOUBLE_PRECISION)
383         printf("%8.3le_2d_8.3le_5ld\n", t, qu, hu, nst);
384     #else
385         printf("%8.3e_2d_8.3e_5ld\n", t, qu, hu, nst);
386     #endif
387
388     printf("Solution");
389
390     #if defined(SUNDIALS_EXTENDED_PRECISION)
391         printf("%12.4Le\n", N_VMaxNorm(u));
392     #elif defined(SUNDIALS_DOUBLE_PRECISION)
393         printf("%12.4le\n", N_VMaxNorm(u));
394     #else
395         printf("%12.4e\n", N_VMaxNorm(u));
396     #endif
397 }
398
399 /*
400  * Print max norm of sensitivities
401  */
402
403 static void PrintOutputS(N_Vector *uS)
404 {
405     printf("Sensitivity");
406     #if defined(SUNDIALS_EXTENDED_PRECISION)
407         printf("%12.4Le\n", N_VMaxNorm(uS[0]));
408     #elif defined(SUNDIALS_DOUBLE_PRECISION)
409         printf("%12.4le\n", N_VMaxNorm(uS[0]));
410     #else
411         printf("%12.4e\n", N_VMaxNorm(uS[0]));

```



```

412 #endif
413
414 printf("Sensitivity_2");
415 #if defined(SUNDIALS_EXTENDED_PRECISION)
416 printf("%12.4Le\n", N_VMaxNorm(uS[1]));
417 #elif defined(SUNDIALS_DOUBLE_PRECISION)
418 printf("%12.4le\n", N_VMaxNorm(uS[1]));
419 #else
420 printf("%12.4e\n", N_VMaxNorm(uS[1]));
421 #endif
422 }
423
424
425 /*
426  * Print some final statistics located in the CVODES memory
427  */
428
429 static void PrintFinalStats(void *cvode_mem, booleantype sensi)
430 {
431     long int nst;
432     long int nfe, nsetups, nni, ncfn, netf;
433     long int nfSe, nfeS, nsetupsS, nniS, ncfnS, netfS;
434     int flag;
435
436     flag = CVodeGetNumSteps(cvode_mem, &nst);
437     check_flag(&flag, "CVodeGetNumSteps", 1);
438     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
439     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
440     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
441     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
442     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
443     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
444     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
445     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
446     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
447     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
448
449     if (sensi) {
450         flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
451         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
452         flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
453         check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
454         flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
455         check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
456         flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
457         check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);
458         flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
459         check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1);
460         flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
461         check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1);
462     }
463
464     printf("\nFinal_Statistics\n\n");
465     printf("nst===== %5ld\n\n", nst);
466     printf("nfe===== %5ld\n", nfe);
467     printf("netf===== %5ld nsetups= %5ld\n", netf, nsetups);
468     printf("nni===== %5ld ncfn===== %5ld\n", nni, ncfn);
469
470     if (sensi) {

```

```

471     printf("\n");
472     printf("nfSe=%5ldnfeS=%5ld\n", nfSe, nfeS);
473     printf("netfs=%5ldnsetupsS=%5ld\n", netfS, nsetupsS);
474     printf("nniS=%5ldncfnS=%5ld\n", nniS, ncfnS);
475 }
476
477 }
478
479 /*
480  * Check function return value...
481  *   opt == 0 means SUNDIALS function allocates memory so check if
482  *       returned NULL pointer
483  *   opt == 1 means SUNDIALS function returns a flag so check if
484  *       flag >= 0
485  *   opt == 2 means function allocates memory so check if returned
486  *       NULL pointer
487  */
488
489 static int check_flag(void *flagvalue, char *funcname, int opt)
490 {
491     int *errflag;
492
493     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
494     if (opt == 0 && flagvalue == NULL) {
495         fprintf(stderr,
496             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
497             funcname);
498         return(1); }
499
500     /* Check if flag < 0 */
501     else if (opt == 1) {
502         errflag = (int *) flagvalue;
503         if (*errflag < 0) {
504             fprintf(stderr,
505                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
506                 funcname, *errflag);
507             return(1); }}
508
509     /* Check if function returned NULL pointer - no memory allocated */
510     else if (opt == 2 && flagvalue == NULL) {
511         fprintf(stderr,
512             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
513             funcname);
514         return(1); }
515
516     return(0);
517 }

```

B Listing of cvsfwddenx.c

```

1  /*
2  * -----
3  * $Revision: 1.1 $
4  * $Date: 2006/07/05 15:50:07 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh, and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * The following is a simple example problem, with the coding
12 * needed for its solution by CVODES. The problem is from chemical
13 * kinetics, and consists of the following three rate equations:
14 *   dy1/dt = -p1*y1 + p2*y2*y3
15 *   dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
16 *   dy3/dt =  p3*(y2)^2
17 * on the interval from t = 0.0 to t = 4.e10, with initial
18 * conditions y1 = 1.0, y2 = y3 = 0. The reaction rates are: p1=0.04,
19 * p2=1e4, and p3=3e7. The problem is stiff.
20 * This program solves the problem with the BDF method, Newton
21 * iteration with the CVODES dense linear solver, and a
22 * user-supplied Jacobian routine.
23 * It uses a scalar relative tolerance and a vector absolute
24 * tolerance.
25 * Output is printed in decades from t = .4 to t = 4.e10.
26 * Run statistics (optional outputs) are printed at the end.
27 *
28 * Optionally, CVODES can compute sensitivities with respect to the
29 * problem parameters p1, p2, and p3.
30 * The sensitivity right hand side is given analytically through the
31 * user routine fS (of type SensRhs1Fn).
32 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
33 * STAGGERED1) can be used and sensitivities may be included in the
34 * error test or not (error control set on TRUE or FALSE,
35 * respectively).
36 *
37 * Execution:
38 *
39 * If no sensitivities are desired:
40 *   % cvsdX -nosensi
41 * If sensitivities are to be computed:
42 *   % cvsdX -sensi sensi_meth err_con
43 * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
44 * {t, f}.
45 * -----
46 */
47
48 #include <stdio.h>
49 #include <stdlib.h>
50 #include <string.h>
51
52 #include <cvodes/cvodes.h>          /* prototypes for CVODES fcts. and consts. */
53 #include <cvodes/cvodes_dense.h>    /* prototype for CVDENSE fcts. and constants */
54 #include <nvector/nvector_serial.h>  /* defs. of serial NVECTOR fcts. and macros */
55
56 #include <sundials/sundials_types.h> /* def. of type realtype */
57 #include <sundials/sundials_math.h>  /* definition of ABS */

```

```

57
58 /* Accessor macros */
59
60 #define Ith(v,i)      NV_Ith_S(v,i-1)      /* i-th vector component i=1..NEQ */
61 #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j=1..NEQ */
62
63 /* Problem Constants */
64
65 #define NEQ      3      /* number of equations */
66 #define Y1      RCONST(1.0) /* initial y components */
67 #define Y2      RCONST(0.0)
68 #define Y3      RCONST(0.0)
69 #define RTOL    RCONST(1e-4) /* scalar relative tolerance */
70 #define ATOL1   RCONST(1e-8) /* vector absolute tolerance components */
71 #define ATOL2   RCONST(1e-14)
72 #define ATOL3   RCONST(1e-6)
73 #define T0      RCONST(0.0) /* initial time */
74 #define T1      RCONST(0.4) /* first output time */
75 #define TMULT   RCONST(10.0) /* output time factor */
76 #define NOUT    12      /* number of output times */
77
78 #define NP      3      /* number of problem parameters */
79 #define NS      3      /* number of sensitivities computed */
80
81 #define ZERO    RCONST(0.0)
82
83 /* Type : UserData */
84
85 typedef struct {
86     realtype p[3];          /* problem parameters */
87 } *UserData;
88
89 /* Prototypes of functions by CVOIDS */
90
91 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
92
93 static int Jac(long int N, DenseMat J, realtype t,
94               N_Vector y, N_Vector fy, void *jac_data,
95               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
96
97 static int fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
98               int iS, N_Vector yS, N_Vector ySdot,
99               void *fS_data, N_Vector tmp1, N_Vector tmp2);
100
101 static int ewt(N_Vector y, N_Vector w, void *e_data);
102
103 /* Prototypes of private functions */
104
105 static void ProcessArgs(int argc, char *argv[],
106                        booleantype *sensi, int *sensi_meth,
107                        booleantype *err_con);
108 static void WrongArgs(char *name);
109 static void PrintOutput(void *cvoid_mem, realtype t, N_Vector u);
110 static void PrintOutputS(N_Vector *uS);
111 static void PrintFinalStats(void *cvoid_mem, booleantype sensi);
112 static int check_flag(void *flagvalue, char *funcname, int opt);
113
114 /*
115  *-----

```

```

116  * MAIN PROGRAM
117  *-----
118  */
119
120  int main(int argc, char *argv[])
121  {
122      void *cnode_mem;
123      UserData data;
124      realtype t, tout;
125      N_Vector y;
126      int iout, flag;
127
128      realtype pbar[NS];
129      int is;
130      N_Vector *yS;
131      booleantype sensi, err_con;
132      int sensi_meth;
133
134      cnode_mem = NULL;
135      data      = NULL;
136      y         = NULL;
137      yS        = NULL;
138
139      /* Process arguments */
140      ProcessArgs(argc, argv, &sensi, &sensi_meth, &err_con);
141
142      /* User data structure */
143      data = (UserData) malloc(sizeof *data);
144      if (check_flag((void *)data, "malloc", 2)) return(1);
145      data->p[0] = RCONST(0.04);
146      data->p[1] = RCONST(1.0e4);
147      data->p[2] = RCONST(3.0e7);
148
149      /* Initial conditions */
150      y = N_VNew_Serial(NEQ);
151      if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
152
153      Ith(y,1) = Y1;
154      Ith(y,2) = Y2;
155      Ith(y,3) = Y3;
156
157      /* Create CNODES object */
158      cnode_mem = CNodeCreate(CV_BDF, CV_NEWTON);
159      if (check_flag((void *)cnode_mem, "CNodeCreate", 0)) return(1);
160
161      /* Allocate space for CNODES */
162      flag = CNodeMalloc(cnode_mem, f, TO, y, CV_WF, 0.0, NULL);
163      if (check_flag(&flag, "CNodeMalloc", 1)) return(1);
164
165      /* Use private function to compute error weights */
166      flag = CNodeSetEwtFn(cnode_mem, ewt, NULL);
167      if (check_flag(&flag, "CNodeSetEwtFn", 1)) return(1);
168
169      /* Attach user data */
170      flag = CNodeSetFdata(cnode_mem, data);
171      if (check_flag(&flag, "CNodeSetFdata", 1)) return(1);
172
173      /* Attach linear solver */
174      flag = CVDense(cnode_mem, NEQ);

```

```

175     if (check_flag(&flag, "CVDense", 1)) return(1);
176
177     flag = CVDenseSetJacFn(cvode_mem, Jac, data);
178     if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
179
180     printf("\n3-species_chemical_kinetics_problem\n");
181
182     /* Sensitivity-related settings */
183     if (sensi) {
184
185         pbar[0] = data->p[0];
186         pbar[1] = data->p[1];
187         pbar[2] = data->p[2];
188
189         yS = N_VCloneVectorArray_Serial(NS, y);
190         if (check_flag((void *)yS, "N_VCloneVectorArray_Serial", 0)) return(1);
191         for (is=0; is<NS; is++) N_VConst(ZERO, yS[is]);
192
193         flag = CVodeSensMalloc(cvode_mem, NS, sensi_meth, yS);
194         if (check_flag(&flag, "CVodeSensMalloc", 1)) return(1);
195
196         flag = CVodeSetSensRhs1Fn(cvode_mem, fS, data);
197         if (check_flag(&flag, "CVodeSetSensRhs1Fn", 1)) return(1);
198         flag = CVodeSetSensErrCon(cvode_mem, err_con);
199         if (check_flag(&flag, "CVodeSetSensErrCon", 1)) return(1);
200         flag = CVodeSetSensParams(cvode_mem, NULL, pbar, NULL);
201         if (check_flag(&flag, "CVodeSetSensParams", 1)) return(1);
202
203         printf("Sensitivity:_YES_");
204         if (sensi_meth == CV_SIMULTANEOUS)
205             printf("(_SIMULTANEOUS_+");
206         else
207             if (sensi_meth == CV_STAGGERED) printf("(_STAGGERED_+");
208             else printf("(_STAGGERED1_+");
209         if (err_con) printf("(_FULL_ERROR_CONTROL_)");
210         else printf("(_PARTIAL_ERROR_CONTROL_)");
211
212     } else {
213
214         printf("Sensitivity:_NO_");
215
216     }
217
218     /* In loop over output points, call CVode, print results, test for error */
219
220     printf("\n\n");
221     printf("=====");
222     printf("=====\n");
223     printf("TTTTTQHHHNNSTTTTTTTTTy1");
224     printf("TTTTTTTTTyy2TTTTTTTTTTTyy3TTT\n");
225     printf("=====");
226     printf("=====\n");
227
228     for (iout=1, tout=T1; iout <= NOUT; iout++, tout += TMULT) {
229
230         flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
231         if (check_flag(&flag, "CVode", 1)) break;
232
233         PrintOutput(cvode_mem, t, y);

```

```

234
235     if (sensi) {
236         flag = CVodeGetSens(cvode_mem, t, yS);
237         if (check_flag(&flag, "CVodeGetSens", 1)) break;
238         PrintOutputS(yS);
239     }
240     printf("-----");
241     printf("-----\n");
242
243 }
244
245 /* Print final statistics */
246 PrintFinalStats(cvode_mem, sensi);
247
248 /* Free memory */
249
250 N_VDestroy_Serial(y);          /* Free y vector */
251 if (sensi) {
252     N_VDestroyVectorArray_Serial(yS, NS); /* Free yS vector */
253 }
254 free(data);                    /* Free user data */
255 CVodeFree(&cvode_mem);         /* Free CVODES memory */
256
257 return(0);
258 }
259
260 /*
261 *-----
262 * FUNCTIONS CALLED BY CVODES
263 *-----
264 */
265
266 /*
267 * f routine. Compute f(t,y).
268 */
269
270 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
271 {
272     realtype y1, y2, y3, yd1, yd3;
273     UserData data;
274     realtype p1, p2, p3;
275
276     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
277     data = (UserData) f_data;
278     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
279
280     yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
281     yd3 = Ith(ydot,3) = p3*y2*y2;
282     Ith(ydot,2) = -yd1 - yd3;
283
284     return(0);
285 }
286
287
288 /*
289 * Jacobian routine. Compute J(t,y).
290 */
291
292 static int Jac(long int N, DenseMat J, realtype t,

```

```

293         N_Vector y, N_Vector fy, void *jac_data,
294         N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
295     {
296         realtype y1, y2, y3;
297         UserData data;
298         realtype p1, p2, p3;
299
300         y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
301         data = (UserData) jac_data;
302         p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
303
304         IJth(J,1,1) = -p1;  IJth(J,1,2) = p2*y3;          IJth(J,1,3) = p2*y2;
305         IJth(J,2,1) =  p1;  IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
306         IJth(J,3,2) = 2*p3*y2;
307
308         return(0);
309     }
310
311     /*
312     * fS routine. Compute sensitivity r.h.s.
313     */
314
315     static int fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
316                 int iS, N_Vector yS, N_Vector ySdot,
317                 void *fS_data, N_Vector tmp1, N_Vector tmp2)
318     {
319         UserData data;
320         realtype p1, p2, p3;
321         realtype y1, y2, y3;
322         realtype s1, s2, s3;
323         realtype sd1, sd2, sd3;
324
325         data = (UserData) fS_data;
326         p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
327
328         y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
329         s1 = Ith(yS,1); s2 = Ith(yS,2); s3 = Ith(yS,3);
330
331         sd1 = -p1*s1 + p2*y3*s2 + p2*y2*s3;
332         sd3 = 2*p3*y2*s2;
333         sd2 = -sd1-sd3;
334
335         switch (iS) {
336         case 0:
337             sd1 += -y1;
338             sd2 +=  y1;
339             break;
340         case 1:
341             sd1 +=  y2*y3;
342             sd2 += -y2*y3;
343             break;
344         case 2:
345             sd2 += -y2*y2;
346             sd3 +=  y2*y2;
347             break;
348         }
349
350         Ith(ySdot,1) = sd1;
351         Ith(ySdot,2) = sd2;

```



```

352     lth(ySdot,3) = sd3;
353
354     return(0);
355 }
356
357 /*
358  * EwtSet function. Computes the error weights at the current solution.
359  */
360
361 static int ewt(N_Vector y, N_Vector w, void *e_data)
362 {
363     int i;
364     realtype yy, ww, rtol, atol[3];
365
366     rtol = RTOL;
367     atol[0] = ATOL1;
368     atol[1] = ATOL2;
369     atol[2] = ATOL3;
370
371     for (i=1; i<=3; i++) {
372         yy = lth(y,i);
373         ww = rtol * ABS(yy) + atol[i-1];
374         if (ww <= 0.0) return (-1);
375         lth(w,i) = 1.0/ww;
376     }
377
378     return(0);
379 }
380
381 /*
382  *-----
383  * PRIVATE FUNCTIONS
384  *-----
385  */
386
387 /*
388  * Process and verify arguments to cvsfwddenx.
389  */
390
391 static void ProcessArgs(int argc, char *argv[],
392                         booleantype *sensi, int *sensi_meth, booleantype *err_con)
393 {
394     *sensi = FALSE;
395     *sensi_meth = -1;
396     *err_con = FALSE;
397
398     if (argc < 2) WrongArgs(argv[0]);
399
400     if (strcmp(argv[1], "-nosensi") == 0)
401         *sensi = FALSE;
402     else if (strcmp(argv[1], "-sensi") == 0)
403         *sensi = TRUE;
404     else
405         WrongArgs(argv[0]);
406
407     if (*sensi) {
408
409         if (argc != 4)
410             WrongArgs(argv[0]);

```

```

411
412     if (strcmp(argv[2], "sim") == 0)
413         *sensi_meth = CV_SIMULTANEOUS;
414     else if (strcmp(argv[2], "stg") == 0)
415         *sensi_meth = CV_STAGGERED;
416     else if (strcmp(argv[2], "stg1") == 0)
417         *sensi_meth = CV_STAGGERED1;
418     else
419         WrongArgs(argv[0]);
420
421     if (strcmp(argv[3], "t") == 0)
422         *err_con = TRUE;
423     else if (strcmp(argv[3], "f") == 0)
424         *err_con = FALSE;
425     else
426         WrongArgs(argv[0]);
427 }
428
429 }
430
431 static void WrongArgs(char *name)
432 {
433     printf("\nUsage: %s [-nosensi] [-sensi_sensi_meth_err_con]\n", name);
434     printf("          sensi_meth=sim, stg, or stg1\n");
435     printf("          err_con=t or f\n");
436
437     exit(0);
438 }
439
440 /*
441  * Print current t, step count, order, stepsize, and solution.
442  */
443
444 static void PrintOutput(void *cnode_mem, realtype t, N_Vector u)
445 {
446     long int nst;
447     int qu, flag;
448     realtype hu, *udata;
449
450     udata = NV_DATA_S(u);
451
452     flag = CVodeGetNumSteps(cnode_mem, &nst);
453     check_flag(&flag, "CVodeGetNumSteps", 1);
454     flag = CVodeGetLastOrder(cnode_mem, &qu);
455     check_flag(&flag, "CVodeGetLastOrder", 1);
456     flag = CVodeGetLastStep(cnode_mem, &hu);
457     check_flag(&flag, "CVodeGetLastStep", 1);
458
459     #if defined(SUNDIALS_EXTENDED_PRECISION)
460         printf("%8.3le%2d%8.3le%5ld\n", t, qu, hu, nst);
461     #elif defined(SUNDIALS_DOUBLE_PRECISION)
462         printf("%8.3le%2d%8.3le%5ld\n", t, qu, hu, nst);
463     #else
464         printf("%8.3e%2d%8.3e%5ld\n", t, qu, hu, nst);
465     #endif
466
467     printf("          Solution");
468
469     #if defined(SUNDIALS_EXTENDED_PRECISION)

```

```

470     printf("%12.4Le_%12.4Le_%12.4Le\n", udata[0], udata[1], udata[2]);
471 #elif defined(SUNDIALS_DOUBLE_PRECISION)
472     printf("%12.4le_%12.4le_%12.4le\n", udata[0], udata[1], udata[2]);
473 #else
474     printf("%12.4e_%12.4e_%12.4e\n", udata[0], udata[1], udata[2]);
475 #endif
476
477 }
478
479 /*
480  * Print sensitivities.
481  */
482
483 static void PrintOutputS(N_Vector *uS)
484 {
485     realtype *sdata;
486
487     sdata = NV_DATA_S(uS[0]);
488     printf("Sensitivity_1");
489
490 #if defined(SUNDIALS_EXTENDED_PRECISION)
491     printf("%12.4Le_%12.4Le_%12.4Le\n", sdata[0], sdata[1], sdata[2]);
492 #elif defined(SUNDIALS_DOUBLE_PRECISION)
493     printf("%12.4le_%12.4le_%12.4le\n", sdata[0], sdata[1], sdata[2]);
494 #else
495     printf("%12.4e_%12.4e_%12.4e\n", sdata[0], sdata[1], sdata[2]);
496 #endif
497
498     sdata = NV_DATA_S(uS[1]);
499     printf("Sensitivity_2");
500
501 #if defined(SUNDIALS_EXTENDED_PRECISION)
502     printf("%12.4Le_%12.4Le_%12.4Le\n", sdata[0], sdata[1], sdata[2]);
503 #elif defined(SUNDIALS_DOUBLE_PRECISION)
504     printf("%12.4le_%12.4le_%12.4le\n", sdata[0], sdata[1], sdata[2]);
505 #else
506     printf("%12.4e_%12.4e_%12.4e\n", sdata[0], sdata[1], sdata[2]);
507 #endif
508
509     sdata = NV_DATA_S(uS[2]);
510     printf("Sensitivity_3");
511
512 #if defined(SUNDIALS_EXTENDED_PRECISION)
513     printf("%12.4Le_%12.4Le_%12.4Le\n", sdata[0], sdata[1], sdata[2]);
514 #elif defined(SUNDIALS_DOUBLE_PRECISION)
515     printf("%12.4le_%12.4le_%12.4le\n", sdata[0], sdata[1], sdata[2]);
516 #else
517     printf("%12.4e_%12.4e_%12.4e\n", sdata[0], sdata[1], sdata[2]);
518 #endif
519 }
520
521 /*
522  * Print some final statistics from the CVODES memory.
523  */
524
525 static void PrintFinalStats(void *cnode_mem, booleantype sensi)
526 {
527     long int nst;
528     long int nfe, nsetups, nni, ncnf, netf;

```

```

529     long int nfSe, nfeS, nsetupsS, nniS, ncfns, netfS;
530     long int nje, nfeLS;
531     int flag;
532
533     flag = CVodeGetNumSteps(cvode_mem, &nst);
534     check_flag(&flag, "CVodeGetNumSteps", 1);
535     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
536     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
537     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
538     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
539     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
540     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
541     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
542     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
543     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
544     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
545
546     if (sensi) {
547         flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
548         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
549         flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
550         check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
551         flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
552         check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
553         flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
554         check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);
555         flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
556         check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1);
557         flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
558         check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1);
559     }
560
561     flag = CVDenseGetNumJacEvals(cvode_mem, &nje);
562     check_flag(&flag, "CVDenseGetNumJacEvals", 1);
563     flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeLS);
564     check_flag(&flag, "CVDenseGetNumRhsEvals", 1);
565
566     printf("\nFinal Statistics\n\n");
567     printf("nst===== %5ld\n\n", nst);
568     printf("nfe===== %5ld\n", nfe);
569     printf("netf===== %5ld nsetups= %5ld\n", netf, nsetups);
570     printf("nni===== %5ld ncfns= %5ld\n", nni, ncfns);
571
572     if (sensi) {
573         printf("\n");
574         printf("nfSe===== %5ld nfeS===== %5ld\n", nfSe, nfeS);
575         printf("netfS===== %5ld nsetupsS= %5ld\n", netfS, nsetupsS);
576         printf("nniS===== %5ld ncfnsS===== %5ld\n", nniS, ncfnsS);
577     }
578
579     printf("\n");
580     printf("nje===== %5ld nfeLS===== %5ld\n", nje, nfeLS);
581
582 }
583
584 /*
585  * Check function return value.
586  *   opt == 0 means SUNDIALS function allocates memory so check if
587  *   returned NULL pointer

```

```

588 *     opt == 1 means SUNDIALS function returns a flag so check if
589 *         flag >= 0
590 *     opt == 2 means function allocates memory so check if returned
591 *         NULL pointer
592 */
593
594 static int check_flag(void *flagvalue, char *funcname, int opt)
595 {
596     int *errflag;
597
598     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
599     if (opt == 0 && flagvalue == NULL) {
600         fprintf(stderr,
601             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
602             funcname);
603         return(1); }
604
605     /* Check if flag < 0 */
606     else if (opt == 1) {
607         errflag = (int *) flagvalue;
608         if (*errflag < 0) {
609             fprintf(stderr,
610                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
611                 funcname, *errflag);
612             return(1); }}
613
614     /* Check if function returned NULL pointer - no memory allocated */
615     else if (opt == 2 && flagvalue == NULL) {
616         fprintf(stderr,
617             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
618             funcname);
619         return(1); }
620
621     return(0);
622 }

```

C Listing of cvsfwdkryx_p.c

```

1  /*
2  * -----
3  * $Revision: 1.3 $
4  * $Date: 2006/10/11 16:33:58 $
5  * -----
6  * Programmer(s): S. D. Cohen, A. C. Hindmarsh, Radu Serban,
7  *                and M. R. Wittman @ LLNL
8  * -----
9  * Example problem:
10 *
11 * An ODE system is generated from the following 2-species diurnal
12 * kinetics advection-diffusion PDE system in 2 space dimensions:
13 *
14 *  $dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy$ 
15 *                 $+ Ri(c1,c2,t)$  for  $i = 1,2$ , where
16 *  $R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2$ ,
17 *  $R2(c1,c2,t) = q1*c1*c3 - q2*c1*c2 - q4(t)*c2$ ,
18 *  $Kv(y) = Kv0*exp(y/5)$ ,
19 *  $Kh, V, Kv0, q1, q2$ , and  $c3$  are constants, and  $q3(t)$  and  $q4(t)$ 
20 * vary diurnally. The problem is posed on the square
21 *  $0 \leq x \leq 20$ ,  $30 \leq y \leq 50$  (all in km),
22 * with homogeneous Neumann boundary conditions, and for time  $t$  in
23 *  $0 \leq t \leq 86400$  sec (1 day).
24 * The PDE system is treated by central differences on a uniform
25 * mesh, with simple polynomial initial profiles.
26 *
27 * The problem is solved by CVODES on NPE processors, treated
28 * as a rectangular process grid of size NPEX by NPEY, with
29 * NPE = NPEX*NPEY. Each processor contains a subgrid of size
30 * MXSUB by MYSUB of the (x,y) mesh. Thus the actual mesh sizes
31 * are MX = MXSUB*NPEX and MY = MYSUB*NPEY, and the ODE system size
32 * is neq = 2*MX*MY.
33 *
34 * The solution with CVODES is done with the BDF/GMRES method (i.e.
35 * using the CVSPGMR linear solver) and the block-diagonal part of
36 * the Newton matrix as a left preconditioner. A copy of the
37 * block-diagonal part of the Jacobian is saved and conditionally
38 * reused within the Precond routine.
39 *
40 * Performance data and sampled solution values are printed at
41 * selected output times, and all performance counters are printed
42 * on completion.
43 *
44 * Optionally, CVODES can compute sensitivities with respect to the
45 * problem parameters  $q1$  and  $q2$ .
46 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
47 * STAGGERED1) can be used and sensitivities may be included in the
48 * error test or not (error control set on FULL or PARTIAL,
49 * respectively).
50 *
51 * Execution:
52 *
53 * Note: This version uses MPI for user routines, and the CVODES
54 * solver. In what follows, N is the number of processors,
55 *  $N = NPEX*NPEY$  (see constants below) and it is assumed that
56 * the MPI script mpirun is used to run a parallel
57 * application.

```

```

58  * If no sensitivities are desired:
59  *   % mpirun -np N cvsfwdkryx_p -nosensi
60  * If sensitivities are to be computed:
61  *   % mpirun -np N cvsfwdkryx_p -sensi sensi_meth err_con
62  * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
63  * {t, f}.
64  * -----
65  */
66
67  #include <stdio.h>
68  #include <stdlib.h>
69  #include <math.h>
70  #include <string.h>
71
72  #include <cvodes/cvodes.h>           /* main CVODES header file */
73  #include <cvodes/cvodes_spgmr.h>     /* defs. for CVSPGMR fcts. and constants */
74  #include <nvector/nvector_parallel.h> /* defs of paralel NVECTOR fcts. and macros */
75  #include <sundials/sundials_smalldense.h> /* generic DENSE solver used in prec. */
76  #include <sundials/sundials_math.h>   /* contains macros SQR and EXP */
77  #include <sundials/sundials_types.h>  /* def. of realtype */
78
79  #include <mpi.h>
80
81
82  /* Problem Constants */
83
84  #define NVARs      2           /* number of species */
85  #define C1_SCALE   RCONST(1.0e6) /* coefficients in initial profiles */
86  #define C2_SCALE   RCONST(1.0e12)
87
88  #define TO         RCONST(0.0)   /* initial time */
89  #define NOUT       12           /* number of output times */
90  #define TWOHR      RCONST(7200.0) /* number of seconds in two hours */
91  #define HALFDAY    RCONST(4.32e4) /* number of seconds in a half day */
92  #define PI         RCONST(3.1415926535898) /* pi */
93
94  #define XMIN       RCONST(0.0)   /* grid boundaries in x */
95  #define XMAX       RCONST(20.0)
96  #define YMIN       RCONST(30.0)  /* grid boundaries in y */
97  #define YMAX       RCONST(50.0)
98
99  #define NPEX       2           /* no. PEs in x direction of PE array */
100 #define NPEY       2           /* no. PEs in y direction of PE array */
101                               /* Total no. PEs = NPEX*NPEY */
102 #define MXSUB      5           /* no. x points per subgrid */
103 #define MYSUB      5           /* no. y points per subgrid */
104
105 #define MX          (NPEX*MXSUB) /* MX = number of x mesh points */
106 #define MY          (NPEY*MYSUB) /* MY = number of y mesh points */
107                               /* Spatial mesh is MX by MY */
108
109  /* CVMalloc Constants */
110
111  #define RTOL        RCONST(1.0e-5) /* scalar relative tolerance */
112  #define FLOOR       RCONST(100.0)  /* value of C1 or C2 at which tols. */
113                               /* change from relative to absolute */
114  #define ATOL        (RTOL*FLOOR)   /* scalar absolute tolerance */
115
116  /* Sensitivity constants */

```

```

117 #define NP          8          /* number of problem parameters      */
118 #define NS          2          /* number of sensitivities          */
119
120 #define ZERO        RCONST(0.0)
121
122
123 /* User-defined matrix accessor macro: IJth */
124
125 /* IJth is defined in order to write code which indexes into small dense
126    matrices with a (row,column) pair, where 1 <= row,column <= NVARs.
127
128    IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
129    where 1 <= i,j <= NVARs. The small matrix routines in dense.h
130    work with matrices stored by column in a 2-dimensional array. In C,
131    arrays are indexed starting at 0, not 1. */
132
133 #define IJth(a,i,j)      (a[j-1][i-1])
134
135 /* Types : UserData and PreconData
136    contain problem parameters, problem constants, preconditioner blocks,
137    pivot arrays, grid constants, and processor indices */
138
139 typedef struct {
140     realtype *p;
141     realtype q4, om, dx, dy, hdco, haco, vdco;
142     realtype uext[NVARs*(MXSUB+2)*(MYSUB+2)];
143     long int my_pe, isubx, isuby, nvmsub, nvmsub2;
144     MPI_Comm comm;
145 } *UserData;
146
147 typedef struct {
148     void *f_data;
149     realtype **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
150     long int *pivot[MXSUB][MYSUB];
151 } *PreconData;
152
153
154 /* Functions Called by the CVODES Solver */
155
156 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
157
158 static int Precond(realtype tn, N_Vector u, N_Vector fu,
159                   boolean_t jok, boolean_t *jcurPtr,
160                   realtype gamma, void *P_data,
161                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
162
163 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
164                  N_Vector r, N_Vector z,
165                  realtype gamma, realtype delta,
166                  int lr, void *P_data, N_Vector vtemp);
167
168 /* Private Helper Functions */
169
170 static void ProcessArgs(int argc, char *argv[], int my_pe,
171                       boolean_t *sensi, int *sensi_meth, boolean_t *err_con);
172 static void WrongArgs(int my_pe, char *name);
173
174 static PreconData AllocPreconData(UserData data);
175 static void FreePreconData(PreconData pdata);

```



```

176 static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
177 static void SetInitialProfiles(N_Vector u, UserData data);
178
179 static void BSend(MPI_Comm comm, int my_pe, long int isubx,
180                  long int isuby, long int dsize,
181                  long int dsizey, realtype udata[]);
182 static void BRecvPost(MPI_Comm comm, MPI_Request request[], int my_pe,
183                      long int isubx, long int isuby,
184                      long int dsize, long int dsizey,
185                      realtype uext[], realtype buffer[]);
186 static void BRecvWait(MPI_Request request[], long int isubx, long int isuby,
187                      long int dsize, realtype uext[], realtype buffer[]);
188 static void ucomm(realtype t, N_Vector u, UserData data);
189 static void fcalc(realtype t, realtype udata[], realtype dudata[], UserData data);
190
191 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
192                       realtype t, N_Vector u);
193 static void PrintOutputS(int my_pe, MPI_Comm comm, N_Vector *uS);
194 static void PrintFinalStats(void *cnode_mem, booleantype sensi);
195 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
196
197 /*
198  *-----
199  * MAIN PROGRAM
200  *-----
201  */
202
203 int main(int argc, char *argv[])
204 {
205     realtype abstol, reltol, t, tout;
206     N_Vector u;
207     UserData data;
208     PreconData predata;
209     void *cnode_mem;
210     int iout, flag, my_pe, npes;
211     long int neq, local_N;
212     MPI_Comm comm;
213
214     realtype *pbar;
215     int is, *plist;
216     N_Vector *uS;
217     booleantype sensi, err_con;
218     int sensi_meth;
219
220     u = NULL;
221     data = NULL;
222     predata = NULL;
223     cnode_mem = NULL;
224     pbar = NULL;
225     plist = NULL;
226     uS = NULL;
227
228     /* Set problem size neq */
229     neq = N_VARS*MX*MY;
230
231     /* Get processor number and total number of pe's */
232     MPI_Init(&argc, &argv);
233     comm = MPI_COMM_WORLD;
234     MPI_Comm_size(comm, &npes);

```

```

235 MPI_Comm_rank(comm, &my_pe);
236
237 if (npes != NPEX*NPEY) {
238     if (my_pe == 0)
239         fprintf(stderr,
240             "\nMPI_ERROR(0): npes=%d is not equal to NPEX*NPEY=%d\n\n",
241             npes, NPEX*NPEY);
242     MPI_Finalize();
243     return(1);
244 }
245
246 /* Process arguments */
247 ProcessArgs(argc, argv, my_pe, &sensi, &sensi_meth, &err_con);
248
249 /* Set local length */
250 local_N = NVAR*MXSUB*MYSUB;
251
252 /* Allocate and load user data block; allocate preconditioner block */
253 data = (UserData) malloc(sizeof *data);
254 data->p = NULL;
255 if (check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
256 data->p = (realtyp * ) malloc(NP*sizeof(realtyp));
257 if (check_flag((void *)data->p, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
258 InitUserData(my_pe, comm, data);
259 predata = AllocPreconData (data);
260 if (check_flag((void *)predata, "AllocPreconData", 2, my_pe)) MPI_Abort(comm, 1);
261
262 /* Allocate u, and set initial values and tolerances */
263 u = N_VNew_Parallel(comm, local_N, neq);
264 if (check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
265 SetInitialProfiles(u, data);
266 abstol = ATOL; reltol = RTOL;
267
268 /* Create CVODES object, set optional input, allocate memory */
269 ccode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
270 if (check_flag((void *)ccode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
271
272 flag = CVodeSetFdata(ccode_mem, data);
273 if (check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
274
275 flag = CVodeSetMaxNumSteps(ccode_mem, 2000);
276 if (check_flag(&flag, "CVodeSetMaxNumSteps", 1, my_pe)) MPI_Abort(comm, 1);
277
278 flag = CVodeMalloc(ccode_mem, f, T0, u, CV_SS, reltol, &abstol);
279 if (check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
280
281 /* Attach linear solver CVSPGMR */
282 flag = CVSpgrmr(ccode_mem, PREC_LEFT, 0);
283 if (check_flag(&flag, "CVSpgrmr", 1, my_pe)) MPI_Abort(comm, 1);
284
285 flag = CVSpilsSetPreconditioner(ccode_mem, Precond, PSolve, predata);
286 if (check_flag(&flag, "CVSpilsSetPreconditioner", 1, my_pe)) MPI_Abort(comm, 1);
287
288 if(my_pe == 0)
289     printf("\n2-species diurnal advection-diffusion problem\n");
290
291 /* Sensitivity-related settings */
292 if( sensi) {
293

```



```

353     if (my_pe == 0)
354         printf("-----\n");
355 }
356
357 /* Print final statistics */
358 if (my_pe == 0) PrintFinalStats(cvode_mem, sensi);
359
360 /* Free memory */
361 N_VDestroy_Parallel(u);
362 if (sensi) {
363     N_VDestroyVectorArray_Parallel(uS, NS);
364     free(plist);
365     free(pbar);
366 }
367 free(data->p);
368 free(data);
369 FreePreconData(predata);
370 CVodeFree(&cvode_mem);
371
372 MPI_Finalize();
373
374 return(0);
375 }
376
377 /*
378 *-----
379 * FUNCTIONS CALLED BY CVODES
380 *-----
381 */
382
383 /*
384 * f routine. Evaluate f(t,y). First call ucomm to do communication of
385 * subgrid boundary data into uest. Then calculate f by a call to fcalc.
386 */
387
388 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
389 {
390     realtype *udata, *dudata;
391     UserData data;
392
393     udata = NV_DATA_P(u);
394     dudata = NV_DATA_P(udot);
395     data = (UserData) f_data;
396
397     /* Call ucomm to do inter-processor communicaiton */
398     ucomm (t, u, data);
399
400     /* Call fcalc to calculate all right-hand sides */
401     fcalc (t, udata, dudata, data);
402
403     return(0);
404 }
405
406 /*
407 * Preconditioner setup routine. Generate and preprocess P.
408 */
409
410 static int Precond(realtype tn, N_Vector u, N_Vector fu,
411                   booleantype jok, booleantype *jcurPtr,

```

```

412         realtype gamma, void *P_data,
413         N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
414 {
415     realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
416     realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
417     int ier;
418     long int nvmsub, *(*pivot)[MYSUB], offset;
419     int lx, ly, jx, jy, isubx, isuby;
420     realtype *udata, **a, **j;
421     PreconData predata;
422     UserData data;
423     realtype Q1, Q2, C3, A3, A4, KH, VEL, KVO;
424
425     /* Make local copies of pointers in P_data, pointer to u's data,
426        and PE index pair */
427     predata = (PreconData) P_data;
428     data = (UserData) (predata->f_data);
429     P = predata->P;
430     Jbd = predata->Jbd;
431     pivot = predata->pivot;
432     udata = NV_DATA_P(u);
433     isubx = data->isubx;    isuby = data->isuby;
434     nvmsub = data->nvmsub;
435
436     /* Load problem coefficients and parameters */
437     Q1 = data->p[0];
438     Q2 = data->p[1];
439     C3 = data->p[2];
440     A3 = data->p[3];
441     A4 = data->p[4];
442     KH = data->p[5];
443     VEL = data->p[6];
444     KVO = data->p[7];
445
446     if (jok) { /* jok = TRUE: Copy Jbd to P */
447
448         for (ly = 0; ly < MYSUB; ly++)
449             for (lx = 0; lx < MXSUB; lx++)
450                 dencopy(Jbd[lx][ly], P[lx][ly], Nvars, Nvars);
451         *jcurPtr = FALSE;
452
453     } else { /* jok = FALSE: Generate Jbd from scratch and copy to P */
454
455         /* Make local copies of problem variables, for efficiency */
456         q4coef = data->q4;
457         dely = data->dy;
458         verdco = data->vdco;
459         hordco = data->hdco;
460
461         /* Compute 2x2 diagonal Jacobian blocks (using q4 values
462            computed on the last f call). Load into P. */
463         for (ly = 0; ly < MYSUB; ly++) {
464             jy = ly + isuby*MYSUB;
465             ydn = YMIN + (jy - RCONST(0.5))*dely;
466             yup = ydn + dely;
467             cydn = verdco*EXP(RCONST(0.2)*ydn);
468             cyup = verdco*EXP(RCONST(0.2)*yup);
469             diag = -(cydn + cyup + RCONST(2.0)*hordco);
470             for (lx = 0; lx < MXSUB; lx++) {

```

```

471         jx = lx + isubx*MXSUB;
472         offset = lx*NVARs + ly*nvmxsub;
473         c1 = udata[offset];
474         c2 = udata[offset+1];
475         j = Jbd[lx][ly];
476         a = P[lx][ly];
477         IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
478         IJth(j,1,2) = -Q2*c1 + q4coef;
479         IJth(j,2,1) = Q1*C3 - Q2*c2;
480         IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
481         dencopy(j, a, NVARS, NVARS);
482     }
483 }
484
485 *jcurPtr = TRUE;
486
487 }
488
489 /* Scale by -gamma */
490 for (ly = 0; ly < MYSUB; ly++)
491     for (lx = 0; lx < MXSUB; lx++)
492         denscale(-gamma, P[lx][ly], NVARS, NVARS);
493
494 /* Add identity matrix and do LU decompositions on blocks in place */
495 for (lx = 0; lx < MXSUB; lx++) {
496     for (ly = 0; ly < MYSUB; ly++) {
497         denaddI(P[lx][ly], NVARS);
498         ier = denGETRF(P[lx][ly], NVARS, NVARS, pivot[lx][ly]);
499         if (ier != 0) return(1);
500     }
501 }
502
503 return(0);
504 }
505
506 /*
507 * Preconditioner solve routine
508 */
509
510 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
511                  N_Vector r, N_Vector z,
512                  realtype gamma, realtype delta,
513                  int lr, void *P_data, N_Vector vtemp)
514 {
515     realtype **(*P)[MYSUB];
516     long int nvmxsub, *(*pivot)[MYSUB];
517     int lx, ly;
518     realtype *zdata, *v;
519     PreconData predata;
520     UserData data;
521
522     /* Extract the P and pivot arrays from P_data */
523     predata = (PreconData) P_data;
524     data = (UserData) (predata->f_data);
525     P = predata->P;
526     pivot = predata->pivot;
527
528     /* Solve the block-diagonal system Px = r using LU factors stored
529        in P and pivot data in pivot, and return the solution in z.

```

```

530     First copy vector r to z. */
531     N_VScale(RCONST(1.0), r, z);
532
533     nvmxsub = data->nvmxsub;
534     zdata = NV_DATA_P(z);
535
536     for (lx = 0; lx < MXSUB; lx++) {
537         for (ly = 0; ly < MYSUB; ly++) {
538             v = &(zdata[lx*NVAR + ly*nvmxsub]);
539             denGETRS(P[lx][ly], NVAR, pivot[lx][ly], v);
540         }
541     }
542
543     return(0);
544 }
545
546 /*
547 -----
548 * PRIVATE FUNCTIONS
549 -----
550 */
551
552 /*
553 * Process and verify arguments to cvsfwdkryx_p.
554 */
555
556 static void ProcessArgs(int argc, char *argv[], int my_pe,
557                        booleantype *sensi, int *sensi_meth, booleantype *err_con)
558 {
559     *sensi = FALSE;
560     *sensi_meth = -1;
561     *err_con = FALSE;
562
563     if (argc < 2) WrongArgs(my_pe, argv[0]);
564
565     if (strcmp(argv[1], "-nosensi") == 0)
566         *sensi = FALSE;
567     else if (strcmp(argv[1], "-sensi") == 0)
568         *sensi = TRUE;
569     else
570         WrongArgs(my_pe, argv[0]);
571
572     if (*sensi) {
573
574         if (argc != 4)
575             WrongArgs(my_pe, argv[0]);
576
577         if (strcmp(argv[2], "sim") == 0)
578             *sensi_meth = CV_SIMULTANEOUS;
579         else if (strcmp(argv[2], "stg") == 0)
580             *sensi_meth = CV_STAGGERED;
581         else if (strcmp(argv[2], "stg1") == 0)
582             *sensi_meth = CV_STAGGERED1;
583         else
584             WrongArgs(my_pe, argv[0]);
585
586         if (strcmp(argv[3], "t") == 0)
587             *err_con = TRUE;
588         else if (strcmp(argv[3], "f") == 0)

```

```

589     *err_con = FALSE;
590     else
591         WrongArgs(my_pe, argv[0]);
592 }
593
594 }
595
596 static void WrongArgs(int my_pe, char *name)
597 {
598     if (my_pe == 0) {
599         printf("\nUsage: %s [-nosensi] [-sensi_sensi_meth_err_con]\n", name);
600         printf("          sensi_meth= sim, stg, or stg1\n");
601         printf("          err_con= t or f\n");
602     }
603     MPI_Finalize();
604     exit(0);
605 }
606
607
608 /*
609  * Allocate memory for data structure of type PreconData.
610  */
611
612 static PreconData AllocPreconData(UserData fdata)
613 {
614     int lx, ly;
615     PreconData pdata;
616
617     pdata = (PreconData) malloc(sizeof *pdata);
618     pdata->f_data = fdata;
619
620     for (lx = 0; lx < MXSUB; lx++) {
621         for (ly = 0; ly < MYSUB; ly++) {
622             (pdata->P)[lx][ly] = denalloc(NVARS, NVARS);
623             (pdata->Jbd)[lx][ly] = denalloc(NVARS, NVARS);
624             (pdata->pivot)[lx][ly] = denallocpiv(NVARS);
625         }
626     }
627
628     return(pdata);
629 }
630
631 /*
632  * Free preconditioner memory.
633  */
634
635 static void FreePreconData(PreconData pdata)
636 {
637     int lx, ly;
638
639     for (lx = 0; lx < MXSUB; lx++) {
640         for (ly = 0; ly < MYSUB; ly++) {
641             denfree((pdata->P)[lx][ly]);
642             denfree((pdata->Jbd)[lx][ly]);
643             denfreepiv((pdata->pivot)[lx][ly]);
644         }
645     }
646
647     free(pdata);

```



```

648 }
649
650 /*
651  * Set user data.
652  */
653
654 static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
655 {
656     long int isubx, isuby;
657     realtype KH, VEL, KVO;
658
659     /* Set problem parameters */
660     data->p[0] = RCONST(1.63e-16); /* Q1 coeffs. q1, q2, c3 */
661     data->p[1] = RCONST(4.66e-16); /* Q2 */
662     data->p[2] = RCONST(3.7e16); /* C3 */
663     data->p[3] = RCONST(22.62); /* A3 coeff. in expression for q3(t) */
664     data->p[4] = RCONST(7.601); /* A4 coeff. in expression for q4(t) */
665     KH = data->p[5] = RCONST(4.0e-6); /* KH horizontal diffusivity Kh */
666     VEL = data->p[6] = RCONST(0.001); /* VEL advection velocity V */
667     KVO = data->p[7] = RCONST(1.0e-8); /* KVO coeff. in Kv(z) */
668
669     /* Set problem constants */
670     data->om = PI/HALFDAY;
671     data->dx = (XMAX-XMIN)/((realtype)(MX-1));
672     data->dy = (YMAX-YMIN)/((realtype)(MY-1));
673     data->hdco = KH/SQR(data->dx);
674     data->haco = VEL/(RCONST(2.0)*data->dx);
675     data->vdco = (RCONST(1.0)/SQR(data->dy))*KVO;
676
677     /* Set machine-related constants */
678     data->comm = comm;
679     data->my_pe = my_pe;
680
681     /* isubx and isuby are the PE grid indices corresponding to my_pe */
682     isuby = my_pe/NPEX;
683     isubx = my_pe - isuby*NPEX;
684     data->isubx = isubx;
685     data->isuby = isuby;
686
687     /* Set the sizes of a boundary x-line in u and uext */
688     data->nvmxsub = NVAR*MXSUB;
689     data->nvmxsub2 = NVAR*(MXSUB+2);
690 }
691
692 /*
693  * Set initial conditions in u.
694  */
695
696 static void SetInitialProfiles(N_Vector u, UserData data)
697 {
698     long int isubx, isuby, lx, ly, jx, jy, offset;
699     realtype dx, dy, x, y, cx, cy, xmid, ymid;
700     realtype *udata;
701
702     /* Set pointer to data array in vector u */
703     udata = NV_DATA_P(u);
704
705     /* Get mesh spacings, and subgrid indices for this PE */
706     dx = data->dx; dy = data->dy;

```

```

707     isubx = data->isubx;    isuby = data->isuby;
708
709     /* Load initial profiles of c1 and c2 into local u vector.
710     Here lx and ly are local mesh point indices on the local subgrid,
711     and jx and jy are the global mesh point indices. */
712     offset = 0;
713     xmid = RCONST(0.5)*(XMIN + XMAX);
714     ymid = RCONST(0.5)*(YMIN + YMAX);
715     for (ly = 0; ly < MYSUB; ly++) {
716         jy = ly + isuby*MYSUB;
717         y = YMIN + jy*dy;
718         cy = SQR(RCONST(0.1)*(y - ymid));
719         cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
720         for (lx = 0; lx < MXSUB; lx++) {
721             jx = lx + isubx*MXSUB;
722             x = XMIN + jx*dx;
723             cx = SQR(RCONST(0.1)*(x - xmid));
724             cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
725             udata[offset] = C1_SCALE*cx*cy;
726             udata[offset+1] = C2_SCALE*cx*cy;
727             offset = offset + 2;
728         }
729     }
730 }
731
732 /*
733  * Routine to send boundary data to neighboring PEs.
734  */
735
736 static void BSend(MPI_Comm comm, int my_pe, long int isubx,
737                  long int isuby, long int dsizex, long int dsizey,
738                  realtype udata[])
739 {
740     int i, ly;
741     long int offsetu, offsetbuf;
742     realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
743
744     /* If isuby > 0, send data from bottom x-line of u */
745     if (isuby != 0)
746         MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
747
748     /* If isuby < NPEY-1, send data from top x-line of u */
749     if (isuby != NPEY-1) {
750         offsetu = (MYSUB-1)*dsizex;
751         MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
752     }
753
754     /* If isubx > 0, send data from left y-line of u (via bufleft) */
755     if (isubx != 0) {
756         for (ly = 0; ly < MYSUB; ly++) {
757             offsetbuf = ly*NVARS;
758             offsetu = ly*dsizex;
759             for (i = 0; i < NVARS; i++)
760                 bufleft[offsetbuf+i] = udata[offsetu+i];
761         }
762         MPI_Send(&bufleft[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
763     }
764
765     /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */

```

```

766     if (isubx != NPEX-1) {
767         for (ly = 0; ly < MYSUB; ly++) {
768             offsetbuf = ly*NVARs;
769             offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARs;
770             for (i = 0; i < NVARs; i++)
771                 bufright[offsetbuf+i] = udata[offsetu+i];
772         }
773         MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
774     }
775 }
776
777 /*
778  * Routine to start receiving boundary data from neighboring PEs.
779  * Notes:
780  * 1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
781  *    passed to both the BRecvPost and BRecvWait functions, and should not
782  *    be manipulated between the two calls.
783  * 2) request should have 4 entries, and should be passed in both calls also.
784  */
785
786 static void BRecvPost(MPI_Comm comm, MPI_Request request[], int my_pe,
787                     long int isubx, long int isuby,
788                     long int dsizey, long int dsizey,
789                     realtype uext[], realtype buffer[])
790 {
791     long int offsetue;
792
793     /* Have bufleft and bufright use the same buffer */
794     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
795
796     /* If isuby > 0, receive data for bottom x-line of uext */
797     if (isuby != 0)
798         MPI_Irecv(&uext[NVARs], dsizey, PVEC_REAL_MPI_TYPE,
799                 my_pe-NPEX, 0, comm, &request[0]);
800
801     /* If isuby < NPEY-1, receive data for top x-line of uext */
802     if (isuby != NPEY-1) {
803         offsetue = NVARs*(1 + (MYSUB+1)*(MXSUB+2));
804         MPI_Irecv(&uext[offsetue], dsizey, PVEC_REAL_MPI_TYPE,
805                 my_pe+NPEX, 0, comm, &request[1]);
806     }
807
808     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
809     if (isubx != 0) {
810         MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
811                 my_pe-1, 0, comm, &request[2]);
812     }
813
814     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
815     if (isubx != NPEX-1) {
816         MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
817                 my_pe+1, 0, comm, &request[3]);
818     }
819 }
820
821 /*
822  * Routine to finish receiving boundary data from neighboring PEs.
823  * Notes:
824  * 1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be

```

```

825 *      passed to both the BRecvPost and BRecvWait functions, and should not
826 *      be manipulated between the two calls.
827 * 2) request should have 4 entries, and should be passed in both calls also.
828 */
829
830 static void BRecvWait(MPI_Request request[], long int isubx, long int isuby,
831                      long int dsize, realtype uext[], realtype buffer[])
832 {
833     int i, ly;
834     long int dsize2, offsetue, offsetbuf;
835     realtype *bufleft = buffer, *bufright = buffer+NVAR*MYSUB;
836     MPI_Status status;
837
838     dsize2 = dsize + 2*NVAR;
839
840     /* If isuby > 0, receive data for bottom x-line of uext */
841     if (isuby != 0)
842         MPI_Wait(&request[0], &status);
843
844     /* If isuby < NPEY-1, receive data for top x-line of uext */
845     if (isuby != NPEY-1)
846         MPI_Wait(&request[1], &status);
847
848     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
849     if (isubx != 0) {
850         MPI_Wait(&request[2], &status);
851
852         /* Copy the buffer to uext */
853         for (ly = 0; ly < MYSUB; ly++) {
854             offsetbuf = ly*NVAR;
855             offsetue = (ly+1)*dsize2;
856             for (i = 0; i < NVAR; i++)
857                 uext[offsetue+i] = bufleft[offsetbuf+i];
858         }
859     }
860
861     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
862     if (isubx != NPEX-1) {
863         MPI_Wait(&request[3], &status);
864
865         /* Copy the buffer to uext */
866         for (ly = 0; ly < MYSUB; ly++) {
867             offsetbuf = ly*NVAR;
868             offsetue = (ly+2)*dsize2 - NVAR;
869             for (i = 0; i < NVAR; i++)
870                 uext[offsetue+i] = bufright[offsetbuf+i];
871         }
872     }
873 }
874
875 /*
876 * ucomm routine. This routine performs all communication
877 * between processors of data needed to calculate f.
878 */
879
880 static void ucomm(realtype t, N_Vector u, UserData data)
881 {
882     realtype *udata, *uext, buffer[2*NVAR*MYSUB];

```

```

884     MPI_Comm comm;
885     int my_pe;
886     long int isubx, isuby, nvmxsub, nvmysub;
887     MPI_Request request[4];
888
889     udata = NV_DATA_P(u);
890
891     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
892     comm = data->comm; my_pe = data->my_pe;
893     isubx = data->isubx; isuby = data->isuby;
894     nvmxsub = data->nvmxsub;
895     nvmysub = NVARSMYSUB;
896     uext = data->uext;
897
898     /* Start receiving boundary data from neighboring PEs */
899     BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);
900
901     /* Send data from boundary of local grid to neighboring PEs */
902     BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, udata);
903
904     /* Finish receiving boundary data from neighboring PEs */
905     BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
906 }
907
908 /*
909  * fcalc routine. Compute f(t,y). This routine assumes that communication
910  * between processors of data needed to calculate f has already been done,
911  * and this data is in the work array uext.
912  */
913
914 static void fcalc(realtype t, realtype udata[], realtype dudata[], UserData data)
915 {
916     realtype *uext;
917     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
918     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
919     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
920     realtype q4coef, dely, verdco, hordco, horaco;
921     int i, lx, ly, jx, jy;
922     long int isubx, isuby, nvmxsub, nvmxsub2, offsetu, offsetue;
923     realtype Q1, Q2, C3, A3, A4, KH, VEL, KVO;
924
925     /* Get subgrid indices, data sizes, extended work array uext */
926     isubx = data->isubx; isuby = data->isuby;
927     nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
928     uext = data->uext;
929
930     /* Load problem coefficients and parameters */
931     Q1 = data->p[0];
932     Q2 = data->p[1];
933     C3 = data->p[2];
934     A3 = data->p[3];
935     A4 = data->p[4];
936     KH = data->p[5];
937     VEL = data->p[6];
938     KVO = data->p[7];
939
940     /* Copy local segment of u vector into the working extended array uext */
941     offsetu = 0;
942     offsetue = nvmxsub2 + NVARSMYSUB;

```

```

943     for (ly = 0; ly < MYSUB; ly++) {
944         for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
945         offsetu = offsetu + nvmsub;
946         offsetue = offsetue + nvmsub2;
947     }
948
949     /* To facilitate homogeneous Neumann boundary conditions, when this is
950     a boundary PE, copy data from the first interior mesh line of u to uext */
951
952     /* If isuby = 0, copy x-line 2 of u to uext */
953     if (isuby == 0) {
954         for (i = 0; i < nvmsub; i++) uext[NVARS+i] = udata[nvmsub+i];
955     }
956
957     /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
958     if (isuby == NPEY-1) {
959         offsetu = (MYSUB-2)*nvmsub;
960         offsetue = (MYSUB+1)*nvmsub2 + NVARS;
961         for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
962     }
963
964     /* If isubx = 0, copy y-line 2 of u to uext */
965     if (isubx == 0) {
966         for (ly = 0; ly < MYSUB; ly++) {
967             offsetu = ly*nvmsub + NVARS;
968             offsetue = (ly+1)*nvmsub2;
969             for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
970         }
971     }
972
973     /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
974     if (isubx == NPEX-1) {
975         for (ly = 0; ly < MYSUB; ly++) {
976             offsetu = (ly+1)*nvmsub - 2*NVARS;
977             offsetue = (ly+2)*nvmsub2 - NVARS;
978             for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
979         }
980     }
981
982     /* Make local copies of problem variables, for efficiency */
983     dely = data->dy;
984     verdco = data->vdco;
985     hordco = data->hdco;
986     horaco = data->haco;
987
988     /* Set diurnal rate coefficients as functions of t, and save q4 in
989     data block for use by preconditioner evaluation routine */
990     s = sin((data->om)*t);
991     if (s > ZERO) {
992         q3 = EXP(-A3/s);
993         q4coef = EXP(-A4/s);
994     } else {
995         q3 = ZERO;
996         q4coef = ZERO;
997     }
998     data->q4 = q4coef;
999
1000    /* Loop over all grid points in local subgrid */
1001    for (ly = 0; ly < MYSUB; ly++) {

```

```

1002     jy = ly + isuby*MYSUB;
1003
1004     /* Set vertical diffusion coefficients at jy +/- 1/2 */
1005     ydn = YMIN + (jy - .5)*dely;
1006     yup = ydn + dely;
1007     cydn = verdco*EXP(RCONST(0.2)*ydn);
1008     cyup = verdco*EXP(RCONST(0.2)*yup);
1009     for (lx = 0; lx < MXSUB; lx++) {
1010         jx = lx + isubx*MXSUB;
1011
1012         /* Extract c1 and c2, and set kinetic rate terms */
1013         offsetue = (lx+1)*NVARs + (ly+1)*nvmxsub2;
1014         c1 = uext[offsetue];
1015         c2 = uext[offsetue+1];
1016         qq1 = Q1*c1*C3;
1017         qq2 = Q2*c1*c2;
1018         qq3 = q3*C3;
1019         qq4 = q4coef*c2;
1020         rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
1021         rkin2 = qq1 - qq2 - qq4;
1022
1023         /* Set vertical diffusion terms */
1024         c1dn = uext[offsetue-nvmxsub2];
1025         c2dn = uext[offsetue-nvmxsub2+1];
1026         c1up = uext[offsetue+nvmxsub2];
1027         c2up = uext[offsetue+nvmxsub2+1];
1028         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
1029         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
1030
1031         /* Set horizontal diffusion and advection terms */
1032         c1lt = uext[offsetue-2];
1033         c2lt = uext[offsetue-1];
1034         c1rt = uext[offsetue+2];
1035         c2rt = uext[offsetue+3];
1036         hord1 = hordco*(c1rt - 2.0*c1 + c1lt);
1037         hord2 = hordco*(c2rt - 2.0*c2 + c2lt);
1038         horad1 = horaco*(c1rt - c1lt);
1039         horad2 = horaco*(c2rt - c2lt);
1040
1041         /* Load all terms into dudata */
1042         offsetu = lx*NVARs + ly*nvmxsub;
1043         dudata[offsetu] = vertd1 + hord1 + horad1 + rkin1;
1044         dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
1045     }
1046 }
1047
1048 }
1049
1050 /*
1051  * Print current t, step count, order, stepsize, and sampled c1,c2 values.
1052  */
1053
1054 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
1055                        realtype t, N_Vector u)
1056 {
1057     long int nst;
1058     int qu, flag;
1059     realtype hu, *udata, tempu[2];
1060     long int npelast, i0, i1;

```

```

1061     MPI_Status status;
1062
1063     npelast = NPEX*NPEY - 1;
1064     udata = NV_DATA_P(u);
1065
1066     /* Send c at top right mesh point to PE 0 */
1067     if (my_pe == npelast) {
1068         i0 = NVAR*MXSUB*MYSUB - 2;
1069         i1 = i0 + 1;
1070         if (npelast != 0)
1071             MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
1072         else {
1073             tempu[0] = udata[i0];
1074             tempu[1] = udata[i1];
1075         }
1076     }
1077
1078     /* On PE 0, receive c at top right, then print performance data
1079        and sampled solution values */
1080     if (my_pe == 0) {
1081
1082         if (npelast != 0)
1083             MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
1084
1085         flag = CNodeGetNumSteps(cnode_mem, &nst);
1086         check_flag(&flag, "CNodeGetNumSteps", 1, my_pe);
1087         flag = CNodeGetLastOrder(cnode_mem, &qu);
1088         check_flag(&flag, "CNodeGetLastOrder", 1, my_pe);
1089         flag = CNodeGetLastStep(cnode_mem, &hu);
1090         check_flag(&flag, "CNodeGetLastStep", 1, my_pe);
1091
1092         #if defined(SUNDIALS_EXTENDED_PRECISION)
1093             printf("%8.3Le_%2d_%8.3Le_%5ld\n", t, qu, hu, nst);
1094         #elif defined(SUNDIALS_DOUBLE_PRECISION)
1095             printf("%8.3le_%2d_%8.3le_%5ld\n", t, qu, hu, nst);
1096         #else
1097             printf("%8.3e_%2d_%8.3e_%5ld\n", t, qu, hu, nst);
1098         #endif
1099
1100         printf("Solution");
1101         #if defined(SUNDIALS_EXTENDED_PRECISION)
1102             printf("%12.4Le_%12.4Le\n", udata[0], tempu[0]);
1103         #elif defined(SUNDIALS_DOUBLE_PRECISION)
1104             printf("%12.4le_%12.4le\n", udata[0], tempu[0]);
1105         #else
1106             printf("%12.4e_%12.4e\n", udata[0], tempu[0]);
1107         #endif
1108
1109         printf(" ");
1110
1111         #if defined(SUNDIALS_EXTENDED_PRECISION)
1112             printf("%12.4Le_%12.4Le\n", udata[1], tempu[1]);
1113         #elif defined(SUNDIALS_DOUBLE_PRECISION)
1114             printf("%12.4le_%12.4le\n", udata[1], tempu[1]);
1115         #else
1116             printf("%12.4e_%12.4e\n", udata[1], tempu[1]);
1117         #endif
1118     }
1119 }

```



```

1120 }
1121 }
1122
1123 /*
1124  * Print sampled sensitivity values.
1125  */
1126
1127 static void PrintOutputS(int my_pe, MPI_Comm comm, N_Vector *uS)
1128 {
1129     realtype *sdata, temps[2];
1130     long int npelast, i0, i1;
1131     MPI_Status status;
1132
1133     npelast = NPEX*NPEY - 1;
1134
1135     sdata = NV_DATA_P(uS[0]);
1136
1137     /* Send s1 at top right mesh point to PE 0 */
1138     if (my_pe == npelast) {
1139         i0 = NVAR*MXSUB*MYSUB - 2;
1140         i1 = i0 + 1;
1141         if (npelast != 0)
1142             MPI_Send(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
1143         else {
1144             temps[0] = sdata[i0];
1145             temps[1] = sdata[i1];
1146         }
1147     }
1148
1149     /* On PE 0, receive s1 at top right, then print sampled sensitivity values */
1150     if (my_pe == 0) {
1151         if (npelast != 0)
1152             MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
1153         printf("~~~~~\n");
1154         printf("Sensitivity_1");
1155         #if defined(SUNDIALS_EXTENDED_PRECISION)
1156             printf("%12.4Le_12.4Le\n", sdata[0], temps[0]);
1157         #elif defined(SUNDIALS_DOUBLE_PRECISION)
1158             printf("%12.4le_12.4le\n", sdata[0], temps[0]);
1159         #else
1160             printf("%12.4e_12.4e\n", sdata[0], temps[0]);
1161         #endif
1162         printf("~~~~~");
1163         #if defined(SUNDIALS_EXTENDED_PRECISION)
1164             printf("%12.4Le_12.4Le\n", sdata[1], temps[1]);
1165         #elif defined(SUNDIALS_DOUBLE_PRECISION)
1166             printf("%12.4le_12.4le\n", sdata[1], temps[1]);
1167         #else
1168             printf("%12.4e_12.4e\n", sdata[1], temps[1]);
1169         #endif
1170     }
1171
1172     sdata = NV_DATA_P(uS[1]);
1173
1174     /* Send s2 at top right mesh point to PE 0 */
1175     if (my_pe == npelast) {
1176         i0 = NVAR*MXSUB*MYSUB - 2;
1177         i1 = i0 + 1;
1178         if (npelast != 0)

```

```

1179 MPI_Send(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
1180 else {
1181     temps[0] = sdata[i0];
1182     temps[1] = sdata[i1];
1183 }
1184 }
1185
1186 /* On PE 0, receive s2 at top right, then print sampled sensitivity values */
1187 if (my_pe == 0) {
1188     if (npelast != 0)
1189         MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
1190     printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%-----\n");
1191     printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Sensitivity_u2_u");
1192 #if defined(SUNDIALS_EXTENDED_PRECISION)
1193     printf("%12.4Le_u%12.4Le_u\n", sdata[0], temps[0]);
1194 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1195     printf("%12.4le_u%12.4le_u\n", sdata[0], temps[0]);
1196 #else
1197     printf("%12.4e_u%12.4e_u\n", sdata[0], temps[0]);
1198 #endif
1199     printf("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");
1200 #if defined(SUNDIALS_EXTENDED_PRECISION)
1201     printf("%12.4Le_u%12.4Le_u\n", sdata[1], temps[1]);
1202 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1203     printf("%12.4le_u%12.4le_u\n", sdata[1], temps[1]);
1204 #else
1205     printf("%12.4e_u%12.4e_u\n", sdata[1], temps[1]);
1206 #endif
1207 }
1208 }
1209
1210 /*
1211  * Print final statistics from the CVODES memory.
1212  */
1213
1214 static void PrintFinalStats(void *cvmem, booleantype sensi)
1215 {
1216     long int nst;
1217     long int nfe, nsetups, nni, ncfn, netf;
1218     long int nfSe, nfeS, nsetupsS, nniS, ncfnS, netfS;
1219     int flag;
1220
1221     flag = CVodeGetNumSteps(cvmem, &nst);
1222     check_flag(&flag, "CVodeGetNumSteps", 1, 0);
1223     flag = CVodeGetNumRhsEvals(cvmem, &nfe);
1224     check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
1225     flag = CVodeGetNumLinSolvSetups(cvmem, &nsetups);
1226     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
1227     flag = CVodeGetNumErrTestFails(cvmem, &netf);
1228     check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
1229     flag = CVodeGetNumNonlinSolvIters(cvmem, &nni);
1230     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
1231     flag = CVodeGetNumNonlinSolvConvFails(cvmem, &ncfn);
1232     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
1233
1234     if (sensi) {
1235         flag = CVodeGetNumSensRhsEvals(cvmem, &nfSe);
1236         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1, 0);
1237         flag = CVodeGetNumRhsEvalsSens(cvmem, &nfeS);

```

```

1238     check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1, 0);
1239     flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
1240     check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1, 0);
1241     flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
1242     check_flag(&flag, "CVodeGetNumSensErrTestFails", 1, 0);
1243     flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
1244     check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1, 0);
1245     flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
1246     check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1, 0);
1247 }
1248
1249 printf("\nFinal Statistics\n\n");
1250 printf("nst===== %5ld\n\n", nst);
1251 printf("nfe===== %5ld\n", nfe);
1252 printf("netf===== %5ld nsetups= %5ld\n", netf, nsetups);
1253 printf("nni===== %5ld ncfns===== %5ld\n", nni, ncfns);
1254
1255 if(sensi) {
1256     printf("\n");
1257     printf("nfSe===== %5ld nfeS===== %5ld\n", nfSe, nfeS);
1258     printf("netfs===== %5ld nsetupsS= %5ld\n", netfS, nsetupsS);
1259     printf("nniS===== %5ld ncfns===== %5ld\n", nniS, ncfns);
1260 }
1261
1262 }
1263
1264 /*
1265  * Check function return value...
1266  *   opt == 0 means SUNDIALS function allocates memory so check if
1267  *   returned NULL pointer
1268  *   opt == 1 means SUNDIALS function returns a flag so check if
1269  *   flag >= 0
1270  *   opt == 2 means function allocates memory so check if returned
1271  *   NULL pointer
1272  */
1273
1274 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
1275 {
1276     int *errflag;
1277
1278     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
1279     if (opt == 0 && flagvalue == NULL) {
1280         fprintf(stderr,
1281             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
1282             id, funcname);
1283         return(1); }
1284
1285     /* Check if flag < 0 */
1286     else if (opt == 1) {
1287         errflag = (int *) flagvalue;
1288         if (*errflag < 0) {
1289             fprintf(stderr,
1290                 "\nSUNDIALS_ERROR(%d): %s() failed with flag= %d\n\n",
1291                 id, funcname, *errflag);
1292             return(1); } }
1293
1294     /* Check if function returned NULL pointer - no memory allocated */
1295     else if (opt == 2 && flagvalue == NULL) {
1296         fprintf(stderr,

```

```
1297         "\nMEMORY_ERROR(%d):_%s()_failed_-_returned_NULL_pointer\n\n",
1298         id, funcname);
1299     return(1); }
1300
1301     return(0);
1302 }
```

D listing of cvsadjdenx.c

```
1  /*
2  * -----
3  * $Revision: 1.3 $
4  * $Date: 2006/08/10 23:17:22 $
5  * -----
6  * Programmer(s): Radu Serban @ LLNL
7  * -----
8  * Copyright (c) 2002, The Regents of the University of California.
9  * Produced at the Lawrence Livermore National Laboratory.
10 * All rights reserved.
11 * For details, see the LICENSE file.
12 * -----
13 * Adjoint sensitivity example problem.
14 * The following is a simple example problem, with the coding
15 * needed for its solution by CVODES. The problem is from chemical
16 * kinetics, and consists of the following three rate equations.
17 *   dy1/dt = -p1*y1 + p2*y2*y3
18 *   dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
19 *   dy3/dt =  p3*(y2)^2
20 * on the interval from t = 0.0 to t = 4.e10, with initial
21 * conditions: y1 = 1.0, y2 = y3 = 0. The reaction rates are:
22 * p1=0.04, p2=1e4, and p3=3e7. The problem is stiff.
23 * This program solves the problem with the BDF method, Newton
24 * iteration with the CVODE dense linear solver, and a user-supplied
25 * Jacobian routine.
26 * It uses a scalar relative tolerance and a vector absolute
27 * tolerance.
28 * Output is printed in decades from t = .4 to t = 4.e10.
29 * Run statistics (optional outputs) are printed at the end.
30 *
31 * Optionally, CVODES can compute sensitivities with respect to
32 * the problem parameters p1, p2, and p3 of the following quantity:
33 *   G = int_t0^t1 g(t,p,y) dt
34 * where
35 *   g(t,p,y) = y3
36 *
37 * The gradient dG/dp is obtained as:
38 *   dG/dp = int_t0^t1 (g_p - lambda^T f_p ) dt - lambda^T(t0)*y0_p
39 *           = - xi^T(t0) - lambda^T(t0)*y0_p
40 * where lambda and xi are solutions of:
41 *   d(lambda)/dt = - (f_y)^T * lambda - (g_y)^T
42 *   lambda(t1) = 0
43 * and
44 *   d(xi)/dt = - (f_p)^T * lambda + (g_p)^T
45 *   xi(t1) = 0
46 *
47 * During the backward integration, CVODES also evaluates G as
48 *   G = - phi(t0)
49 * where
50 *   d(phi)/dt = g(t,y,p)
51 *   phi(t1) = 0
52 * -----
53 */
54
55 #include <stdio.h>
56 #include <stdlib.h>
57
```

```

58 #include <cvodes/cvodes.h>
59 #include <cvodes/cvodes_dense.h>
60 #include <nvector/nvector_serial.h>
61 #include <sundials/sundials_types.h>
62 #include <sundials/sundials_math.h>
63
64 /* Accessor macros */
65
66 #define Ith(v,i)      NV_Ith_S(v,i-1)          /* i-th vector component i= 1..NEQ */
67 #define IJth(A,i,j)  DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j = 1..NEQ */
68
69 /* Problem Constants */
70
71 #define NEQ          3                /* number of equations */
72
73 #define RTOL         RCONST(1e-6)    /* scalar relative tolerance */
74
75 #define ATOL1         RCONST(1e-8)    /* vector absolute tolerance components */
76 #define ATOL2         RCONST(1e-14)
77 #define ATOL3         RCONST(1e-6)
78
79 #define ATOL1         RCONST(1e-8)    /* absolute tolerance for adjoint vars. */
80 #define ATOLq         RCONST(1e-6)    /* absolute tolerance for quadratures */
81
82 #define T0           RCONST(0.0)      /* initial time */
83 #define TOUT          RCONST(4e7)     /* final time */
84
85 #define TB1           RCONST(4e7)     /* starting point for adjoint problem */
86 #define TB2           RCONST(50.0)    /* starting point for adjoint problem */
87
88 #define STEPS         150             /* number of steps between check points */
89
90 #define NP            3               /* number of problem parameters */
91
92 #define ZERO          RCONST(0.0)
93
94
95 /* Type : UserData */
96
97 typedef struct {
98     realtype p[3];
99 } *UserData;
100
101 /* Prototypes of user-supplied functions */
102
103 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
104 static int Jac(long int N, DenseMat J, realtype t,
105               N_Vector y, N_Vector fy, void *jac_data,
106               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
107 static int fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data);
108 static int ewt(N_Vector y, N_Vector w, void *e_data);
109
110 static int fB(realtype t, N_Vector y,
111               N_Vector yB, N_Vector yBdot, void *f_dataB);
112 static int JacB(long int NB, DenseMat JB, realtype t,
113                N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
114                N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B);
115 static int fQB(realtype t, N_Vector y, N_Vector yB,
116                N_Vector qBdot, void *fQ_dataB);

```

```

117
118
119 /* Prototypes of private functions */
120
121 static void PrintOutput(realtypes tfinal, N_Vector yB, N_Vector qB);
122 static int check_flag(void *flagvalue, char *funcname, int opt);
123
124 /*
125  *-----
126  * MAIN PROGRAM
127  *-----
128  */
129
130 int main(int argc, char *argv[])
131 {
132     UserData data;
133
134     void *cvadj_mem;
135     void *cvsode_mem;
136
137     reals reltolQ, abstolQ;
138     N_Vector y, q;
139
140     int steps;
141
142     reals reltolB, abstolB, abstolQB;
143     N_Vector yB, qB;
144
145     reals time;
146     int flag, ncheck;
147
148     long int nst, nstB;
149
150     CVAdjCheckPointRec *ckpnt;
151     int i;
152
153     data = NULL;
154     cvadj_mem = cvsode_mem = NULL;
155     ckpnt = NULL;
156     y = yB = qB = NULL;
157
158     /* Print problem description */
159     printf("\nAdjoint Sensitivity Example for Chemical Kinetics\n");
160     printf("-----\n\n");
161     printf("ODE: dy1/dt = -p1*y1 + p2*y2*y3\n");
162     printf("dy2/dt = p1*y1 - p2*y2*y3 - p3*(y2)^2\n");
163     printf("dy3/dt = p3*(y2)^2\n\n");
164     printf("Find dG/dp for\n");
165     printf("G = int_t0^tB0 g(t,p,y) dt\n");
166     printf("g(t,p,y) = y3\n\n\n");
167
168     /* User data structure */
169     data = (UserData) malloc(sizeof *data);
170     if (check_flag((void *)data, "malloc", 2)) return(1);
171     data->p[0] = RCONST(0.04);
172     data->p[1] = RCONST(1.0e4);
173     data->p[2] = RCONST(3.0e7);
174
175     /* Initialize y */

```

```

176 y = N_VNew_Serial(NEQ);
177 if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
178 lth(y,1) = RCONST(1.0);
179 lth(y,2) = ZERO;
180 lth(y,3) = ZERO;
181
182 /* Initialize q */
183 q = N_VNew_Serial(1);
184 if (check_flag((void *)q, "N_VNew_Serial", 0)) return(1);
185 lth(q,1) = ZERO;
186
187 /* Set the scalar relative and absolute tolerances reltolQ and abstolQ */
188 reltolQ = RTOL;
189 abstolQ = ATOLq;
190
191 /* Create and allocate CVODES memory for forward run */
192 printf("Create and allocate CVODES memory for forward runs\n");
193
194 cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
195 if (check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);
196
197 flag = CVodeMalloc(cvode_mem, f, T0, y, CV_WF, 0.0, NULL);
198 if (check_flag(&flag, "CVodeMalloc", 1)) return(1);
199
200 flag = CVodeSetEwtFn(cvode_mem, ewt, NULL);
201 if (check_flag(&flag, "CVodeSetEwtFn", 1)) return(1);
202
203 flag = CVodeSetFdata(cvode_mem, data);
204 if (check_flag(&flag, "CVodeSetFdata", 1)) return(1);
205
206 flag = CVDense(cvode_mem, NEQ);
207 if (check_flag(&flag, "CVDense", 1)) return(1);
208
209 flag = CVDenseSetJacFn(cvode_mem, Jac, data);
210 if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
211
212 flag = CVodeQuadMalloc(cvode_mem, fQ, q);
213 if (check_flag(&flag, "CVodeQuadMalloc", 1)) return(1);
214
215 flag = CVodeSetQuadFdata(cvode_mem, data);
216 if (check_flag(&flag, "CVodeSetQuadFdata", 1)) return(1);
217
218 flag = CVodeSetQuadErrCon(cvode_mem, TRUE, CV_SS, reltolQ, &abstolQ);
219 if (check_flag(&flag, "CVodeSetQuadErrCon", 1)) return(1);
220
221 /* Allocate global memory */
222
223 steps = STEPS;
224 cvadj_mem = CVadjMalloc(cvode_mem, steps, CV_HERMITE);
225 /*
226 cvadj_mem = CVadjMalloc(cvode_mem, steps, CV_POLYNOMIAL);
227 */
228 if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0)) return(1);
229
230 /* Perform forward run */
231 printf("Forward integration...");
232
233 flag = CVodeF(cvadj_mem, TOUT, y, &time, CV_NORMAL, &ncheck);
234 if (check_flag(&flag, "CVodeF", 1)) return(1);

```



```

235     flag = CVodeGetNumSteps(cvode_mem, &nst);
236     if (check_flag(&flag, "CVodeGetNumSteps", 1)) return(1);
237
238     printf("done_\nst_\n",nst);
239
240     flag = CVodeGetQuad(cvode_mem, TOUT, q);
241     if (check_flag(&flag, "CVodeGetQuad", 1)) return(1);
242
243     printf("-----\n");
244     #if defined(SUNDIALS_EXTENDED_PRECISION)
245         printf("G: %12.4Le\n", Ith(q,1));
246     #elif defined(SUNDIALS_DOUBLE_PRECISION)
247         printf("G: %12.4le\n", Ith(q,1));
248     #else
249         printf("G: %12.4e\n", Ith(q,1));
250     #endif
251     printf("-----\n\n");
252
253     /* Test check point linked list
254        (uncomment next block to print check point information) */
255
256     /*
257     printf("\nList of Check Points (ncheck = %d)\n\n", ncheck);
258     ckpnt = (CVadjCheckPointRec *) malloc ( (ncheck+1)*sizeof(CVadjCheckPointRec));
259     CVadjGetCheckPointsInfo(cvad_j_mem, ckpnt);
260     for (i=0;i<=ncheck;i++) {
261         printf("Address:      %p\n", ckpnt[i].my_addr);
262         printf("Next:        %p\n", ckpnt[i].next_addr);
263         printf("Time interval: %le %le\n", ckpnt[i].t0, ckpnt[i].t1);
264         printf("Step number:   %ld\n", ckpnt[i].nstep);
265         printf("Order:         %d\n", ckpnt[i].order);
266         printf("Step size:      %le\n", ckpnt[i].step);
267         printf("\n");
268     }
269     */
270
271     /* Initialize yB */
272     yB = N_VNew_Serial(NEQ);
273     if (check_flag((void *)yB, "N_VNew_Serial", 0)) return(1);
274     Ith(yB,1) = ZERO;
275     Ith(yB,2) = ZERO;
276     Ith(yB,3) = ZERO;
277
278     /* Initialize qB */
279     qB = N_VNew_Serial(NP);
280     if (check_flag((void *)qB, "N_VNew", 0)) return(1);
281     Ith(qB,1) = ZERO;
282     Ith(qB,2) = ZERO;
283     Ith(qB,3) = ZERO;
284
285     /* Set the scalar relative tolerance reltolB */
286     reltolB = RTOL;
287
288     /* Set the scalar absolute tolerance abstolB */
289     abstolB = ATOLl;
290
291     /* Set the scalar absolute tolerance abstolQB */
292     abstolQB = ATOLq;
293

```

```

294  /* Create and allocate CVODES memory for backward run */
295  printf("Create and allocate CVODES memory for backward run\n");
296
297  flag = CNodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
298  if (check_flag(&flag, "CNodeCreateB", 1)) return(1);
299
300  flag = CNodeMallocB(cvadj_mem, fB, TB1, yB, CV_SS, reltolB, &abstolB);
301  if (check_flag(&flag, "CNodeMallocB", 1)) return(1);
302
303  flag = CNodeSetFdataB(cvadj_mem, data);
304  if (check_flag(&flag, "CNodeSetFdataB", 1)) return(1);
305
306  flag = CVDenseB(cvadj_mem, NEQ);
307  if (check_flag(&flag, "CVDenseB", 1)) return(1);
308
309  flag = CVDenseSetJacFnB(cvadj_mem, JacB, data);
310  if (check_flag(&flag, "CVDenseSetJacFnB", 1)) return(1);
311
312  flag = CNodeQuadMallocB(cvadj_mem, fQB, qB);
313  if (check_flag(&flag, "CNodeQuadMallocB", 1)) return(1);
314
315  flag = CNodeSetQuadFdataB(cvadj_mem, data);
316  if (check_flag(&flag, "CNodeSetQuadFdataB", 1)) return(1);
317
318  flag = CNodeSetQuadErrConB(cvadj_mem, TRUE, CV_SS, reltolB, &abstolQB);
319  if (check_flag(&flag, "CNodeSetQuadErrConB", 1)) return(1);
320
321  /* Backward Integration */
322  printf("Backward integration...");
323
324  flag = CNodeB(cvadj_mem, T0, yB, &time, CV_NORMAL);
325  if (check_flag(&flag, "CNodeB", 1)) return(1);
326  CNodeGetNumSteps(CVadjGetCNodeBmem(cvadj_mem), &nstB);
327  printf("done (nst=%ld)\n", nstB);
328
329  flag = CNodeGetQuadB(cvadj_mem, qB);
330  if (check_flag(&flag, "CNodeGetQuadB", 1)) return(1);
331
332  PrintOutput(TB1, yB, qB);
333
334  /* Reinitialize backward phase (new tB0) */
335
336  Ith(yB,1) = ZERO;
337  Ith(yB,2) = ZERO;
338  Ith(yB,3) = ZERO;
339
340  Ith(qB,1) = ZERO;
341  Ith(qB,2) = ZERO;
342  Ith(qB,3) = ZERO;
343
344  printf("Re-initialize CVODES memory for backward run\n");
345
346  flag = CNodeReInitB(cvadj_mem, fB, TB2, yB, CV_SS, reltolB, &abstolB);
347  if (check_flag(&flag, "CNodeReInitB", 1)) return(1);
348
349  flag = CNodeQuadReInitB(cvadj_mem, fQB, qB);
350  if (check_flag(&flag, "CNodeQuadReInitB", 1)) return(1);
351
352  printf("Backward integration...");

```

```

353
354     flag = CVodeB(cvadj_mem, T0, yB, &time, CV_NORMAL);
355     if (check_flag(&flag, "CVodeB", 1)) return(1);
356     CVodeGetNumSteps(CVadjGetCVodeBmem(cvadj_mem), &nstB);
357     printf("done_\nst_\nld_\n", nstB);
358
359     flag = CVodeGetQuadB(cvadj_mem, qB);
360     if (check_flag(&flag, "CVodeGetQuadB", 1)) return(1);
361
362     PrintOutput(TB2, yB, qB);
363
364     /* Free memory */
365     printf("Free_memory\n\n");
366
367     CVodeFree(&cnode_mem);
368     N_VDestroy_Serial(y);
369     N_VDestroy_Serial(q);
370     N_VDestroy_Serial(yB);
371     N_VDestroy_Serial(qB);
372     CVadjFree(&cvadj_mem);
373
374     if (ckpnt != NULL) free(ckpnt);
375     free(data);
376
377     return(0);
378 }
379
380 /*
381  *-----
382  * FUNCTIONS CALLED BY CVODES
383  *-----
384  */
385
386 /*
387  * f routine. Compute f(t,y).
388  */
389
390 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
391 {
392     realtype y1, y2, y3, yd1, yd3;
393     UserData data;
394     realtype p1, p2, p3;
395
396     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
397     data = (UserData) f_data;
398     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
399
400     yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
401     yd3 = Ith(ydot,3) = p3*y2*y2;
402     Ith(ydot,2) = -yd1 - yd3;
403
404     return(0);
405 }
406
407 /*
408  * Jacobian routine. Compute J(t,y).
409  */
410
411

```

```

412 static int Jac(long int N, DenseMat J, realtype t,
413               N_Vector y, N_Vector fy, void *jac_data,
414               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
415 {
416     realtype y1, y2, y3;
417     UserData data;
418     realtype p1, p2, p3;
419
420     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
421     data = (UserData) jac_data;
422     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
423
424     IJth(J,1,1) = -p1; IJth(J,1,2) = p2*y3; IJth(J,1,3) = p2*y2;
425     IJth(J,2,1) = p1; IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
426     IJth(J,3,2) = 2*p3*y2;
427
428     return(0);
429 }
430
431 /*
432  * fQ routine. Compute fQ(t,y).
433  */
434
435 static int fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
436 {
437     Ith(qdot,1) = Ith(y,3);
438
439     return(0);
440 }
441
442 /*
443  * EwtSet function. Computes the error weights at the current solution.
444  */
445
446 static int ewt(N_Vector y, N_Vector w, void *e_data)
447 {
448     int i;
449     realtype yy, ww, rtol, atol[3];
450
451     rtol = RTOL;
452     atol[0] = ATOL1;
453     atol[1] = ATOL2;
454     atol[2] = ATOL3;
455
456     for (i=1; i<=3; i++) {
457         yy = Ith(y,i);
458         ww = rtol * ABS(yy) + atol[i-1];
459         if (ww <= 0.0) return (-1);
460         Ith(w,i) = 1.0/ww;
461     }
462
463     return(0);
464 }
465
466 /*
467  * fB routine. Compute fB(t,y,yB).
468  */
469
470 static int fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot, void *f_dataB)

```

```

471 {
472     UserData data;
473     realtype y1, y2, y3;
474     realtype p1, p2, p3;
475     realtype l1, l2, l3;
476     realtype l21, l32, y23;
477
478     data = (UserData) f_dataB;
479
480     /* The p vector */
481     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
482
483     /* The y vector */
484     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
485
486     /* The lambda vector */
487     l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);
488
489     /* Temporary variables */
490     l21 = l2-l1;
491     l32 = l3-l2;
492     y23 = y2*y3;
493
494     /* Load yBdot */
495     Ith(yBdot,1) = - p1*l21;
496     Ith(yBdot,2) = p2*y3*l21 - RCONST(2.0)*p3*y2*l32;
497     Ith(yBdot,3) = p2*y2*l21 - RCONST(1.0);
498
499     return(0);
500 }
501
502 /*
503  * JacB routine. Compute JB(t,y,yB).
504  */
505
506 static int JacB(long int NB, DenseMat JB, realtype t,
507                 N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
508                 N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B)
509 {
510     UserData data;
511     realtype y1, y2, y3;
512     realtype p1, p2, p3;
513
514     data = (UserData) jac_dataB;
515
516     /* The p vector */
517     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
518
519     /* The y vector */
520     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
521
522     /* Load JB */
523     IJth(JB,1,1) = p1;      IJth(JB,1,2) = -p1;
524     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;
525     IJth(JB,3,1) = -p2*y2; IJth(JB,3,2) = p2*y2;
526
527     return(0);
528 }
529

```

```

530  /*
531  * fQB routine. Compute integrand for quadratures
532  */
533
534  static int fQB(realtype t, N_Vector y, N_Vector yB,
535                N_Vector qBdot, void *fQ_dataB)
536  {
537      UserData data;
538      realtype y1, y2, y3;
539      realtype p1, p2, p3;
540      realtype l1, l2, l3;
541      realtype l21, l32, y23;
542
543      data = (UserData) fQ_dataB;
544
545      /* The p vector */
546      p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
547
548      /* The y vector */
549      y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
550
551      /* The lambda vector */
552      l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);
553
554      /* Temporary variables */
555      l21 = l2-l1;
556      l32 = l3-l2;
557      y23 = y2*y3;
558
559      Ith(qBdot,1) = y1*l21;
560      Ith(qBdot,2) = - y23*l21;
561      Ith(qBdot,3) = y2*y2*l32;
562
563      return(0);
564  }
565
566  /*
567  *-----
568  * PRIVATE FUNCTIONS
569  *-----
570  */
571
572  /*
573  * Print results after backward integration
574  */
575
576  static void PrintOutput(realtype tfinal, N_Vector yB, N_Vector qB)
577  {
578      printf("-----\n");
579      #if defined(SUNDIALS_EXTENDED_PRECISION)
580          printf("tB0: %12.4Le\n", tfinal);
581          printf("dG/dp: %12.4Le %12.4Le %12.4Le\n",
582                -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
583          printf("lambda(t0): %12.4Le %12.4Le %12.4Le\n",
584                Ith(yB,1), Ith(yB,2), Ith(yB,3));
585      #elif defined(SUNDIALS_DOUBLE_PRECISION)
586          printf("tB0: %12.4le\n", tfinal);
587          printf("dG/dp: %12.4le %12.4le %12.4le\n",
588                -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));

```

```

589     printf("lambda(t0):_%.12.4le_%.12.4le_%.12.4le\n",
590            Ith(yB,1), Ith(yB,2), Ith(yB,3));
591 #else
592     printf("tB0:_.%.12.4e\n",tfinal);
593     printf("dG/dp:_.%.12.4e_%.12.4e_%.12.4e\n",
594            -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
595     printf("lambda(t0):_%.12.4e_%.12.4e_%.12.4e\n",
596            Ith(yB,1), Ith(yB,2), Ith(yB,3));
597 #endif
598     printf("-----\n\n");
599 }
600
601 /*
602  * Check function return value.
603  *   opt == 0 means SUNDIALS function allocates memory so check if
604  *   returned NULL pointer
605  *   opt == 1 means SUNDIALS function returns a flag so check if
606  *   flag >= 0
607  *   opt == 2 means function allocates memory so check if returned
608  *   NULL pointer
609  */
610
611 static int check_flag(void *flagvalue, char *funcname, int opt)
612 {
613     int *errflag;
614
615     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
616     if (opt == 0 && flagvalue == NULL) {
617         fprintf(stderr, "\nSUNDIALS_ERROR:_%s()_failed_-_returned_NULL_pointer\n\n",
618                 funcname);
619         return(1); }
620
621     /* Check if flag < 0 */
622     else if (opt == 1) {
623         errflag = (int *) flagvalue;
624         if (*errflag < 0) {
625             fprintf(stderr, "\nSUNDIALS_ERROR:_%s()_failed_with_flag=_%d\n\n",
626                     funcname, *errflag);
627             return(1); }}
628
629     /* Check if function returned NULL pointer - no memory allocated */
630     else if (opt == 2 && flagvalue == NULL) {
631         fprintf(stderr, "\nMEMORY_ERROR:_%s()_failed_-_returned_NULL_pointer\n\n",
632                 funcname);
633         return(1); }
634
635     return(0);
636 }

```

E Listing of cvsadjnonx_p.c

```

1  /*
2  * -----
3  * $Revision: 1.1 $
4  * $Date: 2006/07/05 15:50:06 $
5  * -----
6  * Programmer(s): Radu Serban @ LLNL
7  * -----
8  * Example problem:
9  *
10 * The following is a simple example problem, with the program for
11 * its solution by CVODE. The problem is the semi-discrete form of
12 * the advection-diffusion equation in 1-D:
13 *  $du/dt = p1 * d^2u / dx^2 + p2 * du / dx$ 
14 * on the interval  $0 \leq x \leq 2$ , and the time interval  $0 \leq t \leq 5$ .
15 * Homogeneous Dirichlet boundary conditions are posed, and the
16 * initial condition is:
17 *  $u(x,t=0) = x(2-x)\exp(2x)$ .
18 * The nominal values of the two parameters are:  $p1=1.0$ ,  $p2=0.5$ 
19 * The PDE is discretized on a uniform grid of size  $MX+2$  with
20 * central differencing, and with boundary values eliminated,
21 * leaving an ODE system of size  $NEQ = MX$ .
22 * This program solves the problem with the option for nonstiff
23 * systems: ADAMS method and functional iteration.
24 * It uses scalar relative and absolute tolerances.
25 *
26 * In addition to the solution, sensitivities with respect to  $p1$ 
27 * and  $p2$  as well as with respect to initial conditions are
28 * computed for the quantity:
29 *  $g(t, u, p) = \int_x u(x,t)$  at  $t = 5$ 
30 * These sensitivities are obtained by solving the adjoint system:
31 *  $dv/dt = -p1 * d^2 v / dx^2 + p2 * dv / dx$ 
32 * with homogeneous Dirichlet boundary conditions and the final
33 * condition:
34 *  $v(x,t=5) = 1.0$ 
35 * Then,  $v(x, t=0)$  represents the sensitivity of  $g(5)$  with respect
36 * to  $u(x, t=0)$  and the gradient of  $g(5)$  with respect to  $p1, p2$  is
37 *  $(dg/dp)^T = [ \int_t \int_x (v * d^2u / dx^2) dx dt ]$ 
38 *  $[ \int_t \int_x (v * du / dx) dx dt ]$ 
39 *
40 * This version uses MPI for user routines.
41 * Execute with Number of Processors = N, with  $1 \leq N \leq MX$ .
42 * -----
43 */
44
45 #include <stdio.h>
46 #include <stdlib.h>
47 #include <math.h>
48
49 #include <cvodes/cvodes.h>
50 #include <nvector/nvector_parallel.h>
51 #include <sundials/sundials_math.h>
52 #include <sundials/sundials_types.h>
53
54 #include <mpi.h>
55
56 /* Problem Constants */
57

```



```

58 #define XMAX RCONST(2.0) /* domain boundary */
59 #define MX 20 /* mesh dimension */
60 #define NEQ MX /* number of equations */
61 #define ATOL RCONST(1.e-5) /* scalar absolute tolerance */
62 #define TO RCONST(0.0) /* initial time */
63 #define TOUT RCONST(2.5) /* output time increment */
64
65 /* Adjoint Problem Constants */
66
67 #define NP 2 /* number of parameters */
68 #define STEPS 200 /* steps between check points */
69
70 #define ZERO RCONST(0.0)
71 #define ONE RCONST(1.0)
72 #define TWO RCONST(2.0)
73
74 /* Type : UserData */
75
76 typedef struct {
77     realtype p[2]; /* model parameters */
78     realtype dx; /* spatial discretization grid */
79     realtype hdcoef, hacoef; /* diffusion and advection coefficients */
80     long int local_N;
81     long int npes, my_pe; /* total number of processes and current ID */
82     long int nperpe, nrem;
83     MPI_Comm comm; /* MPI communicator */
84     realtype *z1, *z2; /* work space */
85 } *UserData;
86
87 /* Prototypes of user-supplied functions */
88
89 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
90 static int fB(realtype t, N_Vector u,
91              N_Vector uB, N_Vector uBdot, void *f_dataB);
92
93 /* Prototypes of private functions */
94
95 static void SetIC(N_Vector u, realtype dx, long int my_length, long int my_base);
96 static void SetICback(N_Vector uB, long int my_base);
97 static realtype Xintgr(realtype *z, long int l, realtype dx);
98 static realtype Compute_g(N_Vector u, UserData data);
99 static void PrintOutput(realtype g_val, N_Vector uB, UserData data);
100 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
101
102 /*
103  *-----
104  * MAIN PROGRAM
105  *-----
106  */
107
108 int main(int argc, char *argv[])
109 {
110     UserData data;
111
112     void *cvadj_mem;
113     void *cvoke_mem;
114
115     N_Vector u;
116     realtype reltol, abstol;

```

```

117
118     N_Vector uB;
119
120     realtype dx, t, g_val;
121     int flag, my_pe, nprocs, npes, ncheck;
122     long int local_N=0, nperpe, nrem, my_base=0;
123
124     MPI_Comm comm;
125
126     data = NULL;
127     cvadj_mem = ccode_mem = NULL;
128     u = uB = NULL;
129
130     /*-----
131      Initialize MPI and get total number of pe's, and my_pe
132      -----*/
133     MPI_Init(&argc, &argv);
134     comm = MPI_COMM_WORLD;
135     MPI_Comm_size(comm, &nprocs);
136     MPI_Comm_rank(comm, &my_pe);
137
138     npes = nprocs - 1; /* pe's dedicated to PDE integration */
139
140     if ( npes <= 0 ) {
141         if (my_pe == npes)
142             fprintf(stderr, "\nMPI_ERROR(%d): number of processes must be >= 2\n\n",
143                     my_pe);
144         MPI_Finalize();
145         return(1);
146     }
147
148     /*-----
149      Set local vector length
150      -----*/
151     nperpe = NEQ/npes;
152     nrem = NEQ - npes*nperpe;
153     if (my_pe < npes) {
154
155         /* PDE vars. distributed to this process */
156         local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
157         my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;
158
159     } else {
160
161         /* Make last process inactive for forward phase */
162         local_N = 0;
163
164     }
165
166     /*-----
167      Allocate and load user data structure
168      -----*/
169     data = (UserData) malloc(sizeof *data);
170     if (check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
171     data->p[0] = ONE;
172     data->p[1] = RCONST(0.5);
173     dx = data->dx = XMAX/((realtype)(MX+1));
174     data->hdcoef = data->p[0]/(dx*dx);
175     data->hacoef = data->p[1]/(TWO*dx);

```

```

176 data->comm = comm;
177 data->npes = npes;
178 data->my_pe = my_pe;
179 data->nperpe = nperpe;
180 data->nrem = nrem;
181 data->local_N = local_N;
182
183 /*-----
184 Forward integration phase
185 -----*/
186
187 /* Set relative and absolute tolerances for forward phase */
188 reltol = ZERO;
189 abstol = ATOL;
190
191 /* Allocate and initialize forward variables */
192 u = N_VNew_Parallel(comm, local_N, NEQ);
193 if (check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
194 SetIC(u, dx, local_N, my_base);
195
196 /* Allocate CVODES memory for forward integration */
197 ccode_mem = CCodeCreate(CV_ADAMS, CV_FUNCTIONAL);
198 if (check_flag((void *)ccode_mem, "CCodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
199
200 flag = CCodeSetFdata(ccode_mem, data);
201 if (check_flag(&flag, "CCodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
202
203 flag = CCodeMalloc(ccode_mem, f, T0, u, CV_SS, reltol, &abstol);
204 if (check_flag(&flag, "CCodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
205
206 /* Allocate combined forward/backward memory */
207 cvadj_mem = CVadjMalloc(ccode_mem, STEPS, CV_HERMITE);
208 if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0, my_pe)) MPI_Abort(comm, 1);
209
210 /* Integrate to TOUT and collect check point information */
211 flag = CCodeF(cvadj_mem, TOUT, u, &t, CV_NORMAL, &ncheck);
212 if (check_flag(&flag, "CCodeF", 1, my_pe)) MPI_Abort(comm, 1);
213
214 /*-----
215 Compute and value of g(t_f)
216 -----*/
217 g_val = Compute_g(u, data);
218
219 /*-----
220 Backward integration phase
221 -----*/
222
223 if (my_pe == npes) {
224
225 /* Activate last process for integration of the quadrature equations */
226 local_N = NP;
227
228 } else {
229
230 /* Allocate work space */
231 data->z1 = (realtype *)malloc(local_N*sizeof(realtype));
232 if (check_flag((void *)data->z1, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
233 data->z2 = (realtype *)malloc(local_N*sizeof(realtype));
234 if (check_flag((void *)data->z2, "malloc", 2, my_pe)) MPI_Abort(comm, 1);

```

```

235
236     }
237
238     /* Allocate and initialize backward variables */
239     uB = N_VNew_Parallel(comm, local_N, NEQ+NP);
240     if (check_flag((void *)uB, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
241     SetICback(uB, my_base);
242
243     /* Allocate CVODES memory for the backward integration */
244     flag = CVodeCreateB(cvadj_mem, CV_ADAMS, CV_FUNCTIONAL);
245     if (check_flag(&flag, "CVodeCreateB", 1, my_pe)) MPI_Abort(comm, 1);
246     flag = CVodeSetFdataB(cvadj_mem, data);
247     if (check_flag(&flag, "CVodeSetFdataB", 1, my_pe)) MPI_Abort(comm, 1);
248     flag = CVodeMallocB(cvadj_mem, fB, TOUT, uB, CV_SS, reltol, &abstol);
249     if (check_flag(&flag, "CVodeMallocB", 1, my_pe)) MPI_Abort(comm, 1);
250
251     /* Integrate to T0 */
252     flag = CVodeB(cvadj_mem, T0, uB, &t, CV_NORMAL);
253     if (check_flag(&flag, "CVodeB", 1, my_pe)) MPI_Abort(comm, 1);
254
255     /* Print results (adjoint states and quadrature variables) */
256     PrintOutput(g_val, uB, data);
257
258
259     /* Free memory */
260     N_VDestroy_Parallel(u);
261     N_VDestroy_Parallel(uB);
262     CVodeFree(&cvode_mem);
263     CVadjFree(&cvadj_mem);
264     if (my_pe != npes) {
265         free(data->z1);
266         free(data->z2);
267     }
268     free(data);
269
270     MPI_Finalize();
271
272     return(0);
273 }
274
275 /*
276  *-----
277  * FUNCTIONS CALLED BY CVODES
278  *-----
279  */
280
281 /*
282  * f routine. Compute f(t,u) for forward phase.
283  */
284
285 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
286 {
287     realtype uLeft, uRight, ui, ult, urt;
288     realtype hordc, horac, hdiff, hadv;
289     realtype *udata, *dudata;
290     long int i, my_length;
291     int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
292     UserData data;
293     MPI_Status status;

```

```

294 MPI_Comm comm;
295
296 /* Extract MPI info. from data */
297 data = (UserData) f_data;
298 comm = data->comm;
299 npes = data->npes;
300 my_pe = data->my_pe;
301
302 /* If this process is inactive, return now */
303 if (my_pe == npes) return(0);
304
305 /* Extract problem constants from data */
306 hordc = data->hdcoef;
307 horac = data->hacoef;
308
309 /* Find related processes */
310 my_pe_m1 = my_pe - 1;
311 my_pe_p1 = my_pe + 1;
312 last_pe = npes - 1;
313
314 /* Obtain local arrays */
315 udata = NV_DATA_P(u);
316 dudata = NV_DATA_P(udot);
317 my_length = NV_LOCLENGTH_P(u);
318 my_last = my_length - 1;
319
320 /* Pass needed data to processes before and after current process. */
321 if (my_pe != 0)
322     MPI_Send(&udata[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
323 if (my_pe != last_pe)
324     MPI_Send(&udata[my_length-1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
325
326 /* Receive needed data from processes before and after current process. */
327 if (my_pe != 0)
328     MPI_Recv(&uLeft, 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
329 else uLeft = ZERO;
330 if (my_pe != last_pe)
331     MPI_Recv(&uRight, 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
332             &status);
333 else uRight = ZERO;
334
335 /* Loop over all grid points in current process. */
336 for (i=0; i<my_length; i++) {
337
338     /* Extract u at x_i and two neighboring points */
339     ui = udata[i];
340     ult = (i==0) ? uLeft : udata[i-1];
341     urt = (i==my_length-1) ? uRight : udata[i+1];
342
343     /* Set diffusion and advection terms and load into udot */
344     hdiff = hordc*(ult - TWO*ui + urt);
345     hadv = horac*(urt - ult);
346     dudata[i] = hdiff + hadv;
347 }
348
349 return(0);
350 }
351
352 /*

```

```

353  * fB routine. Compute right hand side of backward problem
354  */
355
356  static int fB(realtype t, N_Vector u,
357               N_Vector uB, N_Vector uBdot, void *f_dataB)
358  {
359      realtype *uBdata, *duBdata, *udata;
360      realtype uBLeft, uBRight, uBi, uBlT, uBrT;
361      realtype uLeft, uRight, ui, ulT, urT;
362      realtype dx, hordc, horac, hdiff, hadv;
363      realtype *z1, *z2, intgr1, intgr2;
364      long int i, my_length;
365      int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
366      UserData data;
367      realtype data_in[2], data_out[2];
368      MPI_Status status;
369      MPI_Comm comm;
370
371      /* Extract MPI info. from data */
372      data = (UserData) f_dataB;
373      comm = data->comm;
374      npes = data->npes;
375      my_pe = data->my_pe;
376
377      if (my_pe == npes) { /* This process performs the quadratures */
378
379          /* Obtain local arrays */
380          duBdata = NV_DATA_P(uBdot);
381          my_length = NV_LOCLENGTH_P(uB);
382
383          /* Loop over all other processes and load right hand side of quadrature eqs. */
384          duBdata[0] = ZERO;
385          duBdata[1] = ZERO;
386          for (i=0; i<npes; i++) {
387              MPI_Recv(&intgr1, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
388              duBdata[0] += intgr1;
389              MPI_Recv(&intgr2, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
390              duBdata[1] += intgr2;
391          }
392
393      } else { /* This process integrates part of the PDE */
394
395          /* Extract problem constants and work arrays from data */
396          dx = data->dx;
397          hordc = data->hdcoef;
398          horac = data->hacoef;
399          z1 = data->z1;
400          z2 = data->z2;
401
402          /* Obtain local arrays */
403          uBdata = NV_DATA_P(uB);
404          duBdata = NV_DATA_P(uBdot);
405          udata = NV_DATA_P(u);
406          my_length = NV_LOCLENGTH_P(uB);
407
408          /* Compute related parameters. */
409          my_pe_m1 = my_pe - 1;
410          my_pe_p1 = my_pe + 1;
411          last_pe = npes - 1;

```

```

412     my_last = my_length - 1;
413
414     /* Pass needed data to processes before and after current process. */
415     if (my_pe != 0) {
416         data_out[0] = udata[0];
417         data_out[1] = uBdata[0];
418
419         MPI_Send(data_out, 2, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
420     }
421     if (my_pe != last_pe) {
422         data_out[0] = udata[my_length-1];
423         data_out[1] = uBdata[my_length-1];
424
425         MPI_Send(data_out, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
426     }
427
428     /* Receive needed data from processes before and after current process. */
429     if (my_pe != 0) {
430         MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
431
432         uLeft = data_in[0];
433         uBLeft = data_in[1];
434     } else {
435         uLeft = ZERO;
436         uBLeft = ZERO;
437     }
438     if (my_pe != last_pe) {
439         MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm, &status);
440
441         uRight = data_in[0];
442         uBRight = data_in[1];
443     } else {
444         uRight = ZERO;
445         uBRight = ZERO;
446     }
447
448     /* Loop over all grid points in current process. */
449     for (i=0; i<my_length; i++) {
450
451         /* Extract uB at x_i and two neighboring points */
452         uBi = uBdata[i];
453         uBlt = (i==0) ? uBLeft : uBdata[i-1];
454         uBrt = (i==my_length-1) ? uBRight : uBdata[i+1];
455
456         /* Set diffusion and advection terms and load into udot */
457         hdiff = hordc*(uBlt - TWO*uBi + uBrt);
458         hadv = horac*(uBrt - uBlt);
459         duBdata[i] = - hdiff + hadv;
460
461         /* Extract u at x_i and two neighboring points */
462         ui = udata[i];
463         ult = (i==0) ? uLeft : udata[i-1];
464         urt = (i==my_length-1) ? uRight : udata[i+1];
465
466         /* Load integrands of the two space integrals */
467         z1[i] = uBdata[i]*(ult - TWO*ui + urt)/(dx*dx);
468         z2[i] = uBdata[i]*(urt - ult)/(TWO*dx);
469     }
470

```

```

471     /* Compute local integrals */
472     intgr1 = Xintgr(z1, my_length, dx);
473     intgr2 = Xintgr(z2, my_length, dx);
474
475     /* Send local integrals to 'quadrature' process */
476     MPI_Send(&intgr1, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
477     MPI_Send(&intgr2, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
478
479 }
480
481
482 return(0);
483 }
484
485 /*
486 *-----
487 * PRIVATE FUNCTIONS
488 *-----
489 */
490
491 /*
492 * Set initial conditions in u vector
493 */
494
495 static void SetIC(N_Vector u, realtype dx, long int my_length, long int my_base)
496 {
497     int i;
498     long int iglobal;
499     realtype x;
500     realtype *udata;
501
502     /* Set pointer to data array and get local length of u */
503     udata = NV_DATA_P(u);
504     my_length = NV_LOCLENGTH_P(u);
505
506     /* Load initial profile into u vector */
507     for (i=1; i<=my_length; i++) {
508         iglobal = my_base + i;
509         x = iglobal*dx;
510         udata[i-1] = x*(XMAX - x)*EXP(TWO*x);
511     }
512 }
513
514 /*
515 * Set final conditions in uB vector
516 */
517
518 static void SetICback(N_Vector uB, long int my_base)
519 {
520     int i;
521     realtype *uBdata;
522     long int my_length;
523
524     /* Set pointer to data array and get local length of uB */
525     uBdata = NV_DATA_P(uB);
526     my_length = NV_LOCLENGTH_P(uB);
527
528     /* Set adjoint states to 1.0 and quadrature variables to 0.0 */
529     if (my_base == -1) for (i=0; i<my_length; i++) uBdata[i] = ZERO;

```



```

530     else                for (i=0; i<my_length; i++) uBdata[i] = ONE;
531 }
532
533 /*
534  * Compute local value of the space integral  $\int_x z(x) dx$ 
535  */
536
537 static realtype Xintgr(realtype *z, long int l, realtype dx)
538 {
539     realtype my_intgr;
540     long int i;
541
542     my_intgr = RCONST(0.5)*(z[0] + z[l-1]);
543     for (i = 1; i < l-1; i++)
544         my_intgr += z[i];
545     my_intgr *= dx;
546
547     return(my_intgr);
548 }
549
550 /*
551  * Compute value of g(u)
552  */
553
554 static realtype Compute_g(N_Vector u, UserData data)
555 {
556     realtype intgr, my_intgr, dx, *udata;
557     long int my_length;
558     int npes, my_pe, i;
559     MPI_Status status;
560     MPI_Comm comm;
561
562     /* Extract MPI info. from data */
563     comm = data->comm;
564     npes = data->npes;
565     my_pe = data->my_pe;
566
567     dx = data->dx;
568
569     if (my_pe == npes) { /* Loop over all other processes and sum */
570         intgr = ZERO;
571         for (i=0; i<npes; i++) {
572             MPI_Recv(&my_intgr, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
573             intgr += my_intgr;
574         }
575         return(intgr);
576     } else { /* Compute local portion of the integral */
577         udata = NV_DATA_P(u);
578         my_length = NV_LOCLENGTH_P(u);
579         my_intgr = Xintgr(udata, my_length, dx);
580         MPI_Send(&my_intgr, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
581         return(my_intgr);
582     }
583 }
584
585 /*
586  * Print output after backward integration
587  */
588

```

```

589 static void PrintOutput(realtype g_val, N_Vector uB, UserData data)
590 {
591     MPI_Comm comm;
592     MPI_Status status;
593     int npes, my_pe;
594     long int i, Ni, indx, local_N, nperpe, nrem;
595     realtype *uBdata;
596     realtype *mu;
597
598     comm = data->comm;
599     npes = data->npes;
600     my_pe = data->my_pe;
601     local_N = data->local_N;
602     nperpe = data->nperpe;
603     nrem = data->nrem;
604
605     uBdata = NV_DATA_P(uB);
606
607     if (my_pe == npes) {
608
609         #if defined(SUNDIALS_EXTENDED_PRECISION)
610             printf("\ng(tf)=%8Le\n\n", g_val);
611             printf("dgdp(tf)\n%11s: %8Le\n%11s: %8Le\n\n", -uBdata[0], -uBdata[1]);
612         #elif defined(SUNDIALS_DOUBLE_PRECISION)
613             printf("\ng(tf)=%8le\n\n", g_val);
614             printf("dgdp(tf)\n%11s: %8le\n%11s: %8le\n\n", -uBdata[0], -uBdata[1]);
615         #else
616             printf("\ng(tf)=%8e\n\n", g_val);
617             printf("dgdp(tf)\n%11s: %8e\n%11s: %8e\n\n", -uBdata[0], -uBdata[1]);
618         #endif
619
620         mu = (realtype *)malloc(NEQ*sizeof(realtype));
621         if (check_flag((void *)mu, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
622
623         indx = 0;
624         for (i = 0; i < npes; i++) {
625             Ni = (i < nrem) ? nperpe+1 : nperpe;
626             MPI_Recv(&mu[indx], Ni, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
627             indx += Ni;
628         }
629
630         printf("mu(t0)\n");
631
632         #if defined(SUNDIALS_EXTENDED_PRECISION)
633             for (i=0; i<NEQ; i++)
634                 printf("%21d: %8Le\n", i+1, mu[i]);
635         #elif defined(SUNDIALS_DOUBLE_PRECISION)
636             for (i=0; i<NEQ; i++)
637                 printf("%21d: %8le\n", i+1, mu[i]);
638         #else
639             for (i=0; i<NEQ; i++)
640                 printf("%21d: %8e\n", i+1, mu[i]);
641         #endif
642
643         free(mu);
644     } else {
645
646         MPI_Send(uBdata, local_N, PVEC_REAL_MPI_TYPE, npes, 0, comm);
647

```

```

648     }
649 }
650
651 }
652
653 /*
654  * Check function return value.
655  *   opt == 0 means SUNDIALS function allocates memory so check if
656  *   returned NULL pointer
657  *   opt == 1 means SUNDIALS function returns a flag so check if
658  *   flag >= 0
659  *   opt == 2 means function allocates memory so check if returned
660  *   NULL pointer
661  */
662
663 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
664 {
665     int *errflag;
666
667     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
668     if (opt == 0 && flagvalue == NULL) {
669         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
670             id, funcname);
671         return(1); }
672
673     /* Check if flag < 0 */
674     else if (opt == 1) {
675         errflag = (int *) flagvalue;
676         if (*errflag < 0) {
677             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
678                 id, funcname, *errflag);
679             return(1); }}
680
681     /* Check if function returned NULL pointer - no memory allocated */
682     else if (opt == 2 && flagvalue == NULL) {
683         fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
684             id, funcname);
685         return(1); }
686
687     return(0);
688 }

```

F Listing of cvsadjkryx_p.c

```
1  /*
2  * -----
3  * $Revision: 1.1 $
4  * $Date: 2006/07/05 15:50:06 $
5  * -----
6  * Programmer(s): Lukas Jager and Radu Serban @ LLNL
7  * -----
8  * Parallel Krylov adjoint sensitivity example problem.
9  * -----
10 */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <math.h>
15 #include <limits.h>
16
17 #include <cvodes/cvodes.h>
18 #include <cvodes/cvodes_spgmr.h>
19 #include <cvodes/cvodes_bbdpre.h>
20 #include <nvector/nvector_parallel.h>
21 #include <sundials/sundials_types.h>
22 #include <sundials/sundials_math.h>
23
24 #include <mpi.h>
25
26 /*
27 * -----
28 * Constants
29 * -----
30 */
31
32 #ifdef USE3D
33 #define DIM 3
34 #else
35 #define DIM 2
36 #endif
37
38 /* Domain definition */
39
40 #define XMIN RCONST(0.0)
41 #define XMAX RCONST(20.0)
42 #define MX 20 /* no. of divisions in x dir. */
43 #define NPX 2 /* no. of procs. in x dir. */
44
45 #define YMIN RCONST(0.0)
46 #define YMAX RCONST(20.0)
47 #define MY 40 /* no. of divisions in y dir. */
48 #define NPY 2 /* no. of procs. in y dir. */
49
50 #ifdef USE3D
51 #define ZMIN RCONST(0.0)
52 #define ZMAX RCONST(20.0)
53 #define MZ 20 /* no. of divisions in z dir. */
54 #define NPZ 1 /* no. of procs. in z dir. */
55 #endif
56
57 /* Parameters for source Gaussians */
```

```

58
59 #define G1_AMPL      RCONST(1.0)
60 #define G1_SIGMA     RCONST(1.7)
61 #define G1_X         RCONST(4.0)
62 #define G1_Y         RCONST(8.0)
63 #ifdef USE3D
64 #define G1_Z         RCONST(8.0)
65 #endif
66
67 #define G2_AMPL      RCONST(0.8)
68 #define G2_SIGMA     RCONST(3.0)
69 #define G2_X         RCONST(16.0)
70 #define G2_Y         RCONST(12.0)
71 #ifdef USE3D
72 #define G2_Z         RCONST(12.0)
73 #endif
74
75 #define G_MIN        RCONST(1.0e-5)
76
77 /* Diffusion coeff., max. velocity, domain width in y dir. */
78
79 #define DIFF_COEF     RCONST(1.0)
80 #define V_MAX         RCONST(1.0)
81 #define L             (YMAX-YMIN)/RCONST(2.0)
82 #define V_COEFF       V_MAX/L/L
83
84 /* Initial and final times */
85
86 #define ti            RCONST(0.0)
87 #define tf            RCONST(10.0)
88
89 /* Integration tolerances */
90
91 #define RTOL          RCONST(1.0e-8) /* states */
92 #define ATOL          RCONST(1.0e-6)
93
94 #define RTOL_Q        RCONST(1.0e-8) /* forward quadrature */
95 #define ATOL_Q        RCONST(1.0e-6)
96
97 #define RTOL_B        RCONST(1.0e-8) /* adjoint variables */
98 #define ATOL_B        RCONST(1.0e-6)
99
100 #define RTOL_QB       RCONST(1.0e-8) /* backward quadratures */
101 #define ATOL_QB       RCONST(1.0e-6)
102
103 /* Steps between check points */
104
105 #define STEPS         200
106
107 #define ZERO          RCONST(0.0)
108 #define ONE           RCONST(1.0)
109 #define TWO           RCONST(2.0)
110
111 /*
112  *-----
113  *  Macros
114  *-----
115  */
116

```

```

117 #define FOR_DIM for(dim=0; dim<DIM; dim++)
118
119 /* IJth:      (i[0],i[1],i[2])-th vector component */
120 /* IJth_ext: (i[0],i[1],i[2])-th vector component in the extended array */
121
122 #ifdef USE3D
123 #define IJth(y,i)      ( y[(i[0])+(l_m[0]*((i[1])+(i[2])*l_m[1]))] )
124 #define IJth_ext(y,i) ( y[(i[0]+1)+((l_m[0]+2)*((i[1]+1)+(i[2]+1)*(l_m[1]+2)))] )
125 #else
126 #define IJth(y,i)      (y[i[0]+(i[1])*l_m[0]])
127 #define IJth_ext(y,i) (y[ (i[0]+1) + (i[1]+1) * (l_m[0]+2)])
128 #endif
129
130 /*
131  *-----
132  * Type definition: ProblemData
133  *-----
134  */
135
136 typedef struct {
137     /* Domain */
138     realtype xmin[DIM]; /* "left" boundaries */
139     realtype xmax[DIM]; /* "right" boundaries */
140     int m[DIM];          /* number of grid points */
141     realtype dx[DIM];    /* grid spacing */
142     realtype dOmega;     /* differential volume */
143
144     /* Parallel stuff */
145     MPI_Comm comm;       /* MPI communicator */
146     int myId;            /* process id */
147     int npes;            /* total number of processes */
148     int num_procs[DIM];  /* number of processes in each direction */
149     int nbr_left[DIM];   /* MPI ID of "left" neighbor */
150     int nbr_right[DIM];  /* MPI ID of "right" neighbor */
151     int m_start[DIM];    /* "left" index in the global domain */
152     int l_m[DIM];        /* number of local grid points */
153     realtype *y_ext;     /* extended data array */
154     realtype *buf_send;  /* Send buffer */
155     realtype *buf_recv;  /* Receive buffer */
156     int buf_size;        /* Buffer size */
157
158     /* Source */
159     N_Vector p;          /* Source parameters */
160
161 } *ProblemData;
162
163 /*
164  *-----
165  * Interface functions to CVODES
166  *-----
167  */
168
169 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
170 static int f_local(long int Nlocal, realtype t, N_Vector y,
171                   N_Vector ydot, void *f_data);
172
173 static int fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data);
174
175

```

```

176 static int fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
177             void *f_dataB);
178 static int fB_local(long int NlocalB, realtype t,
179                   N_Vector y, N_Vector yB, N_Vector yBdot,
180                   void *f_dataB);
181
182 static int fQB(realtype t, N_Vector y, N_Vector yB,
183               N_Vector qBdot, void *fQ_dataB);
184
185 /*
186  *-----
187  * Private functions
188  *-----
189  */
190
191 static void SetData(ProblemData d, MPI_Comm comm, int npes, int myId,
192                   long int *neq, long int *l_neq);
193 static void SetSource(ProblemData d);
194 static void f_comm( long int Nlocal, realtype t, N_Vector y, void *f_data);
195 static void Load_yext(realtype *src, ProblemData d);
196 static void PrintHeader();
197 static void PrintFinalStats(void *cnode_mem);
198 static void OutputGradient(int myId, N_Vector qB, ProblemData d);
199
200 /*
201  *-----
202  * Main program
203  *-----
204  */
205
206 int main(int argc, char *argv[])
207 {
208     ProblemData d;
209
210     MPI_Comm comm;
211     int npes, npes_needed;
212     int myId;
213
214     long int neq, l_neq;
215
216     void *cnode_mem;
217     N_Vector y, q;
218     realtype abstol, reltol, abstolQ, reltolQ;
219     void *bbdp_data;
220     int mudq, mldq, mukeep, mlkeep;
221
222     void *cvadj_mem;
223     void *cnode_memB;
224     N_Vector yB, qB;
225     realtype abstolB, reltolB, abstolQB, reltolQB;
226     int mudqB, mldqB, mukeepB, mlkeepB;
227
228     realtype tret, *qdata, G;
229
230     int ncheckpnt, flag;
231
232     booleantype output;
233
234     /* Initialize MPI and set Ids */

```

```

235 MPI_Init(&argc, &argv);
236 comm = MPI_COMM_WORLD;
237 MPI_Comm_rank(comm, &myId);
238
239 /* Check number of processes */
240 npes_needed = NPX * NPY;
241 #ifdef USE3D
242 npes_needed *= NPZ;
243 #endif
244 MPI_Comm_size(comm, &npes);
245 if (npes_needed != npes) {
246     if (myId == 0)
247         fprintf(stderr, "I need %d processes but I only got %d\n",
248                 npes_needed, npes);
249     MPI_Abort(comm, EXIT_FAILURE);
250 }
251
252 /* Test if matlab output is requested */
253 if (argc > 1) output = TRUE;
254 else output = FALSE;
255
256 /* Allocate and set problem data structure */
257 d = (ProblemData) malloc(sizeof *d);
258 SetData(d, comm, npes, myId, &neq, &l_neq);
259
260 if (myId == 0) PrintHeader();
261
262 /*-----
263 Forward integration phase
264 -----*/
265
266 /* Allocate space for y and set it with the I.C. */
267 y = N_VNew_Parallel(comm, l_neq, neq);
268 N_VConst(ZERO, y);
269
270 /* Allocate and initialize qB (local contributin to cost) */
271 q = N_VNew_Parallel(comm, 1, npes);
272 N_VConst(ZERO, q);
273
274 /* Create CVODES object, attach user data, and allocate space */
275 cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
276 flag = CVodeSetFdata(cvode_mem, d);
277 abstol = ATOL;
278 reltol = RTOL;
279 flag = CVodeMalloc(cvode_mem, f, ti, y, CV_SS, reltol, &abstol);
280
281 /* Attach preconditioner and linear solver modules */
282 mudq = mldq = d->l_m[0]+1;
283 mukeep = mlkeep = 2;
284 bbdp_data = (void *) CVBBDPrecAlloc(cvode_mem, l_neq, mudq, mldq,
285                                     mukeep, mlkeep, ZERO,
286                                     f_local, NULL);
287 flag = CVBBDSPgmr(cvode_mem, PREC_LEFT, 0, bbdp_data);
288
289 /* Initialize quadrature calculations */
290 abstolQ = ATOL_Q;
291 reltolQ = RTOL_Q;
292 flag = CVodeQuadMalloc(cvode_mem, fQ, q);
293 flag = CVodeSetQuadFdata(cvode_mem, d);

```



```

294     flag = CVodeSetQuadErrCon(cvode_mem, TRUE, CV_SS, reltolQ, &abstolQ);
295
296     /* Allocate space for the adjoint calculation */
297     cvadj_mem = CVadjMalloc(cvode_mem, STEPS, CV_HERMITE);
298
299     /* Integrate forward in time while storing check points */
300     if (myId == 0) printf("Begin forward integration...\n");
301     flag = CVodeF(cvadj_mem, tf, y, &tret, CV_NORMAL, &ncheckpnt);
302     if (myId == 0) printf("done.\n");
303
304     /* Extract quadratures */
305     flag = CVodeGetQuad(cvode_mem, tf, q);
306     qdata = NV_DATA_P(q);
307     MPI_Allreduce(&qdata[0], &G, 1, PVEC_REAL_MPI_TYPE, MPI_SUM, comm);
308     #if defined(SUNDIALS_EXTENDED_PRECISION)
309     if (myId == 0) printf("G=%Le\n", G);
310     #elif defined(SUNDIALS_DOUBLE_PRECISION)
311     if (myId == 0) printf("G=%le\n", G);
312     #else
313     if (myId == 0) printf("G=%e\n", G);
314     #endif
315
316     /* Print statistics for forward run */
317     if (myId == 0) PrintFinalStats(cvode_mem);
318
319     /*-----
320     Backward integration phase
321     -----*/
322
323     /* Allocate and initialize yB */
324     yB = N_VNew_Parallel(comm, l_neq, neq);
325     N_VConst(ZERO, yB);
326
327     /* Allocate and initialize qB (gradient) */
328     qB = N_VNew_Parallel(comm, l_neq, neq);
329     N_VConst(ZERO, qB);
330
331     /* Create and allocate backward CVODE memory */
332     flag = CVodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
333     flag = CVodeSetFdataB(cvadj_mem, d);
334     abstolB = ATOL_B;
335     reltolB = RTOL_B;
336     flag = CVodeMallocB(cvadj_mem, fB, tf, yB, CV_SS, reltolB, &abstolB);
337
338     /* Attach preconditioner and linear solver modules */
339     mudqB = mldqB = d->l_m[0]+1;
340     mukeepB = mlkeepB = 2;
341     flag = CVBBDPrecAllocB(cvadj_mem, l_neq, mudqB, mldqB,
342                           mukeepB, mlkeepB, ZERO, fB_local, NULL);
343     flag = CVBBDSPgmrB(cvadj_mem, PREC_LEFT, 0);
344
345     /* Initialize quadrature calculations */
346     abstolQB = ATOL_QB;
347     reltolQB = RTOL_QB;
348     flag = CVodeQuadMallocB(cvadj_mem, fQB, qB);
349     flag = CVodeSetQuadFdataB(cvadj_mem, d);
350     flag = CVodeSetQuadErrConB(cvadj_mem, TRUE, CV_SS, reltolQB, &abstolQB);
351
352     /* Integrate backwards */

```

```

353     if (myId == 0) printf("Begin backward integration...\n");
354     flag = CVodeB(cvadj_mem, ti, yB, &tret, CV_NORMAL);
355     if (myId == 0) printf("done.\n");
356
357     /* Print statistics for backward run */
358     if (myId == 0) {
359         ccode_memB = CVadjGetCCodeBmem(cvadj_mem);
360         PrintFinalStats(ccode_memB);
361     }
362
363     /* Extract quadratures */
364     flag = CVodeGetQuadB(cvadj_mem, qB);
365
366     /* Process 0 collects the gradient components and prints them */
367     if (output) {
368         OutputGradient(myId, qB, d);
369         if (myId == 0) printf("Wrote matlab file 'grad.m'.\n");
370     }
371
372     /* Free memory */
373     N_VDestroy_Parallel(y);
374     N_VDestroy_Parallel(q);
375     N_VDestroy_Parallel(qB);
376     N_VDestroy_Parallel(yB);
377
378     CVBBDPrecFree(&bbdp_data);
379     CVodeFree(&ccode_mem);
380
381     CVBBDPrecFreeB(cvadj_mem);
382     CVadjFree(&cvadj_mem);
383
384     MPI_Finalize();
385
386     return(0);
387 }
388
389 /*
390 *-----
391 * SetData:
392 * Allocate space for the ProblemData structure.
393 * Set fields in the ProblemData structure.
394 * Return local and global problem dimensions.
395 *
396 * SetSource:
397 * Instantiates the source parameters for a combination of two
398 * Gaussian sources.
399 *-----
400 */
401
402 static void SetData(ProblemData d, MPI_Comm comm, int npes, int myId,
403                    long int *neq, long int *l_neq)
404 {
405     int n[DIM], nd[DIM];
406     int dim, size;
407
408     /* Set MPI communicator, id, and total number of processes */
409
410     d->comm = comm;
411     d->myId = myId;

```

```

412     d->npes = npes;
413
414     /* Set domain boundaries */
415
416     d->xmin[0] = XMIN;
417     d->xmax[0] = XMAX;
418     d->m[0]    = MX;
419
420     d->xmin[1] = YMIN;
421     d->xmax[1] = YMAX;
422     d->m[1]    = MY;
423
424     #ifdef USE3D
425     d->xmin[2] = ZMIN;
426     d->xmax[2] = ZMAX;
427     d->m[2]    = MZ;
428     #endif
429
430     /* Calculate grid spacing and differential volume */
431
432     d->dOmega = ONE;
433     FOR_DIM {
434         d->dx[dim] = (d->xmax[dim] - d->xmin[dim]) / d->m[dim];
435         d->m[dim] += 1;
436         d->dOmega *= d->dx[dim];
437     }
438
439     /* Set partitioning */
440
441     d->num_procs[0] = NPX;
442     n[0] = NPX;
443     nd[0] = d->m[0] / NPX;
444
445     d->num_procs[1] = NPY;
446     n[1] = NPY;
447     nd[1] = d->m[1] / NPY;
448
449     #ifdef USE3D
450     d->num_procs[2] = NPZ;
451     n[2] = NPZ;
452     nd[2] = d->m[2] / NPZ;
453     #endif
454
455     /* Compute the neighbors */
456
457     d->nbr_left[0]  = (myId%n[0]) == 0           ? myId : myId-1;
458     d->nbr_right[0] = (myId%n[0]) == n[0]-1      ? myId : myId+1;
459
460     d->nbr_left[1]  = (myId/n[0])%n[1] == 0       ? myId : myId-n[0];
461     d->nbr_right[1] = (myId/n[0])%n[1] == n[1]-1  ? myId : myId+n[0];
462
463     #ifdef USE3D
464     d->nbr_left[2]  = (myId/n[0]/n[1])%n[2] == 0   ? myId : myId-n[0]*n[1];
465     d->nbr_right[2] = (myId/n[0]/n[1])%n[2] == n[2]-1 ? myId : myId+n[0]*n[1];
466     #endif
467
468     /* Compute the local subdomains
469     m_start: left border in global index space
470     l_m:      length of the subdomain */

```

```

471
472     d->m_start[0] = (myId%n[0])*nd[0];
473     d->l_m[0]      = d->nbr_right[0] == myId ? d->m[0] - d->m_start[0] : nd[0];
474
475     d->m_start[1] = ((myId/n[0])%n[1])*nd[1];
476     d->l_m[1]      = d->nbr_right[1] == myId ? d->m[1] - d->m_start[1] : nd[1];
477
478     #ifdef USE3D
479     d->m_start[2] = (myId/n[0]/n[1])*nd[2];
480     d->l_m[2]      = d->nbr_right[2] == myId ? d->m[2] - d->m_start[2] : nd[2];
481     #endif
482
483     /* Allocate memory for the y_ext array
484        (local solution + data from neighbors) */
485
486     size = 1;
487     FOR_DIM size *= d->l_m[dim]+2;
488     d->y_ext = (realtype *) malloc( size*sizeof(realtype));
489
490     /* Initialize Buffer field.
491        Size of buffer is checked when needed */
492
493     d->buf_send = NULL;
494     d->buf_recv = NULL;
495     d->buf_size = 0;
496
497     /* Allocate space for the source parameters */
498
499     *neq = 1; *l_neq = 1;
500     FOR_DIM { *neq *= d->m[dim]; *l_neq *= d->l_m[dim]; }
501     d->p = N_VNew_Parallel(comm, *l_neq, *neq);
502
503     /* Initialize the parameters for a source with Gaussian profile */
504
505     SetSource(d);
506
507 }
508
509 static void SetSource(ProblemData d)
510 {
511     int *l_m, *m_start;
512     realtype *xmin, *xmax, *dx;
513     realtype x[DIM], g, *pdata;
514     int i[DIM];
515
516     l_m = d->l_m;
517     m_start = d->m_start;
518     xmin = d->xmin;
519     xmax = d->xmax;
520     dx = d->dx;
521
522
523     pdata = NV_DATA_P(d->p);
524
525     for(i[0]=0; i[0]<l_m[0]; i[0]++) {
526         x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
527         for(i[1]=0; i[1]<l_m[1]; i[1]++) {
528             x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
529     #ifdef USE3D

```

```

530     for(i[2]=0; i[2]<l_m[2]; i[2]++) {
531         x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];
532
533         g = G1_AMPL
534             * EXP( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
535             * EXP( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) )
536             * EXP( -SQR(G1_Z-x[2])/SQR(G1_SIGMA) );
537
538         g += G2_AMPL
539             * EXP( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )
540             * EXP( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) )
541             * EXP( -SQR(G2_Z-x[2])/SQR(G2_SIGMA) );
542
543         if( g < G_MIN ) g = ZERO;
544
545         IJth(pdata, i) = g;
546     }
547 #else
548     g = G1_AMPL
549         * EXP( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
550         * EXP( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) );
551
552     g += G2_AMPL
553         * EXP( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )
554         * EXP( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) );
555
556     if( g < G_MIN ) g = ZERO;
557
558     IJth(pdata, i) = g;
559 #endif
560 }
561 }
562 }
563
564 /*
565  *-----
566  * f_comm:
567  * Function for inter-process communication
568  * Used both for the forward and backward phase.
569  *-----
570  */
571
572 static void f_comm(long int N_local, realtype t, N_Vector y, void *f_data)
573 {
574     int id, n[DIM], proc_cond[DIM], nbr[DIM][2];
575     ProblemData d;
576     realtype *yextdata, *ydata;
577     int l_m[DIM], dim;
578     int c, i[DIM], l[DIM-1];
579     realtype *buf_send, *buf_recv;
580     MPI_Status stat;
581     MPI_Comm comm;
582     int dir, size = 1, small = INT_MAX;
583
584     d = (ProblemData) f_data;
585     comm = d->comm;
586     id = d->myId;
587
588     /* extract data from domain*/

```

```

589     FOR_DIM {
590         n[dim] = d->num_procs[dim];
591         l_m[dim] = d->l_m[dim];
592     }
593     yextdata = d->y_ext;
594     ydata     = NV_DATA_P(y);
595
596     /* Calculate required buffer size */
597     FOR_DIM {
598         size *= l_m[dim];
599         if( l_m[dim] < small) small = l_m[dim];
600     }
601     size /= small;
602
603     /* Adjust buffer size if necessary */
604     if( d->buf_size < size ) {
605         d->buf_send = (realtype*) realloc( d->buf_send, size * sizeof(realtype));
606         d->buf_recv = (realtype*) realloc( d->buf_recv, size * sizeof(realtype));
607         d->buf_size = size;
608     }
609
610     buf_send = d->buf_send;
611     buf_recv = d->buf_recv;
612
613     /* Compute the communication pattern; who sends first? */
614     /* if proc_cond==1 , process sends first in this dimension */
615     proc_cond[0] = (id%n[0])%2;
616     proc_cond[1] = ((id/n[0])%n[1])%2;
617     #ifdef USE3D
618     proc_cond[2] = (id/n[0]/n[1])%2;
619     #endif
620
621     /* Compute the actual communication pattern */
622     /* nbr[dim][0] is first proc to communicate with in dimension dim */
623     /* nbr[dim][1] the second one */
624     FOR_DIM {
625         nbr[dim][proc_cond[dim]] = d->nbr_left[dim];
626         nbr[dim][!proc_cond[dim]] = d->nbr_right[dim];
627     }
628
629     /* Communication: loop over dimension and direction (left/right) */
630     FOR_DIM {
631
632         for (dir=0; dir<=1; dir++) {
633
634             /* If subdomain at boundary, no communication in this direction */
635
636             if (id != nbr[dim][dir]) {
637                 c=0;
638                 /* Compute the index of the boundary (right or left) */
639                 i[dim] = (dir ^ proc_cond[dim]) ? (l_m[dim]-1) : 0;
640                 /* Loop over all other dimensions and copy data into buf_send */
641                 l[0]=(dim+1)%DIM;
642                 #ifdef USE3D
643                 l[1]=(dim+2)%DIM;
644                 for(i[l[1]]=0; i[l[1]]<l_m[l[1]]; i[l[1]]++)
645                 #endif
646                 for(i[l[0]]=0; i[l[0]]<l_m[l[0]]; i[l[0]]++)
647                     buf_send[c++] = IJth(ydata, i);

```

```

648
649     if ( proc_cond[dim] ) {
650         /* Send buf_send and receive into buf_recv */
651         MPI_Send(buf_send, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm);
652         MPI_Recv(buf_recv, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm, &stat);
653     } else {
654         /* Receive into buf_recv and send buf_send*/
655         MPI_Recv(buf_recv, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm, &stat);
656         MPI_Send(buf_send, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm);
657     }
658
659     c=0;
660
661     /* Compute the index of the boundary (right or left) in yextdata */
662     i[dim] = (dir ^ proc_cond[dim]) ? l_m[dim] : -1;
663
664     /* Loop over all other dimensions and copy data into yextdata */
665 #ifdef USE3D
666     for(i[l[1]]=0; i[l[1]]<l_m[l[1]]; i[l[1]]++)
667 #endif
668         for(i[l[0]]=0; i[l[0]]<l_m[l[0]]; i[l[0]]++)
669             IJth_ext(yextdata, i) = buf_recv[c++];
670     }
671 } /* end loop over direction */
672 } /* end loop over dimension */
673 }
674
675 /*
676 -----
677 * f and f_local:
678 * Forward phase ODE right-hand side
679 -----
680 */
681
682 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
683 {
684     ProblemData d;
685     int l_neq=1;
686     int dim;
687
688     d = (ProblemData) f_data;
689     FOR_DIM l_neq *= d->l_m[dim];
690
691     /* Do all inter-processor communication */
692     f_comm(l_neq, t, y, f_data);
693
694     /* Compute right-hand side locally */
695     f_local(l_neq, t, y, ydot, f_data);
696
697     return(0);
698 }
699
700 static int f_local(long int Nlocal, realtype t, N_Vector y,
701                  N_Vector ydot, void *f_data)
702 {
703     realtype *Ydata, *dydata, *pdata;
704     realtype dx[DIM], c, v[DIM], cl[DIM], cr[DIM];
705     realtype adv[DIM], diff[DIM];
706     realtype xmin[DIM], xmax[DIM], x[DIM], x1;

```

```

707     int i[DIM], l_m[DIM], m_start[DIM], nbr_left[DIM], nbr_right[DIM], id;
708     ProblemData d;
709     int dim;
710
711     d = (ProblemData) f_data;
712
713     /* Extract stuff from data structure */
714     id = d->myId;
715     FOR_DIM {
716         xmin[dim]      = d->xmin[dim];
717         xmax[dim]      = d->xmax[dim];
718         l_m[dim]       = d->l_m[dim];
719         m_start[dim]   = d->m_start[dim];
720         dx[dim]        = d->dx[dim];
721         nbr_left[dim]  = d->nbr_left[dim];
722         nbr_right[dim] = d->nbr_right[dim];
723     }
724
725     /* Get pointers to vector data */
726     dydata = NV_DATA_P(ydot);
727     pdata  = NV_DATA_P(d->p);
728
729     /* Copy local segment of y to y_ext */
730     Load_yext(NV_DATA_P(y), d);
731     Ydata = d->y_ext;
732
733     /* Velocity components in x1 and x2 directions (Poiseuille profile) */
734     v[1] = ZERO;
735     #ifdef USE3D
736     v[2] = ZERO;
737     #endif
738
739     /* Local domain is [xmin+(m_start+1)*dx, xmin+(m_start+1+l_m-1)*dx] */
740     #ifdef USE3D
741     for(i[2]=0; i[2]<l_m[2]; i[2]++) {
742
743         x[2] = xmin[2] + (m_start[2]+i[2])*dx[2];
744     #endif
745     for(i[1]=0; i[1]<l_m[1]; i[1]++) {
746
747         x[1] = xmin[1] + (m_start[1]+i[1])*dx[1];
748
749         /* Velocity component in x0 direction (Poiseuille profile) */
750         x1 = x[1] - xmin[1] - L;
751         v[0] = V_COEFF * (L + x1) * (L - x1);
752
753         for(i[0]=0; i[0]<l_m[0]; i[0]++) {
754
755             x[0] = xmin[0] + (m_start[0]+i[0])*dx[0];
756
757             c = IJth_ext(Ydata, i);
758
759             /* Source term*/
760             IJth(dydata, i) = IJth(pdata, i);
761
762             FOR_DIM {
763                 i[dim]+=1;
764                 cr[dim] = IJth_ext(Ydata, i);
765                 i[dim]-=2;

```



```

766         cl[dim] = IJth_ext(Ydata, i);
767         i[dim] += 1;
768
769         /* Boundary conditions for the state variables */
770         if( i[dim] == l_m[dim] - 1 && nbr_right[dim] == id )
771             cr[dim] = cl[dim];
772         else if( i[dim] == 0 && nbr_left[dim] == id )
773             cl[dim] = cr[dim];
774
775         adv[dim] = v[dim] * (cr[dim] - cl[dim]) / (TWO * dx[dim]);
776         diff[dim] = DIFF_COEF * (cr[dim] - TWO * c + cl[dim]) / SQR(dx[dim]);
777
778         IJth(dydata, i) += (diff[dim] - adv[dim]);
779     }
780 }
781 }
782 #ifdef USE3D
783 }
784 #endif
785
786     return(0);
787 }
788
789 /*
790 *-----
791 * fQ:
792 * Right-hand side of quadrature equations on forward integration.
793 * The only quadrature on this phase computes the local contribution
794 * to the function G.
795 *-----
796 */
797
798 static int fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
799 {
800     ProblemData d;
801     realtype *dqdata;
802
803     d = (ProblemData) fQ_data;
804
805     dqdata = NV_DATA_P(qdot);
806
807     dqdata[0] = N_VDotProd_Parallel(y, y);
808     dqdata[0] *= RCONST(0.5) * (d->d0omega);
809
810     return(0);
811 }
812
813 /*
814 *-----
815 * fB and fB_local:
816 * Backward phase ODE right-hand side (the discretized adjoint PDE)
817 *-----
818 */
819
820 static int fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
821              void *f_dataB)
822 {
823     ProblemData d;
824     int l_neq = 1;

```

```

825     int dim;
826
827     d = (ProblemData) f_dataB;
828     FOR_DIM l_neq *= d->l_m[dim];
829
830     /* Do all inter-processor communication */
831     f_comm(l_neq, t, yB, f_dataB);
832
833     /* Compute right-hand side locally */
834     fB_local(l_neq, t, y, yB, yBdot, f_dataB);
835
836     return(0);
837 }
838
839 static int fB_local(long int NlocalB, realtype t,
840                   N_Vector y, N_Vector yB, N_Vector dyB,
841                   void *f_dataB)
842 {
843     realtype *YBdata, *dyBdata, *ydata;
844     realtype dx[DIM], c, v[DIM], cl[DIM], cr[DIM];
845     realtype adv[DIM], diff[DIM];
846     realtype xmin[DIM], xmax[DIM], x[DIM], x1;
847     int i[DIM], l_m[DIM], m_start[DIM], nbr_left[DIM], nbr_right[DIM], id;
848     ProblemData d;
849     int dim;
850
851     d = (ProblemData) f_dataB;
852
853     /* Extract stuff from data structure */
854     id = d->myId;
855     FOR_DIM {
856         xmin[dim] = d->xmin[dim];
857         xmax[dim] = d->xmax[dim];
858         l_m[dim] = d->l_m[dim];
859         m_start[dim] = d->m_start[dim];
860         dx[dim] = d->dx[dim];
861         nbr_left[dim] = d->nbr_left[dim];
862         nbr_right[dim] = d->nbr_right[dim];
863     }
864
865     dyBdata = NV_DATA_P(dyB);
866     ydata = NV_DATA_P(y);
867
868     /* Copy local segment of yB to y_ext */
869     Load_yext(NV_DATA_P(yB), d);
870     YBdata = d->y_ext;
871
872     /* Velocity components in x1 and x2 directions (Poiseuille profile) */
873     v[1] = ZERO;
874     #ifdef USE3D
875     v[2] = ZERO;
876     #endif
877
878     /* local domain is [xmin+(m_start)*dx, xmin+(m_start+l_m-1)*dx] */
879     #ifdef USE3D
880     for(i[2]=0; i[2]<l_m[2]; i[2]++) {
881
882         x[2] = xmin[2] + (m_start[2]+i[2])*dx[2];
883     #endif

```

```

884
885     for(i[1]=0; i[1]<l_m[1]; i[1]++) {
886
887         x[1] = xmin[1] + (m_start[1]+i[1])*dx[1];
888
889         /* Velocity component in x0 direction (Poiseuille profile) */
890         x1 = x[1] - xmin[1] - L;
891         v[0] = V_COEFF * (L + x1) * (L - x1);
892
893         for(i[0]=0; i[0]<l_m[0]; i[0]++) {
894
895             x[0] = xmin[0] + (m_start[0]+i[0])*dx[0];
896
897             c = IJth_ext(YBdata, i);
898
899             /* Source term for adjoint PDE */
900             IJth(dyBdata, i) = -IJth(ydata, i);
901
902             FOR_DIM {
903
904                 i[dim]+=1;
905                 cr[dim] = IJth_ext(YBdata, i);
906                 i[dim]-=2;
907                 cl[dim] = IJth_ext(YBdata, i);
908                 i[dim]+=1;
909
910                 /* Boundary conditions for the adjoint variables */
911                 if( i[dim]==l_m[dim]-1 && nbr_right[dim]==id)
912                     cr[dim] = cl[dim]-(TWO*dx[dim]*v[dim]/DIFF_COEF)*c;
913                 else if( i[dim]==0 && nbr_left[dim]==id )
914                     cl[dim] = cr[dim]+(TWO*dx[dim]*v[dim]/DIFF_COEF)*c;
915
916                 adv[dim] = v[dim] * (cr[dim]-cl[dim]) / (TWO*dx[dim]);
917                 diff[dim] = DIFF_COEF * (cr[dim]-TWO*c+cl[dim]) / SQR(dx[dim]);
918
919                 IJth(dyBdata, i) -= (diff[dim] + adv[dim]);
920             }
921         }
922     }
923 #ifdef USE3D
924 }
925 #endif
926
927     return(0);
928 }
929
930 /*
931  *-----
932  * fQB:
933  * Right-hand side of quadrature equations on backward integration
934  * The i-th component of the gradient is nothing but int_t yB_i dt
935  *-----
936  */
937
938 static int fQB(realtype t, N_Vector y, N_Vector yB, N_Vector qBdot,
939               void *fQ_dataB)
940 {
941     ProblemData d;
942

```

```

943     d = (ProblemData) fQ_dataB;
944
945     N_VScale_Parallel(-(d->dOmega), yB, qBdot);
946
947     return(0);
948 }
949
950 /*
951 *-----
952 * Load_yext:
953 * copies data from src (y or yB) into y_ext, which already contains
954 * data from neighboring processes.
955 *-----
956 */
957
958 static void Load_yext(realtype *src, ProblemData d)
959 {
960     int i[DIM], l_m[DIM], dim;
961
962     FOR_DIM l_m[dim] = d->l_m[dim];
963
964     /* copy local segment */
965 #ifdef USE3D
966     for (i[2]=0; i[2]<l_m[2]; i[2]++)
967 #endif
968     for(i[1]=0; i[1]<l_m[1]; i[1]++)
969     for(i[0]=0; i[0]<l_m[0]; i[0]++)
970         IJth_ext(d->y_ext, i) = IJth(src, i);
971 }
972
973 /*
974 *-----
975 * PrintHeader:
976 * Print first lines of output (problem description)
977 *-----
978 */
979
980 static void PrintHeader()
981 {
982     printf("\nParallel_Krylov_adjoint_sensitivity_analysis_example\n");
983     printf("%1dD_Advection_diffusion_PDE_with_homogeneous_Neumann_B.C.\n", DIM);
984     printf("Computes_gradient_of_G_u_int_t_Omega_(u_c_i^2_u)_dt_dOmega\n");
985     printf("with_respect_to_the_source_values_at_each_grid_point.\n\n");
986
987     printf("Domain:\n");
988
989 #if defined(SUNDIALS_EXTENDED_PRECISION)
990     printf("uuuLf_u<u_xu<uLf_uuuuux_u=%d_uu_npe_x_u=%d_u\n", XMIN, XMAX, MX, NPX);
991     printf("uuuLf_u<u_yu<uLf_uuuuuy_u=%d_uu_npe_y_u=%d_u\n", YMIN, YMAX, MY, NPY);
992 #else
993     printf("uuuLf_u<u_xu<uLf_uuuuux_u=%d_uu_npe_x_u=%d_u\n", XMIN, XMAX, MX, NPX);
994     printf("uuuLf_u<u_yu<uLf_uuuuuy_u=%d_uu_npe_y_u=%d_u\n", YMIN, YMAX, MY, NPY);
995 #endif
996
997 #ifdef USE3D
998 #if defined(SUNDIALS_EXTENDED_PRECISION)
999     printf("uuuLf_u<u_zu<uLf_uuuuuz_u=%d_uu_npe_z_u=%d_u\n", ZMIN, ZMAX, MZ, NPZ);
1000 #else
1001     printf("uuuLf_u<u_zu<uLf_uuuuuz_u=%d_uu_npe_z_u=%d_u\n", ZMIN, ZMAX, MZ, NPZ);

```

```

1002 #endif
1003 #endif
1004
1005     printf("\n");
1006 }
1007
1008 /*
1009 *-----
1010 * PrintFinalStats:
1011 * Print final statistics contained in cvode_mem
1012 *-----
1013 */
1014
1015 static void PrintFinalStats(void *cvode_mem)
1016 {
1017     long int lenrw, leniw ;
1018     long int lenrwSPGMR, leniwSPGMR;
1019     long int nst, nfe, nsetups, nni, ncfn, netf;
1020     long int nli, npe, nps, ncfl, nfeSPGMR;
1021     int flag;
1022
1023     flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
1024     flag = CVodeGetNumSteps(cvode_mem, &nst);
1025     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
1026     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
1027     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
1028     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
1029     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
1030
1031     flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwSPGMR, &leniwSPGMR);
1032     flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
1033     flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
1034     flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
1035     flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
1036     flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeSPGMR);
1037
1038     printf("\nFinal Statistics...\n\n");
1039     printf("lenrw=%6ld leniw=%6ld\n", lenrw, leniw);
1040     printf("llrw=%6ld lliw=%6ld\n", lenrwSPGMR, leniwSPGMR);
1041     printf("nst=%6ld\n", nst);
1042     printf("nfe=%6ld nfeSPGMR=%6ld\n", nfe, nfeSPGMR);
1043     printf("nni=%6ld nli=%6ld\n", nni, nli);
1044     printf("nsetups=%6ld netf=%6ld\n", nsetups, netf);
1045     printf("npe=%6ld nps=%6ld\n", npe, nps);
1046     printf("ncfn=%6ld ncfl=%6ld\n", ncfn, ncfl);
1047 }
1048
1049 /*
1050 *-----
1051 * OutputGradient:
1052 * Generate matlab m files for visualization
1053 * One file gradXXXX.m from each process + a driver grad.m
1054 *-----
1055 */
1056
1057 static void OutputGradient(int myId, N_Vector qB, ProblemData d)
1058 {
1059     FILE *fid;
1060     char filename[20];

```

```

1061     int *l_m, *m_start, i[DIM],ip;
1062     realtype *xmin, *xmax, *dx;
1063     realtype x[DIM], *pdata, p, *qBdata, g;
1064
1065     sprintf(filename,"grad%03d.m",myId);
1066     fid = fopen(filename,"w");
1067
1068     l_m = d->l_m;
1069     m_start = d->m_start;
1070     xmin = d->xmin;
1071     xmax = d->xmax;
1072     dx = d->dx;
1073
1074     qBdata = NV_DATA_P(qB);
1075     pdata = NV_DATA_P(d->p);
1076
1077     /* Write matlab files with solutions from each process */
1078
1079     for(i[0]=0; i[0]<l_m[0]; i[0]++) {
1080         x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
1081         for(i[1]=0; i[1]<l_m[1]; i[1]++) {
1082             x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
1083 #ifdef USE3D
1084             for(i[2]=0; i[2]<l_m[2]; i[2]++) {
1085                 x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];
1086                 g = IJth(qBdata, i);
1087                 p = IJth(pdata, i);
1088 #if defined(SUNDIALS_EXTENDED_PRECISION)
1089                 fprintf(fid,"x%d(%d,1)=%Le;\n", myId, i[0]+1, x[0]);
1090                 fprintf(fid,"y%d(%d,1)=%Le;\n", myId, i[1]+1, x[1]);
1091                 fprintf(fid,"z%d(%d,1)=%Le;\n", myId, i[2]+1, x[2]);
1092                 fprintf(fid,"p%d(%d,%d,%d)=%Le;\n", myId, i[1]+1, i[0]+1, i[2]+1, p);
1093                 fprintf(fid,"g%d(%d,%d,%d)=%Le;\n", myId, i[1]+1, i[0]+1, i[2]+1, g);
1094 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1095                 fprintf(fid,"x%d(%d,1)=%le;\n", myId, i[0]+1, x[0]);
1096                 fprintf(fid,"y%d(%d,1)=%le;\n", myId, i[1]+1, x[1]);
1097                 fprintf(fid,"z%d(%d,1)=%le;\n", myId, i[2]+1, x[2]);
1098                 fprintf(fid,"p%d(%d,%d,%d)=%le;\n", myId, i[1]+1, i[0]+1, i[2]+1, p);
1099                 fprintf(fid,"g%d(%d,%d,%d)=%le;\n", myId, i[1]+1, i[0]+1, i[2]+1, g);
1100 #else
1101                 fprintf(fid,"x%d(%d,1)=%e;\n", myId, i[0]+1, x[0]);
1102                 fprintf(fid,"y%d(%d,1)=%e;\n", myId, i[1]+1, x[1]);
1103                 fprintf(fid,"z%d(%d,1)=%e;\n", myId, i[2]+1, x[2]);
1104                 fprintf(fid,"p%d(%d,%d,%d)=%e;\n", myId, i[1]+1, i[0]+1, i[2]+1, p);
1105                 fprintf(fid,"g%d(%d,%d,%d)=%e;\n", myId, i[1]+1, i[0]+1, i[2]+1, g);
1106 #endif
1107             }
1108 #else
1109             g = IJth(qBdata, i);
1110             p = IJth(pdata, i);
1111 #if defined(SUNDIALS_EXTENDED_PRECISION)
1112             fprintf(fid,"x%d(%d,1)=%Le;\n", myId, i[0]+1, x[0]);
1113             fprintf(fid,"y%d(%d,1)=%Le;\n", myId, i[1]+1, x[1]);
1114             fprintf(fid,"p%d(%d,%d)=%Le;\n", myId, i[1]+1, i[0]+1, p);
1115             fprintf(fid,"g%d(%d,%d)=%Le;\n", myId, i[1]+1, i[0]+1, g);
1116 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1117             fprintf(fid,"x%d(%d,1)=%le;\n", myId, i[0]+1, x[0]);
1118             fprintf(fid,"y%d(%d,1)=%le;\n", myId, i[1]+1, x[1]);
1119             fprintf(fid,"p%d(%d,%d)=%le;\n", myId, i[1]+1, i[0]+1, p);

```

```

1120         fprintf(fid,"g%d(%d,%d)=%le;\n", myId, i[1]+1, i[0]+1, g);
1121     #else
1122         fprintf(fid,"x%d(%d,1)=%e;\n", myId, i[0]+1, x[0]);
1123         fprintf(fid,"y%d(%d,1)=%e;\n", myId, i[1]+1, x[1]);
1124         fprintf(fid,"p%d(%d,%d)=%e;\n", myId, i[1]+1, i[0]+1, p);
1125         fprintf(fid,"g%d(%d,%d)=%e;\n", myId, i[1]+1, i[0]+1, g);
1126     #endif
1127 #endif
1128     }
1129 }
1130 fclose(fid);
1131
1132 /* Write matlab driver */
1133
1134 if (myId == 0) {
1135
1136     fid = fopen("grad.m","w");
1137
1138     #ifdef USE3D
1139         fprintf(fid,"clear;\nfigure;\nhold_on\n");
1140         fprintf(fid,"trans=0.7;\n");
1141         fprintf(fid,"ecol=none';\n");
1142     #if defined(SUNDIALS_EXTENDED_PRECISION)
1143         fprintf(fid,"xp=[%Lf%Lf];\n",G1_X,G2_X);
1144         fprintf(fid,"yp=[%Lf%Lf];\n",G1_Y,G2_Y);
1145         fprintf(fid,"zp=[%Lf%Lf];\n",G1_Z,G2_Z);
1146     #else
1147         fprintf(fid,"xp=[%f%f];\n",G1_X,G2_X);
1148         fprintf(fid,"yp=[%f%f];\n",G1_Y,G2_Y);
1149         fprintf(fid,"zp=[%f%f];\n",G1_Z,G2_Z);
1150     #endif
1151     fprintf(fid,"ns=length(xp)*length(yp)*length(zp);\n");
1152
1153     for (ip=0; ip<d->npes; ip++) {
1154         fprintf(fid,"\ngrad%03d;\n",ip);
1155         fprintf(fid,"[X,Y,Z]=meshgrid(x%d,y%d,z%d);\n",ip,ip,ip);
1156         fprintf(fid,"s%d=slice(X,Y,Z,g%d,xp,yp,zp);\n",ip,ip);
1157         fprintf(fid,"for i=1:ns\n");
1158         fprintf(fid,"    set(s%d(i),'FaceAlpha',trans);\n",ip);
1159         fprintf(fid,"    set(s%d(i),'EdgeColor',ecol);\n",ip);
1160         fprintf(fid,"end\n");
1161     }
1162
1163     fprintf(fid,"view(3)\n");
1164     fprintf(fid,"\nshading interp\naxis equal\n");
1165 #else
1166     fprintf(fid,"clear;\nfigure;\n");
1167     fprintf(fid,"trans=0.7;\n");
1168     fprintf(fid,"ecol=none';\n");
1169
1170     for (ip=0; ip<d->npes; ip++) {
1171
1172         fprintf(fid,"\ngrad%03d;\n",ip);
1173
1174         fprintf(fid,"\nsubplot(1,2,1)\n");
1175         fprintf(fid,"s=surf(x%d,y%d,g%d);\n",ip,ip,ip);
1176         fprintf(fid,"set(s,'FaceAlpha',trans);\n");
1177         fprintf(fid,"set(s,'EdgeColor',ecol);\n");
1178         fprintf(fid,"hold_on\n");

```

```

1179         fprintf(fid,"axis_tight\n");
1180         fprintf(fid,"box_on\n");
1181
1182         fprintf(fid,"\nsubplot(1,2,2)\n");
1183         fprintf(fid,"s=surf(x%d,y%d,p%d);\n",ip,ip,ip);
1184         fprintf(fid,"set(s,'CData',g%d);\n",ip);
1185         fprintf(fid,"set(s,'FaceAlpha',trans);\n");
1186         fprintf(fid,"set(s,'EdgeColor',ecol);\n");
1187         fprintf(fid,"hold_on\n");
1188         fprintf(fid,"axis_tight\n");
1189         fprintf(fid,"box_on\n");
1190
1191     }
1192 #endif
1193     fclose(fid);
1194 }
1195 }

```