

Example Programs for CVMODES v2.1.0

Alan C. Hindmarsh and Radu Serban

U.S. Department of Energy



November 2004

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 is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

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.

Example Programs for CVODES

v2.1.0

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

Contents

1	Introduction	1
2	Forward sensitivity analysis example problems	5
2.1	A serial nonstiff example: <code>cvfnx</code>	5
2.2	A serial dense example: <code>cvfdx</code>	10
2.3	An SPGMR parallel example with user preconditioner: <code>pvfkx</code>	17
3	Adjoint sensitivity analysis example problems	25
3.1	A serial dense example: <code>cvadx</code>	25
3.2	A parallel nonstiff example: <code>pvanx</code>	29
3.3	An SPGMR parallel example using the CVBBDPRE module: <code>pvakx</code>	32
4	Parallel tests	36
	References	38
A	Listing of <code>cvfnx.c</code>	39
B	Listing of <code>cvfdx.c</code>	49
C	Listing of <code>pvfkx.c</code>	61
D	listing of <code>cvadx.c</code>	86
E	Listing of <code>pvanx.c</code>	97
F	Listing of <code>pvakx.c</code>	110

1 Introduction

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

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

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

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

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

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

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

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

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

The problem is solved with the BDF/GMRES method (i.e. using the CVSPGMR linear solver) and the block-diagonal part of the Newton matrix as a left preconditioner. A copy of the block-diagonal part of the Jacobian is saved and conditionally reused within the preconditioner setup routine.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The adjoint capability of CVODES is used to compute gradients of a functional of the solution with respect to the three reaction rate constants appearing in the model. This program solves both the forward and backward problems with the BDF method, Newton iteration with the CVDENSE linear solver, and user-supplied Jacobian routines.

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

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

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

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

- **pvnx** solves the semi-discrete form of an advection-diffusion equation in 1-D.
This program solves the problem with the option for nonstiff systems, i.e. Adams method and functional iteration.
- **pvkx** is the parallel implementation of **cvkx**.
- **pvkxb** solves the same problem as **pvkx**, with the BDF/GMRES method and a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the module `CVBBDPRE`.
- **pvfnx** is the parallel version of **cvfnx**.
- **pvfkx** is the parallel version of **cvfkx**.
- **pvanx** solves the semi-discrete form of an advection-diffusion equation in 1-D.
The adjoint capability of CVODES is used to compute gradients of the average over space of the solution at the final time with respect to both the initial conditions and the advection and diffusion coefficients in the model. This program solves both the forward and backward problems with the option for nonstiff systems, i.e. Adams method and functional iteration.
- **pvakx** solves an adjoint sensitivity problem for an advection-diffusion PDE in 2-D or 3-D using the BDF/GMRES method and the `CVBBDPRE` preconditioner module on both the forward and backward phases.
The adjoint capability of CVODES is used to compute the gradient of the space-time average of the squared solution norm with respect to problem parameters which parametrize a distributed volume source.

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

The Appendices contain complete listings of the examples described below. We also give our output files for each of these examples, but users should be cautioned that their

results may differ slightly from these. Differences in solution values may differ within the tolerances, and differences in cumulative counters, such as numbers of steps or Newton iterations, may differ from one machine environment to another by as much as 10% to 20%.

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

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

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

2 Forward sensitivity analysis example problems

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

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

2.1 A serial nonstiff example: `cvfnx`

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

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

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

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

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

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

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

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

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

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

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

cvfnx sample output

1-D advection-diffusion equation, mesh size = 10

Sensitivity: YES (SIMULTANEOUS + FULL ERROR CONTROL)

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

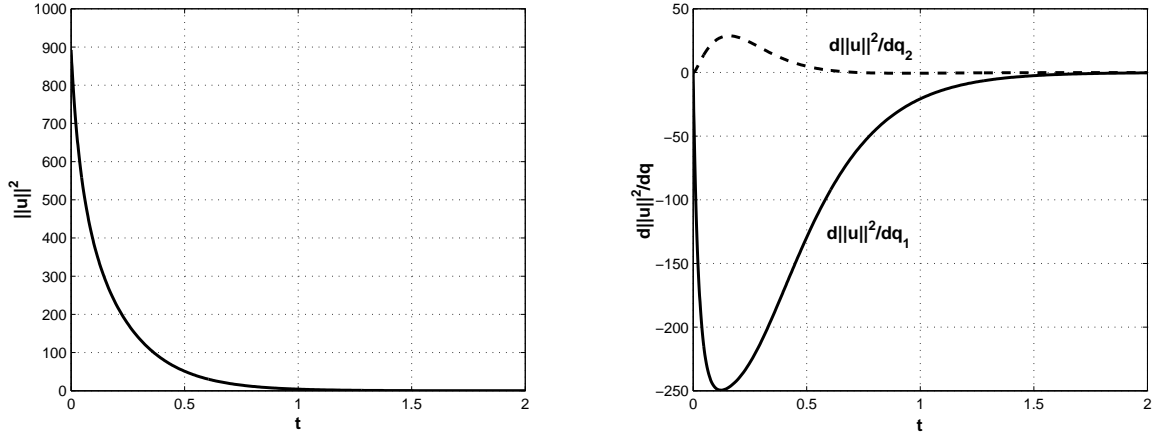


Figure 1: Results for the `cvfnx` example problem. The time evolution of the squared solution norm, $\|u\|^2$, is shown on the left. The figure on the right shows the evolution of the sensitivities of $\|u\|^2$ with respect to the two problem parameters.

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

				Solution	4.6945e-04
				Sensitivity 1	4.5300e-03
				Sensitivity 2	3.8674e-04

4.500e+00	3	4.744e-03	680		
				Solution	1.3422e-04
				Sensitivity 1	1.4548e-03
				Sensitivity 2	1.2589e-04

5.000e+00	1	4.010e-03	757		
				Solution	3.8656e-05
				Sensitivity 1	4.6451e-04
				Sensitivity 2	4.0616e-05

Final Statistics					
nst	=	757			
nfe	=	1372			
netf	=	1	nsetups	=	0
nni	=	1369	ncfn	=	117
nfSe	=	2744	nfeS	=	5488
netfs	=	0	nsetupsS	=	0
nniS	=	0	ncfnS	=	0

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

----- cvfnx sample output -----					
1-D advection-diffusion equation, mesh size = 10					
Sensitivity: YES (STAGGERED + PARTIAL ERROR CONTROL)					
=====					
T	Q	H	NST		Max norm
=====					
5.000e-01	3	7.876e-03	115		
				Solution	3.0529e+00
				Sensitivity 1	3.8668e+00
				Sensitivity 2	6.2020e-01

1.000e+00	3	1.145e-02	208		
				Solution	8.7533e-01
				Sensitivity 1	2.1743e+00
				Sensitivity 2	1.8909e-01

1.500e+00	2	9.985e-03	287		
				Solution	2.4948e-01
				Sensitivity 1	9.1826e-01
				Sensitivity 2	7.3913e-02

2.000e+00	2	4.223e-03	388		
				Solution	7.1096e-02
				Sensitivity 1	3.4667e-01
				Sensitivity 2	2.8228e-02

2.500e+00	2	4.220e-03	507		
				Solution	2.0261e-02
				Sensitivity 1	1.2301e-01
				Sensitivity 2	1.0085e-02

3.000e+00	2	4.220e-03	625		
				Solution	5.7738e-03
				Sensitivity 1	4.1957e-02
				Sensitivity 2	3.4557e-03

3.500e+00	2	4.220e-03	744		
				Solution	1.6454e-03
				Sensitivity 1	1.3923e-02
				Sensitivity 2	1.1670e-03

4.000e+00	2	4.220e-03	862		
				Solution	4.6887e-04
				Sensitivity 1	4.5282e-03
				Sensitivity 2	3.8632e-04

4.500e+00	2	4.220e-03	981		
				Solution	1.3364e-04
				Sensitivity 1	1.4502e-03
				Sensitivity 2	1.2546e-04

5.000e+00	2	4.220e-03	1099		
				Solution	3.8105e-05
				Sensitivity 1	4.5891e-04
				Sensitivity 2	4.0166e-05

Final Statistics					
nst	=	1099			
nfe	=	3157			
netf	=	3	nsetups	=	0
nni	=	1657	ncfn	=	11
nfSe	=	4838	nfeS	=	9676
netfs	=	0	nsetupsS	=	0
nniS	=	2418	ncfnS	=	398

2.2 A serial dense example: cvfdx

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

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

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

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

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 `cvdx` in [1], will also provide the user with a template for instrumenting an existing simulation code to perform forward sensitivity analysis. As will be seen from this example, an existing simulation code can be modified to compute sensitivity variables (in addition to state variables) by only inserting a few CVODES calls into the main program.

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

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

Three 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`.

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

The `main` function begins with definitions and type declarations. Among these, it defines the vector `pbar` of `NP` scaling factors for the model parameters `p`, the array `plist` of integer flags specifying the sensitivity parameters among the model parameters, and the array `ys` of

`N_Vector` which will contain the initial conditions and solutions for the sensitivity variables. It also declares the variable `data` of type `UserData` which will contain the user-defined data structure to be passed to CVODES and used in the evaluation of the ODE right-hand sides.

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

The user's data structure is then allocated and its field `p` is set to contain the values of the three problem parameters. The next block of code is identical to that in `cvdx.c` (see [1]) and involves allocation and initialization of the state variables and integration tolerances for the state variables, and creation and initialization of `cvode_mem`, the CVODES solver memory. 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) and the array of flags `plist`. The choice $plist_i = i + 1$ indicates that sensitivities with respect to all model parameters will be computed. Next, the program allocates memory for `yS`, by calling the `NVECTOR_SERIAL` function `N_VNewVectorArray_Serial`, and initializes all sensitivity variables to 0.0.

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 scalings for the model parameters. In this example, `pbar` is needed for the estimation of absolute sensitivity variables tolerances. The call to `CVodeSensMalloc` specifies the sensitivity solution method through `sensi_meth` (read from the command line arguments) as `CV_SIMULTANEOUS`, `CV_STAGGERED`, or `CV_STAGGERED1`. 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 `NOOUT` output times, the program calls the integration routine `CVode` which, if sensitivity analysis was initialized through the call to `CVodeSensMalloc`, computes both state and sensitivity variables. However, `CVode` returns only the state solution at `tout` in the vector `y`. The program tests the return from `CVode` for a value other than `CV_SUCCESS` and prints the state variables. Sensitivity variables at `tout` are loaded into `yS` by calling `CVodeGetSens`. The program tests the return from `CVodeGetSens` for a value other than `CV_SUCCESS` and then prints the sensitivity variables.

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

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

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

```
% cvfdx -nosensi
```

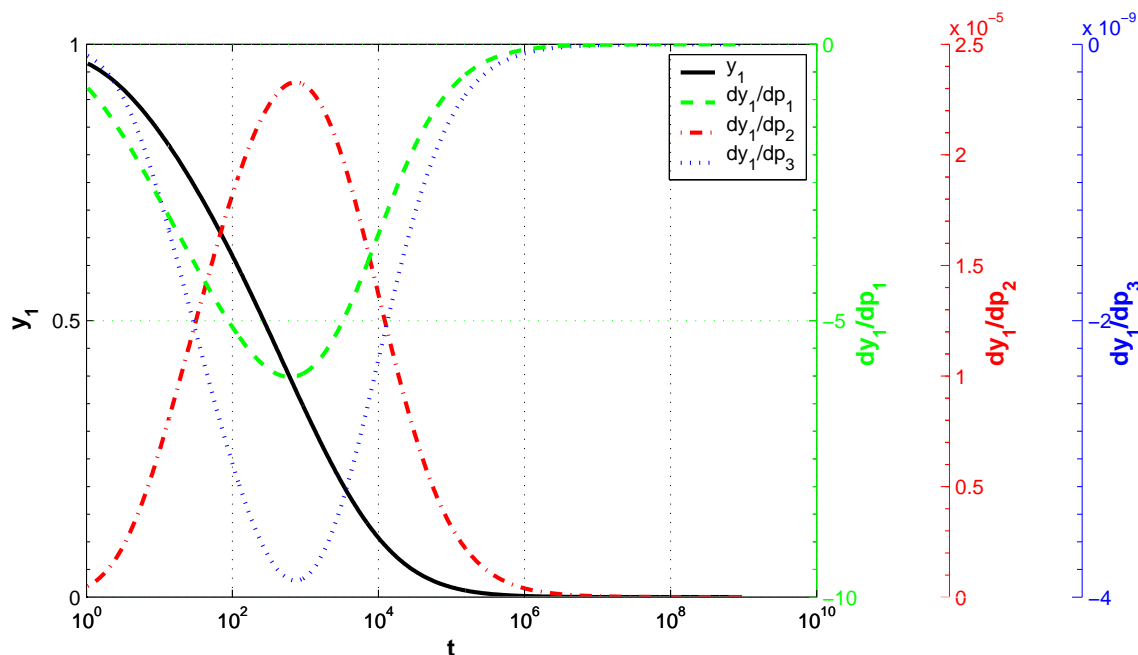


Figure 2: Results for the `cvfdx` example problem: time evolution of y_1 and its sensitivities with respect to the three problem parameters.

if no sensitivity calculations are desired, or

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

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

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

cvfdx sample output							
3-species chemical kinetics problem							
Sensitivity: YES (SIMULTANEOUS + FULL ERROR CONTROL)							
T	Q	H	NST	y1	y2	y3	
4.000e-01	3	4.881e-02	115				
			Solution	9.8517e-01	3.3864e-05	1.4794e-02	
			Sensitivity 1	-3.5595e-01	3.9025e-04	3.5556e-01	
			Sensitivity 2	9.5431e-08	-2.1309e-10	-9.5218e-08	
			Sensitivity 3	-1.5833e-11	-5.2900e-13	1.6362e-11	
4.000e+00	5	2.363e-01	138				
			Solution	9.0552e-01	2.2405e-05	9.4459e-02	
			Sensitivity 1	-1.8761e+00	1.7922e-04	1.8759e+00	
			Sensitivity 2	2.9614e-06	-5.8305e-10	-2.9608e-06	

			Sensitivity 3	-4.9334e-10	-2.7626e-13	4.9362e-10

4.000e+01	3	1.485e+00	219			
			Solution	7.1583e-01	9.1856e-06	2.8416e-01
			Sensitivity 1	-4.2475e+00	4.5913e-05	4.2475e+00
			Sensitivity 2	1.3731e-05	-2.3573e-10	-1.3730e-05
			Sensitivity 3	-2.2883e-09	-1.1380e-13	2.2884e-09

4.000e+02	3	8.882e+00	331			
			Solution	4.5052e-01	3.2229e-06	5.4947e-01
			Sensitivity 1	-5.9584e+00	3.5431e-06	5.9584e+00
			Sensitivity 2	2.2738e-05	-2.2605e-11	-2.2738e-05
			Sensitivity 3	-3.7896e-09	-4.9948e-14	3.7897e-09

4.000e+03	2	9.644e+01	474			
			Solution	1.8318e-01	8.9412e-07	8.1682e-01
			Sensitivity 1	-4.7500e+00	-5.9945e-06	4.7500e+00
			Sensitivity 2	1.8809e-05	2.3130e-11	-1.8809e-05
			Sensitivity 3	-3.1348e-09	-1.8758e-14	3.1348e-09

4.000e+04	3	8.330e+02	561			
			Solution	3.8975e-02	1.6214e-07	9.6102e-01
			Sensitivity 1	-1.5748e+00	-2.7619e-06	1.5748e+00
			Sensitivity 2	6.2867e-06	1.1002e-11	-6.2867e-06
			Sensitivity 3	-1.0478e-09	-4.5361e-15	1.0478e-09

4.000e+05	3	1.304e+04	619			
			Solution	4.9392e-03	1.9854e-08	9.9506e-01
			Sensitivity 1	-2.3640e-01	-4.5859e-07	2.3640e-01
			Sensitivity 2	9.4529e-07	1.8333e-12	-9.4529e-07
			Sensitivity 3	-1.5753e-10	-6.3634e-16	1.5753e-10

4.000e+06	4	1.893e+05	671			
			Solution	5.1680e-04	2.0682e-09	9.9948e-01
			Sensitivity 1	-2.5663e-02	-5.1050e-08	2.5663e-02
			Sensitivity 2	1.0265e-07	2.0418e-13	-1.0265e-07
			Sensitivity 3	-1.7109e-11	-6.8508e-17	1.7109e-11

4.000e+07	4	1.597e+06	724			
			Solution	5.2038e-05	2.0816e-10	9.9995e-01
			Sensitivity 1	-2.5993e-03	-5.1940e-09	2.5993e-03
			Sensitivity 2	1.0397e-08	2.0776e-14	-1.0397e-08
			Sensitivity 3	-1.7330e-12	-6.9327e-18	1.7330e-12

4.000e+08	4	2.242e+07	768			
			Solution	5.2083e-06	2.0833e-11	9.9999e-01
			Sensitivity 1	-2.6037e-04	-5.2042e-10	2.6037e-04
			Sensitivity 2	1.0415e-09	2.0817e-15	-1.0415e-09
			Sensitivity 3	-1.7359e-13	-6.9437e-19	1.7359e-13

4.000e+09	3	2.632e+08	813			
			Solution	5.2116e-07	2.0846e-12	1.0000e-00
			Sensitivity 1	-2.6075e-05	-5.2219e-11	2.6075e-05
			Sensitivity 2	1.0430e-10	2.0888e-16	-1.0430e-10

			Sensitivity 3	-1.7372e-14	-6.9486e-20	1.7372e-14
4.000e+10	3	2.187e+09	836			
			Solution	2.7766e-08	1.1107e-13	1.0000e-00
			Sensitivity 1	-2.4992e-06	-7.2304e-12	2.4992e-06
			Sensitivity 2	9.9970e-12	2.8922e-17	-9.9970e-12
			Sensitivity 3	-9.2555e-16	-3.7022e-21	9.2556e-16

Final Statistics						
nst	=	836				
nfe	=	1149				
netf	=	24	nsetups	=	134	
nni	=	1146	ncfn	=	4	
nfSe	=	3447	nfeS	=	0	
netfs	=	0	nsetupsS	=	0	
nniS	=	0	ncfnS	=	0	
njeD	=	24	nfeD	=	0	

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

----- cvfdx sample output -----						
3-species chemical kinetics problem						
Sensitivity: YES (STAGGERED + PARTIAL ERROR CONTROL)						
=====						
T	Q	H	NST	y1	y2	y3
=====						
4.000e-01	3	1.205e-01	59			
			Solution	9.8517e-01	3.3863e-05	1.4797e-02
			Sensitivity 1	-3.5611e-01	3.9023e-04	3.5572e-01
			Sensitivity 2	9.4831e-08	-2.1325e-10	-9.4618e-08
			Sensitivity 3	-1.5733e-11	-5.2897e-13	1.6262e-11

4.000e+00	4	5.316e-01	74			
			Solution	9.0552e-01	2.2404e-05	9.4461e-02
			Sensitivity 1	-1.8761e+00	1.7922e-04	1.8760e+00
			Sensitivity 2	2.9612e-06	-5.8308e-10	-2.9606e-06
			Sensitivity 3	-4.9330e-10	-2.7624e-13	4.9357e-10

4.000e+01	3	1.445e+00	116			
			Solution	7.1584e-01	9.1854e-06	2.8415e-01
			Sensitivity 1	-4.2474e+00	4.5928e-05	4.2473e+00
			Sensitivity 2	1.3730e-05	-2.3573e-10	-1.3729e-05
			Sensitivity 3	-2.2883e-09	-1.1380e-13	2.2884e-09

4.000e+02	3	1.605e+01	164			

			Solution	4.5054e-01	3.2228e-06	5.4946e-01
			Sensitivity 1	-5.9582e+00	3.5498e-06	5.9582e+00
			Sensitivity 2	2.2737e-05	-2.2593e-11	-2.2737e-05
			Sensitivity 3	-3.7895e-09	-4.9947e-14	3.7896e-09

4.000e+03	3	1.474e+02	227			
			Solution	1.8321e-01	8.9422e-07	8.1679e-01
			Sensitivity 1	-4.7501e+00	-5.9934e-06	4.7501e+00
			Sensitivity 2	1.8809e-05	2.3126e-11	-1.8809e-05
			Sensitivity 3	-3.1348e-09	-1.8759e-14	3.1348e-09

4.000e+04	3	2.331e+03	307			
			Solution	3.8978e-02	1.6215e-07	9.6102e-01
			Sensitivity 1	-1.5749e+00	-2.7623e-06	1.5749e+00
			Sensitivity 2	6.2868e-06	1.1001e-11	-6.2868e-06
			Sensitivity 3	-1.0479e-09	-4.5364e-15	1.0479e-09

4.000e+05	3	2.342e+04	349			
			Solution	4.9410e-03	1.9861e-08	9.9506e-01
			Sensitivity 1	-2.3638e-01	-4.5834e-07	2.3638e-01
			Sensitivity 2	9.4515e-07	1.8319e-12	-9.4515e-07
			Sensitivity 3	-1.5757e-10	-6.3653e-16	1.5757e-10

4.000e+06	4	1.723e+05	391			
			Solution	5.1690e-04	2.0686e-09	9.9948e-01
			Sensitivity 1	-2.5662e-02	-5.1036e-08	2.5662e-02
			Sensitivity 2	1.0264e-07	2.0412e-13	-1.0264e-07
			Sensitivity 3	-1.7110e-11	-6.8509e-17	1.7110e-11

4.000e+07	4	4.952e+06	439			
			Solution	5.1984e-05	2.0795e-10	9.9995e-01
			Sensitivity 1	-2.5970e-03	-5.1903e-09	2.5970e-03
			Sensitivity 2	1.0388e-08	2.0761e-14	-1.0388e-08
			Sensitivity 3	-1.7312e-12	-6.9256e-18	1.7312e-12

4.000e+08	3	2.444e+07	491			
			Solution	5.2121e-06	2.0849e-11	9.9999e-01
			Sensitivity 1	-2.6067e-04	-5.2146e-10	2.6067e-04
			Sensitivity 2	1.0427e-09	2.0858e-15	-1.0427e-09
			Sensitivity 3	-1.7385e-13	-6.9541e-19	1.7385e-13

4.000e+09	4	1.450e+08	525			
			Solution	5.0539e-07	2.0216e-12	1.0000e-00
			Sensitivity 1	-2.6111e-05	-5.3906e-11	2.6111e-05
			Sensitivity 2	1.0445e-10	2.1562e-16	-1.0445e-10
			Sensitivity 3	-1.7437e-14	-6.9746e-20	1.7437e-14

4.000e+10	5	7.934e+08	579			
			Solution	5.9422e-08	2.3769e-13	1.0000e-00
			Sensitivity 1	-2.8267e-06	-5.3645e-12	2.8267e-06
			Sensitivity 2	1.1307e-11	2.1458e-17	-1.1307e-11
			Sensitivity 3	-1.7387e-15	-6.9549e-21	1.7387e-15

Final Statistics

nst = 579

nfe = 1379

netf = 25 nsetups = 109

nni = 797 ncfn = 0

nfSe = 2832 nfeS = 0

netfs = 0 nsetupsS = 3

nniS = 943 ncfnS = 0

njeD = 11 nfeD = 0

2.3 An SPGMR parallel example with user preconditioner: pvfxx

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

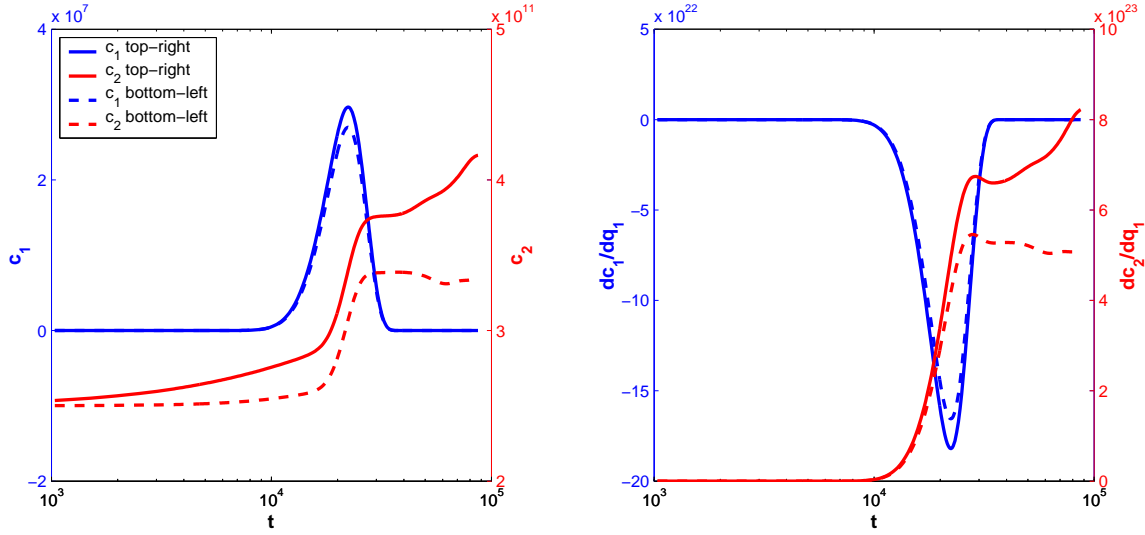


Figure 3: Results for the `pvfkn` example problem: time evolution of c_1 and c_2 at the bottom-left and top-right corners (left) and of their sensitivities with respect to q_1 .

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

The source code for this example is listed in App. C. The overall structure of the `main` function is very similar to that of the code `cvfdx` described above with differences arising from the use of the parallel `NVECTOR` module - `NVECTOR_PARALLEL`. On the other hand, the user-supplied routines in `pvfkn`, `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 `pvkn` 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 `pvfkn` 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 `pvfkn` are shown in Fig. 3. These results were generated on a $(2 \times 40) \times (2 \times 40)$ grid.

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

```
% mpirun -np nproc pvfkn -nosensi
```

if no sensitivity calculations are desired, or

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

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

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

2-species diurnal advection-diffusion problem

Sensitivity: YES (SIMULTANEOUS + FULL ERROR CONTROL)

T	Q	H	NST		Bottom left	Top right
7.200e+03	3	3.518e+01	543			
				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 -2.7843e+18
1.440e+04	3	6.557e+01	880			
				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	4.727e+01	1143			
				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 -9.4501e+22
2.880e+04	2	2.825e+01	1464			
				Solution	8.7021e+06 3.3804e+11	9.6500e+06 3.7510e+11
				Sensitivity 1	-5.3375e+22 5.4487e+23	-5.9187e+22 6.7430e+23
				Sensitivity 2	-4.8855e+18 -1.7194e+23	-6.1040e+18 -2.1518e+23
3.600e+04	4	5.126e+01	1595			
				Solution	1.4040e+04 3.3868e+11	1.5609e+04 3.7652e+11
				Sensitivity 1	-8.6141e+19	-9.5761e+19

					5.2718e+23	6.6029e+23

				Sensitivity 2	-8.4327e+15	-1.0549e+16
					-1.8439e+23	-2.3096e+23

4.320e+04	4	1.712e+02	2692			
				Solution	1.6522e-06	1.8674e-06
					3.3823e+11	3.8035e+11

				Sensitivity 1	5.6192e+09	6.2910e+09
					5.2753e+23	6.7447e+23

				Sensitivity 2	4.1977e+04	5.2892e+04
					-1.8454e+23	-2.3595e+23

5.040e+04	4	1.650e+02	2728			
				Solution	-2.6212e-06	-3.0111e-06
					3.3582e+11	3.8644e+11

				Sensitivity 1	-1.3942e+11	-1.6021e+11
					5.2066e+23	6.9664e+23

				Sensitivity 2	-1.4786e+09	-1.9410e+09
					-1.8214e+23	-2.4370e+23

5.760e+04	4	1.650e+02	2771			
				Solution	6.8963e-06	7.7078e-06
					3.3203e+11	3.9090e+11

				Sensitivity 1	7.1758e+11	8.0131e+11
					5.0825e+23	7.1205e+23

				Sensitivity 2	-7.0945e+09	-8.9904e+09
					-1.7780e+23	-2.4909e+23

6.480e+04	3	1.915e+02	2828			
				Solution	2.9461e-08	3.1789e-08
					3.3130e+11	3.9634e+11

				Sensitivity 1	3.1873e+10	3.4505e+10
					5.0442e+23	7.3273e+23

				Sensitivity 2	-5.8848e+07	-7.2835e+07
					-1.7646e+23	-2.5633e+23

7.200e+04	5	5.520e+02	2850			
				Solution	-1.1692e-11	-1.2544e-11
					3.3297e+11	4.0389e+11

				Sensitivity 1	-8.9686e+07	-9.6814e+07
					5.0783e+23	7.6382e+23

				Sensitivity 2	-5.2476e+04	-6.5984e+04
					-1.7765e+23	-2.6720e+23


```

-----
7.920e+04  5  5.520e+02  2863
Solution      -1.5543e-14  -1.6649e-14
              3.3344e+11   4.1203e+11
-----
Sensitivity 1   5.6155e+06   6.0433e+06
              5.0730e+23   7.9959e+23
-----
Sensitivity 2   3.1785e+03   3.9830e+03
              -1.7747e+23  -2.7972e+23
-----

8.640e+04  5  5.520e+02  2876
Solution      -7.1964e-17  -7.7677e-17
              3.3518e+11   4.1625e+11
-----
Sensitivity 1  -4.0473e+05  -4.3422e+05
              5.1171e+23   8.2142e+23
-----
Sensitivity 2  -2.2891e+02  -2.8592e+02
              -1.7901e+23  -2.8736e+23
-----

Final Statistics

nst      = 2876

nfe      = 3817
netf     = 138   nsetups = 411
nni      = 3814  ncfn    = 4

nfSe     = 7634   nfeS    = 15268
netfs    = 0     nsetupsS = 0
nniS     = 0     ncfnS    = 0

```

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

```

----- pvfkk 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	2.067e+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	6.721e+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	3.910e+02	405		
				Solution	1.2884e-08
					1.1021e-06
					3.3823e+11
					3.8035e+11

				Sensitivity 1	5.0965e+08
					-5.7742e+09
					5.2753e+23
					6.7448e+23

				Sensitivity 2	-2.7179e+06
					-9.7347e+06
					-1.8454e+23
					-2.3595e+23

5.040e+04	4	4.146e+02	437		
				Solution	-1.7388e-07
					-7.8035e-07

					3.3582e+11	3.8644e+11

				Sensitivity 1	6.4992e+09	2.3414e+10
					5.2067e+23	6.9664e+23

				Sensitivity 2	-3.5657e+07	-4.6120e+08
					-1.8214e+23	-2.4370e+23

5.760e+04	4	1.821e+02	456			
				Solution	-4.1077e-07	-1.7367e-06
					3.3203e+11	3.9090e+11

				Sensitivity 1	2.2547e+09	9.5107e+09
					5.0825e+23	7.1205e+23

				Sensitivity 2	-2.5418e+07	-3.0921e+08
					-1.7780e+23	-2.4910e+23

6.480e+04	4	5.349e+02	471			
				Solution	-6.7422e-11	-2.7778e-10
					3.3130e+11	3.9634e+11

				Sensitivity 1	-1.4830e+07	-5.7126e+07
					5.0442e+23	7.3274e+23

				Sensitivity 2	-2.0493e+06	-2.5006e+07
					-1.7646e+23	-2.5633e+23

7.200e+04	4	5.349e+02	485			
				Solution	-5.6725e-15	-2.3106e-14
					3.3297e+11	4.0388e+11

				Sensitivity 1	3.3206e+05	1.2740e+06
					5.0783e+23	7.6382e+23

				Sensitivity 2	4.8796e+04	5.9585e+05
					-1.7765e+23	-2.6720e+23

7.920e+04	4	5.349e+02	498			
				Solution	1.3597e-18	5.5900e-18
					3.3344e+11	4.1203e+11

				Sensitivity 1	-3.1852e+02	-1.2223e+03
					5.0730e+23	7.9960e+23

				Sensitivity 2	-4.6725e+01	-5.7127e+02
					-1.7747e+23	-2.7972e+23

8.640e+04	4	5.349e+02	512			
				Solution	5.7715e-23	2.3512e-22
					3.3518e+11	4.1625e+11

				Sensitivity 1	5.0307e-03	1.9328e-02
					5.1171e+23	8.2142e+23

Sensitivity 2		7.3228e-04	9.0116e-03
		-1.7901e+23	-2.8736e+23

Final Statistics			
nst	=	512	
nfe	=	1192	
netf	=	39	nsetups = 100
nni	=	677	ncfn = 0
nfSe	=	1312	nfeS = 2624
netfs	=	0	nsetupsS = 0
nniS	=	655	ncfnS = 0

3 Adjoint sensitivity analysis example problems

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

3.1 A serial dense example: `cvadx`

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

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

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

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

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

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

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` and `cvodea.h` for CVODES definitions and interface function prototypes, the header file `cvdense.h` for the CVDENSE linear solver module, and the header file `nvector_serial.h` for the definition of the serial implementation of the NVECTOR module - NVECTOR_SERIAL. 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 `reltol` and vector absolute tolerance `abstol` for the state variables as well as the absolute tolerance for the quadrature variable.

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 next call specifies the optional user data pointer, while the call to `CVodeMalloc` initializes the forward integration by specifying the initial conditions and integration tolerances. 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 calls to `CVDenseSetJacFn` and `CVDenseSetJacData`, respectively.

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.

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

The next segment of code deals with the setup of the backward problem. First, a serial vector `yB` of length `NEQ` is allocated and 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 `fQ` implements (16), while `fB`, `JacB`, and `fQB` are mere translations of the backward problem (14) and (15).

The output generated by `cvadx` is shown below.

```

----- cvadx sample output -----

Adjoint Sensitivity Example for Chemical Kinetics
-----

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

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

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

List of Check Points (ncheck = 2)
  2  addr: 0x8072a30  time = [ 2.070e+04 4.202e+07 ]  next: 0x8072210
  1  addr: 0x8072210  time = [ 9.254e+01 2.070e+04 ]  next: 0x8064598
  0  addr: 0x8064598  time = [ 0.000e+00 9.254e+01 ]  next: (nil)

Create and allocate CVODES memory for backward run
Integrate backwards
-----
tB0:           4.0000e+07

```

dG/dp:	7.6998e+05	-3.0740e+00	5.0750e-04
lambda(t0):	3.9967e+07	3.9967e+07	3.9967e+07

Re-initialize CVODES memory for backward run
Integrate backwards

tB0:	4.0000e+07		
dG/dp:	1.7335e+02	-5.0534e-04	8.4218e-08
lambda(t0):	8.4156e+00	1.6093e+01	1.6094e+01

Free memory

3.2 A parallel nonstiff example: pvanx

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

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

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

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

and

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

where 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 `cvadx` discussed previously with differences arising from the use of the parallel `NVECTOR` module. Unlike `cvadx`, the example `pvanx` illustrates computation of the additional quadrature variables by appending `NP` equations to the adjoint system. This approach can be a better alternative to using special treatment of the quadrature equations when their number is too small for parallel treatment.

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

```

----- pvanx sample output -----
(PE# 3) Number of check points: 6
 6 addr: 0x80cb850 time = [ 2.241e+00 2.500e+00 ] next: 0x80cb430
 5 addr: 0x80cb430 time = [ 1.853e+00 2.241e+00 ] next: 0x80cb010
 4 addr: 0x80cb010 time = [ 1.464e+00 1.853e+00 ] next: 0x80cabf0
 3 addr: 0x80cabf0 time = [ 1.054e+00 1.464e+00 ] next: 0x80ca7d0
 2 addr: 0x80ca7d0 time = [ 6.938e-01 1.054e+00 ] next: 0x80ca320
 1 addr: 0x80ca320 time = [ 2.896e-01 6.938e-01 ] next: 0x80ba630
 0 addr: 0x80ba630 time = [ 0.000e+00 2.896e-01 ] next: (nil)

g(tf) = 2.130309e-02

```

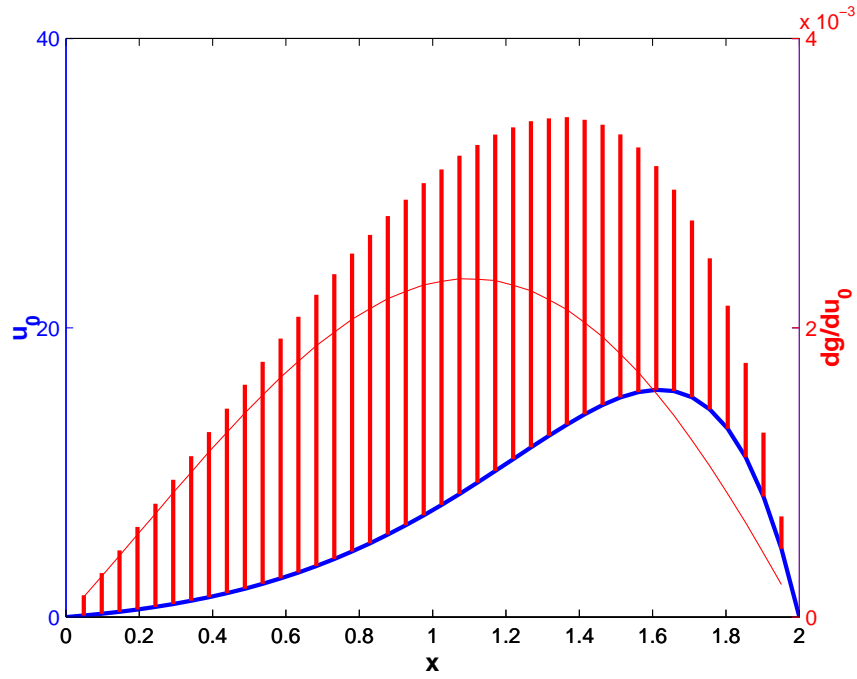


Figure 4: Results for the pvanx example problem. The gradient of $g(t_f)$ with respect to the initial conditions u_0 is shown superimposed over the values u_0 .

```

dgdg(tf)
[ 1]: -1.129203e+00
[ 2]: -1.008884e+00

mu(t0)
[ 1]: 2.775483e-04
[ 2]: 5.619516e-04
[ 3]: 8.473974e-04
[ 4]: 1.126360e-03
[ 5]: 1.393213e-03
[ 6]: 1.639532e-03
[ 7]: 1.860431e-03
[ 8]: 2.047303e-03
[ 9]: 2.196545e-03
[10]: 2.300169e-03
[11]: 2.356330e-03
[12]: 2.358484e-03
[13]: 2.306893e-03
[14]: 2.197231e-03
[15]: 2.032050e-03
[16]: 1.809876e-03
[17]: 1.535540e-03
[18]: 1.210842e-03
[19]: 8.426592e-04
[20]: 4.362227e-04

```

3.3 An SPGMR parallel example using the CVBBDPRE module: pvakx

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

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

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

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

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

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

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

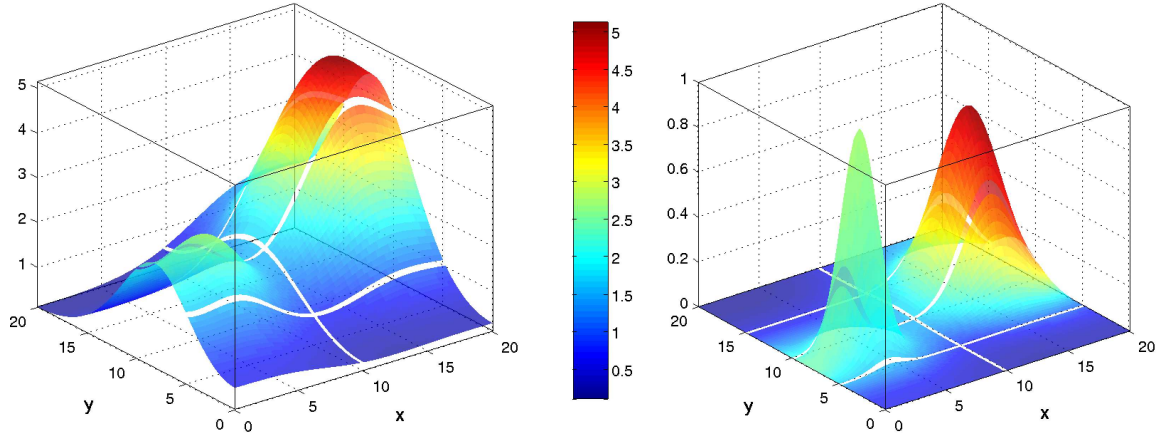


Figure 5: Results for the `pvakx` example problem in 2D. The gradient with respect to the source parameters is pictured on the left. On the right, the gradient was color coded and superimposed over the nominal value of the source parameters.

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

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

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

```

pvakx sample output

Parallel Krylov adjoint sensitivity analysis example
2D Advection diffusion PDE with homogeneous Neumann B.C.
Computes gradient of  $G = \int_{\Omega} c_i^2 dt$ 
with respect to the source values at each grid point.

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

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

Final Statistics..

lenrw   =   8650       leniw =   160

```

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

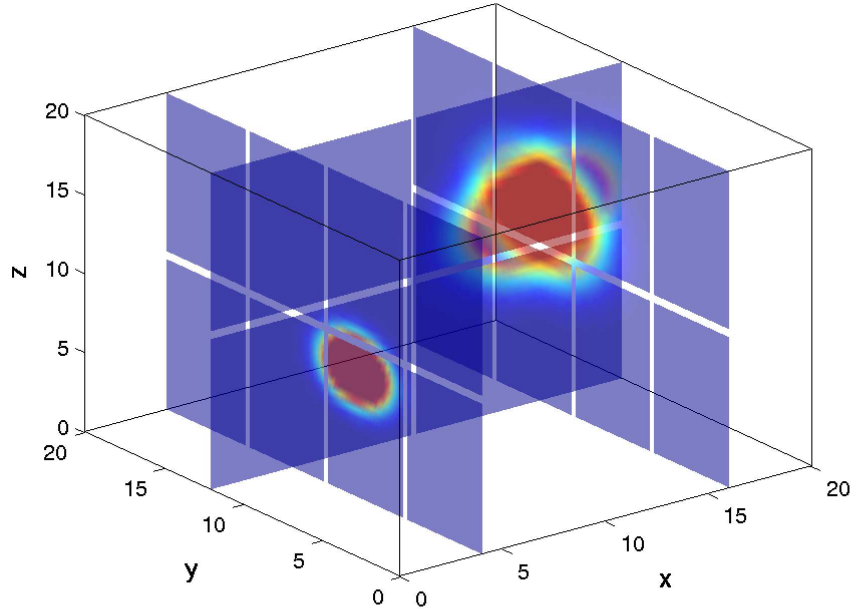


Figure 6: Results for the `pvakx` example problem in 3D. Nominal values of the source parameters.

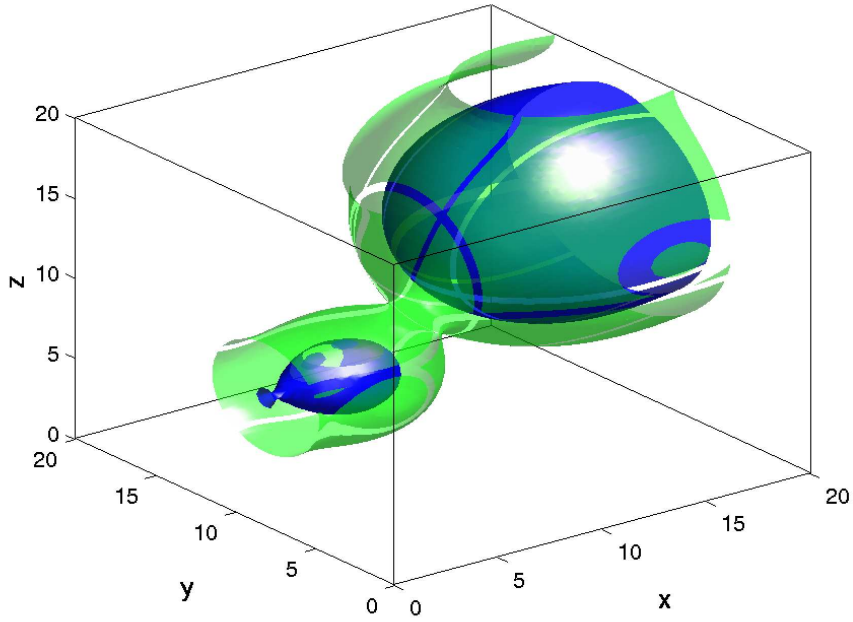


Figure 7: Results for the `pvakx` example problem in 3D. Two isosurfaces of the gradient with respect to the source parameters. They correspond to values of 0.25 (green) and 0.4 (blue).

llrw	=	8656	lliw	=	80
nst	=	104			
nfe	=	108	nfel	=	126
nni	=	105	nli	=	126
nsetups	=	16	netf	=	0
npe	=	2	nps	=	215
ncfn	=	0	ncfl	=	0

Begin backward integration... done.

Final Statistics..

lenrw	=	17220	leniw	=	160
llrw	=	8656	lliw	=	80
nst	=	78			
nfe	=	90	nfel	=	138
nni	=	87	nli	=	138
nsetups	=	17	netf	=	0
npe	=	2	nps	=	217
ncfn	=	0	ncfl	=	0

4 Parallel tests

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

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

where

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

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.

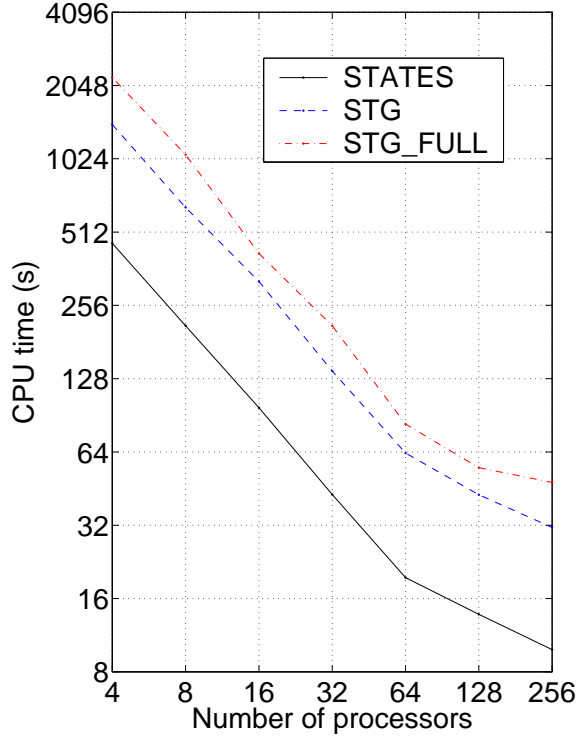


Figure 8: Speedup results for the integration of the state equations only (solid line and column 'STATES'), staggered sensitivity analysis without error control on the sensitivity variables (dashed line and column 'STG'), and staggered sensitivity analysis with full error control (dotted line and column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

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

References

- [1] A. C. Hindmarsh and R. Serban. Example Programs for CVODE v2.2.0. Technical report, LLNL, 2004. UCRL-SM-208110.
- [2] A. C. Hindmarsh and R. Serban. User Documentation for CVODES v2.1.0. Technical report, LLNL, 2004. UCRL-SM-208111.
- [3] R. Serban and A. C. Hindmarsh. CVODES, an ODE solver with sensitivity analysis capabilities. Technical Report UCRL-TR-xxxxxx, LLNL, 2004.

A Listing of cvfnx.c

```

1  /*
2  * -----
3  * $Revision: 1.17 $
4  * $Date: 2004/11/22 20:50:25 $
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 *   % cvsnx -nosensi
39 * If sensitivities are to be computed:
40 *   % cvsnx -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 #include "sundialstypes.h"
51 #include "cvodes.h"
52 #include "nvector_serial.h"

```

```

53
54 /* Problem Constants */
55 #define XMAX RCONST(2.0) /* domain boundary */
56 #define MX 10 /* mesh dimension */
57 #define NEQ MX /* number of equations */
58 #define ATOL RCONST(1.e-5) /* scalar absolute tolerance */
59 #define T0 RCONST(0.0) /* initial time */
60 #define T1 RCONST(0.5) /* first output time */
61 #define DTOUT RCONST(0.5) /* output time increment */
62 #define NOUT 10 /* number of output times */
63
64 #define NP 2
65 #define NS 2
66
67 #define ZERO RCONST(0.0)
68
69 /* Type : UserData
70 contains problem parameters, grid constants, work array. */
71
72 typedef struct {
73     realtype *p;
74     realtype dx;
75 } *UserData;
76
77 /* Functions Called by the CVODES Solver */
78
79 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
80
81 /* Private Helper Functions */
82
83 static void ProcessArgs(int argc, char *argv[],
84                         booleantype *sensi, int *sensi_meth,
85                         booleantype *err_con);
86 static void WrongArgs(char *name);
87 static void SetIC(N_Vector u, realtype dx);
88 static void PrintOutput(void *cnode_mem, realtype t, N_Vector u);
89 static void PrintOutputS(N_Vector *uS);
90 static void PrintFinalStats(void *cnode_mem, booleantype sensi);
91
92 static int check_flag(void *flagvalue, char *funcname, int opt);
93
94 /*
95  *-----
96  * MAIN PROGRAM
97  *-----
98  */
99
100 int main(int argc, char *argv[])
101 {
102     void *cnode_mem;
103     UserData data;
104     realtype dx, reltol, abstol, t, tout;
105     N_Vector u;
106     int iout, flag;

```

```

107
108     realtype *pbar;
109     int is, *plist;
110     N_Vector *uS;
111     booleantype sensi, err_con;
112     int sensi_meth;
113
114     cvode_mem = NULL;
115     data = NULL;
116     u = NULL;
117     pbar = NULL;
118     plist = NULL;
119     uS = NULL;
120
121     /* Process arguments */
122     ProcessArgs(argc, argv, &sensi, &sensi_meth, &err_con);
123
124     /* Set user data */
125     data = (UserData) malloc(sizeof *data); /* Allocate data memory */
126     if(check_flag((void *)data, "malloc", 2)) return(1);
127     data->p = (realtype *) malloc(NP * sizeof(realtype));
128     dx = data->dx = XMAX/((realtype)(MX+1));
129     data->p[0] = RCONST(1.0);
130     data->p[1] = RCONST(0.5);
131
132     /* Allocate and set initial states */
133     u = N_VNew_Serial(NEQ);
134     if(check_flag((void *)u, "N_VNew_Serial", 0)) return(1);
135     SetIC(u, dx);
136
137     /* Set integration tolerances */
138     reltol = ZERO;
139     abstol = ATOL;
140
141     /* Create CVODES object */
142     cvode_mem = CNodeCreate(CV_ADAMS, CV_FUNCTIONAL);
143     if(check_flag((void *)cvode_mem, "CNodeCreate", 0)) return(1);
144
145     flag = CNodeSetFdata(cvode_mem, data);
146     if(check_flag(&flag, "CNodeSetFdata", 1)) return(1);
147
148     /* Allocate CVODES memory */
149     flag = CNodeMalloc(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
150     if(check_flag(&flag, "CNodeMalloc", 1)) return(1);
151
152     printf("\n1-D advection-diffusion equation, mesh size =%3d\n", MX);
153
154     /* Sensitivity-related settings */
155     if(sensi) {
156
157         pbar = (realtype *) malloc(NP * sizeof(realtype));
158         if(check_flag((void *)pbar, "malloc", 2)) return(1);
159         pbar[0] = RCONST(1.0);
160         pbar[1] = RCONST(0.5);

```

```

161     plist = (int *) malloc(NS * sizeof(int));
162     if(check_flag((void *)plist, "malloc", 2)) return(1);
163     for(is=0; is<NS; is++)
164         plist[is] = is+1; /* sensitivity w.r.t. i-th parameter */
165
166     uS = N_VNewVectorArray_Serial(NS, NEQ);
167     if(check_flag((void *)uS, "N_VNew", 0)) return(1);
168     for(is=0; is<NS; is++)
169         N_VConst(ZERO, uS[is]);
170
171     flag = CNodeSetSensErrCon(cvode_mem, err_con);
172     if(check_flag(&flag, "CNodeSetSensErrCon", 1)) return(1);
173
174     flag = CNodeSetSensRho(cvode_mem, ZERO);
175     if(check_flag(&flag, "CNodeSetSensRho", 1)) return(1);
176
177     flag = CNodeSetSensPbar(cvode_mem, pbar);
178     if(check_flag(&flag, "CNodeSetSensPbar", 1)) return(1);
179
180     flag = CNodeSensMalloc(cvode_mem, NS, sensi_meth, data->p, plist, uS);
181     if(check_flag(&flag, "CNodeSensMalloc", 1)) return(1);
182
183     printf("Sensitivity: YES ");
184     if(sensi_meth == CV_SIMULTANEOUS)
185         printf("( SIMULTANEOUS +");
186     else
187         if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
188         else printf("( STAGGERED1 +");
189     if(err_con) printf(" FULL ERROR CONTROL ");
190     else printf(" PARTIAL ERROR CONTROL ");
191
192 } else {
193
194     printf("Sensitivity: NO ");
195
196 }
197
198 /* In loop over output points, call CNode, print results, test for error */
199
200 printf("\n\n");
201 printf("=====\n");
202 printf("      T      Q      H      NST                      Max norm   \n");
203 printf("=====\n");
204
205 for (iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
206     flag = CNode(cvode_mem, tout, u, &t, CV_NORMAL);
207     if(check_flag(&flag, "CNode", 1)) break;
208     PrintOutput(cvode_mem, t, u);
209     if (sensi) {
210         flag = CNodeGetSens(cvode_mem, t, uS);
211         if(check_flag(&flag, "CNodeGetSens", 1)) break;
212         PrintOutputS(uS);
213     }
214     printf("-----\n");

```

```

215     }
216
217     /* Print final statistics */
218     PrintFinalStats(cvode_mem, sensi);
219
220     /* Free memory */
221     N_VDestroy_Serial(u);
222     if (sensi) {
223         N_VDestroyVectorArray_Serial(uS, NS);
224         free(plist);
225         free(pbar);
226     }
227     free(data);
228     CVodeFree(cvode_mem);
229
230     return(0);
231 }
232
233 /*
234  *-----
235  * FUNCTIONS CALLED BY CVODES
236  *-----
237  */
238
239 /*
240  * f routine. Compute f(t,u).
241  */
242
243 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
244 {
245     realtype ui, ult, urt, hordc, horac, hdiff, hadv;
246     realtype dx;
247     realtype *udata, *dudata;
248     int i;
249     UserData data;
250
251     udata = NV_DATA_S(u);
252     dudata = NV_DATA_S(udot);
253
254     /* Extract needed problem constants from data */
255     data = (UserData) f_data;
256     dx = data->dx;
257     hordc = data->p[0]/(dx*dx);
258     horac = data->p[1]/(RCONST(2.0)*dx);
259
260     /* Loop over all grid points. */
261     for (i=0; i<NEQ; i++) {
262
263         /* Extract u at x_i and two neighboring points */
264         ui = udata[i];
265         if (i!=0)
266             ult = udata[i-1];
267         else
268             ult = ZERO;

```

```

269     if(i!=NEQ-1)
270         urt = udata[i+1];
271     else
272         urt = ZERO;
273
274     /* Set diffusion and advection terms and load into udot */
275     hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
276     hadv = horac*(urt - ult);
277     dudata[i] = hdiff + hadv;
278 }
279 }
280
281 /*
282 *-----
283 * PRIVATE FUNCTIONS
284 *-----
285 */
286
287 /*
288 * Process and verify arguments to cvfnx.
289 */
290
291 static void ProcessArgs(int argc, char *argv[],
292                        booleantype *sensi, int *sensi_meth, booleantype *err_con)
293 {
294     *sensi = FALSE;
295     *sensi_meth = -1;
296     *err_con = FALSE;
297
298     if (argc < 2) WrongArgs(argv[0]);
299
300     if (strcmp(argv[1],"-nosensi") == 0)
301         *sensi = FALSE;
302     else if (strcmp(argv[1],"-sensi") == 0)
303         *sensi = TRUE;
304     else
305         WrongArgs(argv[0]);
306
307     if (*sensi) {
308
309         if (argc != 4)
310             WrongArgs(argv[0]);
311
312         if (strcmp(argv[2],"sim") == 0)
313             *sensi_meth = CV_SIMULTANEOUS;
314         else if (strcmp(argv[2],"stg") == 0)
315             *sensi_meth = CV_STAGGERED;
316         else if (strcmp(argv[2],"stg1") == 0)
317             *sensi_meth = CV_STAGGERED1;
318         else
319             WrongArgs(argv[0]);
320
321         if (strcmp(argv[3],"t") == 0)
322             *err_con = TRUE;

```



```

323     else if (strcmp(argv[3],"f") == 0)
324         *err_con = FALSE;
325     else
326         WrongArgs(argv[0]);
327 }
328
329 }
330
331 static void WrongArgs(char *name)
332 {
333     printf("\nUsage: %s [-nosensi] [-sensi sensi_meth err_con]\n",name);
334     printf("          sensi_meth = sim, stg, or stg1\n");
335     printf("          err_con      = t or f\n");
336
337     exit(0);
338 }
339
340 /*
341  * Set initial conditions in u vector.
342  */
343
344 static void SetIC(N_Vector u, realtype dx)
345 {
346     int i;
347     realtype x;
348     realtype *udata;
349
350     /* Set pointer to data array and get local length of u. */
351     udata = NV_DATA_S(u);
352
353     /* Load initial profile into u vector */
354     for (i=0; i<NEQ; i++) {
355         x = (i+1)*dx;
356         udata[i] = x*(XMAX - x)*exp(RCONST(2.0)*x);
357     }
358 }
359
360 /*
361  * Print current t, step count, order, stepsize, and max norm of solution
362  */
363
364 static void PrintOutput(void *cnode_mem, realtype t, N_Vector u)
365 {
366     long int nst;
367     int qu, flag;
368     realtype hu;
369
370     flag = CVodeGetNumSteps(cnode_mem, &nst);
371     check_flag(&flag, "CVodeGetNumSteps", 1);
372     flag = CVodeGetLastOrder(cnode_mem, &qu);
373     check_flag(&flag, "CVodeGetLastOrder", 1);
374     flag = CVodeGetLastStep(cnode_mem, &hu);
375     check_flag(&flag, "CVodeGetLastStep", 1);
376

```

```

377 #if defined(SUNDIALS_EXTENDED_PRECISION)
378     printf("%8.3Le %2d %8.3Le %5ld\n", t, qu, hu ,nst);
379 #elif defined(SUNDIALS_DOUBLE_PRECISION)
380     printf("%8.3le %2d %8.3le %5ld\n", t, qu, hu ,nst);
381 #else
382     printf("%8.3e %2d %8.3e %5ld\n", t, qu, hu ,nst);
383 #endif
384
385     printf("                                Solution                ");
386
387 #if defined(SUNDIALS_EXTENDED_PRECISION)
388     printf("%12.4Le \n", N_VMaxNorm(u));
389 #elif defined(SUNDIALS_DOUBLE_PRECISION)
390     printf("%12.4le \n", N_VMaxNorm(u));
391 #else
392     printf("%12.4e \n", N_VMaxNorm(u));
393 #endif
394 }
395
396 /*
397  * Print max norm of sensitivities
398  */
399
400 static void PrintOutputS(N_Vector *uS)
401 {
402     printf("                                Sensitivity 1  ");
403 #if defined(SUNDIALS_EXTENDED_PRECISION)
404     printf("%12.4Le \n", N_VMaxNorm(uS[0]));
405 #elif defined(SUNDIALS_DOUBLE_PRECISION)
406     printf("%12.4le \n", N_VMaxNorm(uS[0]));
407 #else
408     printf("%12.4e \n", N_VMaxNorm(uS[0]));
409 #endif
410
411     printf("                                Sensitivity 2  ");
412 #if defined(SUNDIALS_EXTENDED_PRECISION)
413     printf("%12.4Le \n", N_VMaxNorm(uS[1]));
414 #elif defined(SUNDIALS_DOUBLE_PRECISION)
415     printf("%12.4le \n", N_VMaxNorm(uS[1]));
416 #else
417     printf("%12.4e \n", N_VMaxNorm(uS[1]));
418 #endif
419 }
420
421
422 /*
423  * Print some final statistics located in the CVODES memory
424  */
425
426 static void PrintFinalStats(void *cvode_mem, booleantype sensi)
427 {
428     long int nst;
429     long int nfe, nsetups, nni, ncnf, netf;
430     long int nfSe, nfeS, nsetupsS, nniS, ncnfS, netfS;

```

```

431     int flag;
432
433     flag = CVodeGetNumSteps(cvode_mem, &nst);
434     check_flag(&flag, "CVodeGetNumSteps", 1);
435     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
436     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
437     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
438     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
439     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
440     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
441     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
442     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
443     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
444     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
445
446     if (sensi) {
447         flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
448         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
449         flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
450         check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
451         flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
452         check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
453         flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
454         check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);
455         flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
456         check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1);
457         flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
458         check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1);
459     }
460
461     printf("\nFinal Statistics\n\n");
462     printf("nst      = %5ld\n\n", nst);
463     printf("nfe      = %5ld\n", nfe);
464     printf("netf     = %5ld      nsetups = %5ld\n", netf, nsetups);
465     printf("nni      = %5ld      ncfn     = %5ld\n", nni, ncfn);
466
467     if(sensi) {
468         printf("\n");
469         printf("nfSe     = %5ld      nfeS      = %5ld\n", nfSe, nfeS);
470         printf("netfs    = %5ld      nsetupsS = %5ld\n", netfS, nsetupsS);
471         printf("nniS     = %5ld      ncfnS     = %5ld\n", nniS, ncfnS);
472     }
473
474 }
475
476 /*
477  * Check function return value...
478  *   opt == 0 means SUNDIALS function allocates memory so check if
479  *       returned NULL pointer
480  *   opt == 1 means SUNDIALS function returns a flag so check if
481  *       flag >= 0
482  *   opt == 2 means function allocates memory so check if returned
483  *       NULL pointer
484  */

```

```

485
486 static int check_flag(void *flagvalue, char *funcname, int opt)
487 {
488     int *errflag;
489
490     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
491     if (opt == 0 && flagvalue == NULL) {
492         fprintf(stderr,
493             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
494             funcname);
495         return(1); }
496
497     /* Check if flag < 0 */
498     else if (opt == 1) {
499         errflag = flagvalue;
500         if (*errflag < 0) {
501             fprintf(stderr,
502                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
503                 funcname, *errflag);
504             return(1); }}
505
506     /* Check if function returned NULL pointer - no memory allocated */
507     else if (opt == 2 && flagvalue == NULL) {
508         fprintf(stderr,
509             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
510             funcname);
511         return(1); }
512
513     return(0);
514 }

```

B Listing of cvfdx.c

```
1  /*
2  * -----
3  * $Revision: 1.21 $
4  * $Date: 2004/11/22 20:50:25 $
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 #include "sundialstypes.h" /* def. of type realtype */
52 #include "cvodes.h"       /* prototypes for CVODES functions and constants */
```

```

53 #include "cvdense.h"          /* prototype for CVDENSE functions and constants */
54 #include "nvector_serial.h"    /* defs. of serial NVECTOR functions and macros */
55 #include "dense.h"             /* defs. of type DenseMat, macro DENSE_ELEM */
56
57 /* Accessor macros */
58
59 #define Ith(v,i)    NV_Ith_S(v,i-1)      /* i-th vector component i=1..NEQ */
60 #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j=1..NEQ */
61
62 /* Problem Constants */
63
64 #define NEQ    3                /* number of equations */
65 #define Y1     RCONST(1.0)      /* initial y components */
66 #define Y2     RCONST(0.0)
67 #define Y3     RCONST(0.0)
68 #define RTOL   RCONST(1e-4)     /* scalar relative tolerance */
69 #define ATOL1  RCONST(1e-8)     /* vector absolute tolerance components */
70 #define ATOL2  RCONST(1e-14)
71 #define ATOL3  RCONST(1e-6)
72 #define T0     RCONST(0.0)      /* initial time */
73 #define T1     RCONST(0.4)      /* first output time */
74 #define TMULT  RCONST(10.0)     /* output time factor */
75 #define NOUT   12               /* number of output times */
76
77 #define NP     3                /* number of problem parameters */
78 #define NS     3                /* number of sensitivities computed */
79
80 #define ZERO   RCONST(0.0)
81
82 /* Type : UserData */
83
84 typedef struct {
85     realtype p[3];              /* problem parameters */
86 } *UserData;
87
88 /* Prototypes of functions by CVOIDS */
89
90 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
91
92 static void Jac(long int N, DenseMat J, realtype t,
93               N_Vector y, N_Vector fy, void *jac_data,
94               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
95
96 static void fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
97               int iS, N_Vector yS, N_Vector ySdot,
98               void *fS_data, N_Vector tmp1, N_Vector tmp2);
99
100 /* Prototypes of private functions */
101
102 static void ProcessArgs(int argc, char *argv[],
103                       boolean_type *sensi, int *sensi_meth,
104                       boolean_type *err_con);
105 static void WrongArgs(char *name);
106 static void PrintOutput(void *cvoid_mem, realtype t, N_Vector u);

```

```

107 static void PrintOutputS(N_Vector *uS);
108 static void PrintFinalStats(void *cnode_mem, booleantype sensi);
109 static int check_flag(void *flagvalue, char *funcname, int opt);
110
111 /*
112  *-----
113  * MAIN PROGRAM
114  *-----
115  */
116
117 int main(int argc, char *argv[])
118 {
119     void *cnode_mem;
120     UserData data;
121     realtype reltol, t, tout;
122     N_Vector y, abstol;
123     int iout, flag;
124
125     realtype pbar[NP];
126     int is, *plist;
127     N_Vector *yS;
128     booleantype sensi, err_con;
129     int sensi_meth;
130
131     cnode_mem = NULL;
132     data = NULL;
133     y = abstol = NULL;
134     yS = NULL;
135     plist = NULL;
136
137     /* Process arguments */
138     ProcessArgs(argc, argv, &sensi, &sensi_meth, &err_con);
139
140     /* User data structure */
141     data = (UserData) malloc(sizeof *data);
142     if (check_flag((void *)data, "malloc", 2)) return(1);
143     data->p[0] = RCONST(0.04);
144     data->p[1] = RCONST(1.0e4);
145     data->p[2] = RCONST(3.0e7);
146
147     /* Initial conditions */
148     y = N_VNew_Serial(NEQ);
149     if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
150
151     Ith(y,1) = Y1;
152     Ith(y,2) = Y2;
153     Ith(y,3) = Y3;
154
155     /* Tolerances:
156        scalar relative tolerance, vector absolute tolerance */
157     reltol = RTOL;
158
159     abstol = N_VNew_Serial(NEQ);
160     if (check_flag((void *)abstol, "N_VNew_Serial", 0)) return(1);

```

```

161 Ith(abstol,1) = ATOL1;
162 Ith(abstol,2) = ATOL2;
163 Ith(abstol,3) = ATOL3;
164
165 /* Create CVODES object */
166 cvode_mem = CNodeCreate(CV_BDF, CV_NEWTON);
167 if (check_flag((void *)cvode_mem, "CNodeCreate", 0)) return(1);
168
169 flag = CNodeSetFdata(cvode_mem, data);
170 if (check_flag(&flag, "CNodeSetFdata", 1)) return(1);
171
172 /* Allocate space for CVODES */
173 flag = CNodeMalloc(cvode_mem, f, T0, y, CV_SV, &reltol, abstol);
174 if (check_flag(&flag, "CNodeMalloc", 1)) return(1);
175
176 /* Attach linear solver */
177 flag = CVDense(cvode_mem, NEQ);
178 if (check_flag(&flag, "CVDense", 1)) return(1);
179
180 flag = CVDenseSetJacFn(cvode_mem, Jac);
181 if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
182
183 flag = CVDenseSetJacData(cvode_mem, data);
184 if (check_flag(&flag, "CVDenseSetJacData", 1)) return(1);
185
186 printf("\n3-species chemical kinetics problem\n");
187
188 /* Sensitivity-related settings */
189 if (sensi) {
190
191     pbar[0] = data->p[0];
192     pbar[1] = data->p[1];
193     pbar[2] = data->p[2];
194     plist = (int *) malloc(NS * sizeof(int));
195     if (check_flag((void *)plist, "malloc", 2)) return(1);
196     for (is=0; is<NS; is++) plist[is] = is+1;
197
198     yS = N_VNewVectorArray_Serial(NS, NEQ);
199     if (check_flag((void *)yS, "N_VNewVectorArray_Serial", 0)) return(1);
200     for (is=0; is<NS; is++)
201         N_VConst(ZERO, yS[is]);
202
203     flag = CNodeSetSensRhs1Fn(cvode_mem, fS);
204     if (check_flag(&flag, "CNodeSetSensRhs1Fn", 1)) return(1);
205     flag = CNodeSetSensErrCon(cvode_mem, err_con);
206     if (check_flag(&flag, "CNodeSetSensFdata", 1)) return(1);
207     flag = CNodeSetSensFdata(cvode_mem, data);
208     if (check_flag(&flag, "CNodeSetSensFdata", 1)) return(1);
209     flag = CNodeSetSensPbar(cvode_mem, pbar);
210     if (check_flag(&flag, "CNodeSetSensPbar", 1)) return(1);
211
212     flag = CNodeSensMalloc(cvode_mem, NS, sensi_meth, data->p, plist, yS);
213     if (check_flag(&flag, "CNodeSensMalloc", 1)) return(1);
214

```



```

215     printf("Sensitivity: YES ");
216     if(sensi_meth == CV_SIMULTANEOUS)
217         printf("( SIMULTANEOUS +");
218     else
219         if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
220         else printf("( STAGGERED1 +");
221     if(err_con) printf(" FULL ERROR CONTROL ");
222     else printf(" PARTIAL ERROR CONTROL ");
223
224 } else {
225
226     printf("Sensitivity: NO ");
227
228 }
229
230 /* In loop over output points, call CVode, print results, test for error */
231
232 printf("\n\n");
233 printf("=====");
234 printf("=====\\n");
235 printf("      T      Q      H      NST                      y1");
236 printf("              y2              y3      \\n");
237 printf("=====");
238 printf("=====\\n");
239
240 for (iout=1, tout=T1; iout <= NOUT; iout++, tout *= TMULT) {
241
242     flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
243     if (check_flag(&flag, "CVode", 1)) break;
244
245     PrintOutput(cvode_mem, t, y);
246
247     if (sensi) {
248         flag = CVodeGetSens(cvode_mem, t, yS);
249         if (check_flag(&flag, "CVodeGetSens", 1)) break;
250         PrintOutputS(yS);
251     }
252     printf("-----");
253     printf("-----\\n");
254
255 }
256
257 /* Print final statistics */
258 PrintFinalStats(cvode_mem, sensi);
259
260 /* Free memory */
261
262 N_VDestroy_Serial(y); /* Free y vector */
263 N_VDestroy_Serial(abstol); /* Free abstol vector */
264 if (sensi) {
265     N_VDestroyVectorArray_Serial(yS, NS); /* Free yS vector */
266     free(plist); /* Free plist */
267 }
268 free(data); /* Free user data */

```

```

269     CNodeFree(cvode_mem);                      /* Free CVODES memory */
270
271     return(0);
272 }
273
274 /*
275  *-----
276  * FUNCTIONS CALLED BY CVODES
277  *-----
278  */
279
280 /*
281  * f routine. Compute f(t,y).
282  */
283
284 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
285 {
286     realtype y1, y2, y3, yd1, yd3;
287     UserData data;
288     realtype p1, p2, p3;
289
290     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
291     data = (UserData) f_data;
292     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
293
294     yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
295     yd3 = Ith(ydot,3) = p3*y2*y2;
296     Ith(ydot,2) = -yd1 - yd3;
297 }
298
299
300 /*
301  * Jacobian routine. Compute J(t,y).
302  */
303
304 static void Jac(long int N, DenseMat J, realtype t,
305                N_Vector y, N_Vector fy, void *jac_data,
306                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
307 {
308     realtype y1, y2, y3;
309     UserData data;
310     realtype p1, p2, p3;
311
312     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
313     data = (UserData) jac_data;
314     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
315
316     IJth(J,1,1) = -p1;   IJth(J,1,2) = p2*y3;       IJth(J,1,3) = p2*y2;
317     IJth(J,2,1) =  p1;   IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
318     IJth(J,3,2) = 2*p3*y2;
319 }
320
321 /*
322  * fS routine. Compute sensitivity r.h.s.

```

```

323  */
324
325 static void fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
326               int iS, N_Vector yS, N_Vector ySdot,
327               void *fS_data, N_Vector tmp1, N_Vector tmp2)
328 {
329     UserData data;
330     realtype p1, p2, p3;
331     realtype y1, y2, y3;
332     realtype s1, s2, s3;
333     realtype sd1, sd2, sd3;
334
335     data = (UserData) fS_data;
336     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
337
338     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
339     s1 = Ith(yS,1); s2 = Ith(yS,2); s3 = Ith(yS,3);
340
341     sd1 = -p1*s1 + p2*y3*s2 + p2*y2*s3;
342     sd3 = 2*p3*y2*s2;
343     sd2 = -sd1-sd3;
344
345     switch (iS) {
346     case 0:
347         sd1 += -y1;
348         sd2 += y1;
349         break;
350     case 1:
351         sd1 += y2*y3;
352         sd2 += -y2*y3;
353         break;
354     case 2:
355         sd2 += -y2*y2;
356         sd3 += y2*y2;
357         break;
358     }
359
360     Ith(ySdot,1) = sd1;
361     Ith(ySdot,2) = sd2;
362     Ith(ySdot,3) = sd3;
363 }
364
365 /*
366  *-----
367  * PRIVATE FUNCTIONS
368  *-----
369  */
370
371 /*
372  * Process and verify arguments to cvfdx.
373  */
374
375 static void ProcessArgs(int argc, char *argv[],
376                         booleantype *sensi, int *sensi_meth, booleantype *err_con)

```

```

377 {
378     *sensi = FALSE;
379     *sensi_meth = -1;
380     *err_con = FALSE;
381
382     if (argc < 2) WrongArgs(argv[0]);
383
384     if (strcmp(argv[1], "-nosensi") == 0)
385         *sensi = FALSE;
386     else if (strcmp(argv[1], "-sensi") == 0)
387         *sensi = TRUE;
388     else
389         WrongArgs(argv[0]);
390
391     if (*sensi) {
392
393         if (argc != 4)
394             WrongArgs(argv[0]);
395
396         if (strcmp(argv[2], "sim") == 0)
397             *sensi_meth = CV_SIMULTANEOUS;
398         else if (strcmp(argv[2], "stg") == 0)
399             *sensi_meth = CV_STAGGERED;
400         else if (strcmp(argv[2], "stg1") == 0)
401             *sensi_meth = CV_STAGGERED1;
402         else
403             WrongArgs(argv[0]);
404
405         if (strcmp(argv[3], "t") == 0)
406             *err_con = TRUE;
407         else if (strcmp(argv[3], "f") == 0)
408             *err_con = FALSE;
409         else
410             WrongArgs(argv[0]);
411     }
412 }
413
414 static void WrongArgs(char *name)
415 {
416     printf("\nUsage: %s [-nosensi] [-sensi sensi_meth err_con]\n", name);
417     printf("          sensi_meth = sim, stg, or stg1\n");
418     printf("          err_con    = t or f\n");
419
420     exit(0);
421 }
422
423 /*
424  * Print current t, step count, order, stepsize, and solution.
425  */
426
427 static void PrintOutput(void *cnode_mem, realtype t, N_Vector u)
428 {
429     long int nst;
430

```

```

431     int qu, flag;
432     realtype hu, *udata;
433
434     udata = NV_DATA_S(u);
435
436     flag = CVodeGetNumSteps(cvode_mem, &nst);
437     check_flag(&flag, "CVodeGetNumSteps", 1);
438     flag = CVodeGetLastOrder(cvode_mem, &qu);
439     check_flag(&flag, "CVodeGetLastOrder", 1);
440     flag = CVodeGetLastStep(cvode_mem, &hu);
441     check_flag(&flag, "CVodeGetLastStep", 1);
442
443     #if defined(SUNDIALS_EXTENDED_PRECISION)
444         printf("%8.3Le %2d %8.3Le %5ld\n", t, qu, hu, nst);
445     #elif defined(SUNDIALS_DOUBLE_PRECISION)
446         printf("%8.3le %2d %8.3le %5ld\n", t, qu, hu, nst);
447     #else
448         printf("%8.3e %2d %8.3e %5ld\n", t, qu, hu, nst);
449     #endif
450
451     printf("                Solution                ");
452
453     #if defined(SUNDIALS_EXTENDED_PRECISION)
454         printf("%12.4Le %12.4Le %12.4Le \n", udata[0], udata[1], udata[2]);
455     #elif defined(SUNDIALS_DOUBLE_PRECISION)
456         printf("%12.4le %12.4le %12.4le \n", udata[0], udata[1], udata[2]);
457     #else
458         printf("%12.4e %12.4e %12.4e \n", udata[0], udata[1], udata[2]);
459     #endif
460
461 }
462
463 /*
464  * Print sensitivities.
465 */
466
467 static void PrintOutputS(N_Vector *uS)
468 {
469     realtype *sdata;
470
471     sdata = NV_DATA_S(uS[0]);
472     printf("                Sensitivity 1 ");
473
474     #if defined(SUNDIALS_EXTENDED_PRECISION)
475         printf("%12.4Le %12.4Le %12.4Le \n", sdata[0], sdata[1], sdata[2]);
476     #elif defined(SUNDIALS_DOUBLE_PRECISION)
477         printf("%12.4le %12.4le %12.4le \n", sdata[0], sdata[1], sdata[2]);
478     #else
479         printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
480     #endif
481
482     sdata = NV_DATA_S(uS[1]);
483     printf("                Sensitivity 2 ");
484

```

```

485 #if defined(SUNDIALS_EXTENDED_PRECISION)
486     printf("%12.4Le %12.4Le %12.4Le \n", sdata[0], sdata[1], sdata[2]);
487 #elif defined(SUNDIALS_DOUBLE_PRECISION)
488     printf("%12.4le %12.4le %12.4le \n", sdata[0], sdata[1], sdata[2]);
489 #else
490     printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
491 #endif
492
493     sdata = NV_DATA_S(uS[2]);
494     printf("                Sensitivity 3  ");
495
496 #if defined(SUNDIALS_EXTENDED_PRECISION)
497     printf("%12.4Le %12.4Le %12.4Le \n", sdata[0], sdata[1], sdata[2]);
498 #elif defined(SUNDIALS_DOUBLE_PRECISION)
499     printf("%12.4le %12.4le %12.4le \n", sdata[0], sdata[1], sdata[2]);
500 #else
501     printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
502 #endif
503 }
504
505 /*
506  * Print some final statistics from the CVODES memory.
507  */
508
509 static void PrintFinalStats(void *cvode_mem, booleantype sensi)
510 {
511     long int nst;
512     long int nfe, nsetups, nni, ncnf, netf;
513     long int nfSe, nfeS, nsetupsS, nniS, ncnfS, netfS;
514     long int njeD, nfeD;
515     int flag;
516
517     flag = CVodeGetNumSteps(cvode_mem, &nst);
518     check_flag(&flag, "CVodeGetNumSteps", 1);
519     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
520     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
521     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
522     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
523     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
524     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
525     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
526     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
527     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncnf);
528     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
529
530     if (sensi) {
531         flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
532         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
533         flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
534         check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
535         flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
536         check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
537         flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
538         check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);

```

```

539     flag = CNodeGetNumSensNonlinSolvIters(cvode_mem, &nnsS);
540     check_flag(&flag, "CNodeGetNumSensNonlinSolvIters", 1);
541     flag = CNodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
542     check_flag(&flag, "CNodeGetNumSensNonlinSolvConvFails", 1);
543 }
544
545 flag = CVDenseGetNumJacEvals(cvode_mem, &njeD);
546 check_flag(&flag, "CVDenseGetNumJacEvals", 1);
547 flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeD);
548 check_flag(&flag, "CVDenseGetNumRhsEvals", 1);
549
550 printf("\nFinal Statistics\n\n");
551 printf("nst      = %5ld\n\n", nst);
552 printf("nfe      = %5ld\n", nfe);
553 printf("netf      = %5ld      nsetups = %5ld\n", netf, nsetups);
554 printf("nni      = %5ld      ncfn      = %5ld\n", nni, ncfn);
555
556 if(sensi) {
557     printf("\n");
558     printf("nfSe      = %5ld      nfeS      = %5ld\n", nfSe, nfeS);
559     printf("netfs     = %5ld      nsetupsS = %5ld\n", netfS, nsetupsS);
560     printf("nniS      = %5ld      ncfnS      = %5ld\n", nniS, ncfnS);
561 }
562
563 printf("\n");
564 printf("njeD      = %5ld      nfeD      = %5ld\n", njeD, nfeD);
565
566 }
567
568 /*
569  * Check function return value.
570  *   opt == 0 means SUNDIALS function allocates memory so check if
571  *       returned NULL pointer
572  *   opt == 1 means SUNDIALS function returns a flag so check if
573  *       flag >= 0
574  *   opt == 2 means function allocates memory so check if returned
575  *       NULL pointer
576  */
577
578 static int check_flag(void *flagvalue, char *funcname, int opt)
579 {
580     int *errflag;
581
582     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
583     if (opt == 0 && flagvalue == NULL) {
584         fprintf(stderr,
585             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
586             funcname);
587         return(1); }
588
589     /* Check if flag < 0 */
590     else if (opt == 1) {
591         errflag = flagvalue;
592         if (*errflag < 0) {

```

```

593     fprintf(stderr,
594         "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
595         funcname, *errflag);
596     return(1); }}
597
598 /* Check if function returned NULL pointer - no memory allocated */
599 else if (opt == 2 && flagvalue == NULL) {
600     fprintf(stderr,
601         "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
602         funcname);
603     return(1); }
604
605 return(0);
606 }

```


C Listing of pvfkn.c

```

1  /*
2  * -----
3  * $Revision: 1.20 $
4  * $Date: 2004/11/22 20:50:33 $
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 pvfmx -nosensi
60 * If sensitivities are to be computed:
61 * % mpirun -np N pvfmx -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 #include "sundialstypes.h" /* def. of realtype */
72 #include "cvodes.h" /* main CVODES header file */
73 #include "iterative.h" /* types of preconditioning */
74 #include "cvspgmr.h" /* defs. for CVSPGMR functions and constants */
75 #include "smalldense.h" /* generic DENSE solver used in preconditioning */
76 #include "nvector_parallel.h" /* defs of parallel NVECTOR functions and macros */
77 #include "sundialsmath.h" /* contains SQR macro */
78 #include "mpi.h"
79
80
81 /* Problem Constants */
82
83 #define NVARs 2 /* number of species */
84 #define C1_SCALE RCONST(1.0e6) /* coefficients in initial profiles */
85 #define C2_SCALE RCONST(1.0e12)
86
87 #define TO RCONST(0.0) /* initial time */
88 #define NOUT 12 /* number of output times */
89 #define TWOHR RCONST(7200.0) /* number of seconds in two hours */
90 #define HALFDAY RCONST(4.32e4) /* number of seconds in a half day */
91 #define PI RCONST(3.1415926535898) /* pi */
92
93 #define XMIN RCONST(0.0) /* grid boundaries in x */
94 #define XMAX RCONST(20.0)
95 #define YMIN RCONST(30.0) /* grid boundaries in y */
96 #define YMAX RCONST(50.0)
97
98 #define NPEX 2 /* no. PEs in x direction of PE array */
99 #define NPEY 2 /* no. PEs in y direction of PE array */
100 /* Total no. PEs = NPEX*NPEY */
101 #define MXSUB 5 /* no. x points per subgrid */
102 #define MYSUB 5 /* no. y points per subgrid */
103
104 #define MX (NPEX*MXSUB) /* MX = number of x mesh points */
105 #define MY (NPEY*MYSUB) /* MY = number of y mesh points */
106 /* Spatial mesh is MX by MY */

```

```

107
108 /* CNodeMalloc Constants */
109
110 #define RTOL      RCONST(1.0e-5) /* scalar relative tolerance      */
111 #define FLOOR     RCONST(100.0) /* value of C1 or C2 at which tols. */
112                                     /* change from relative to absolute */
113 #define ATOL      (RTOL*FLOOR) /* scalar absolute tolerance      */
114
115 /* Sensitivity constants */
116 #define NP        8             /* number of problem parameters    */
117 #define NS        2             /* number of sensitivities          */
118
119 #define ZERO      RCONST(0.0)
120
121
122 /* User-defined matrix accessor macro: IJth */
123
124 /* IJth is defined in order to write code which indexes into small dense
125    matrices with a (row,column) pair, where 1 <= row,column <= NVARs.
126
127    IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
128    where 1 <= i,j <= NVARs. The small matrix routines in dense.h
129    work with matrices stored by column in a 2-dimensional array. In C,
130    arrays are indexed starting at 0, not 1. */
131
132 #define IJth(a,i,j)      (a[j-1][i-1])
133
134 /* Types : UserData and PreconData
135    contain problem parameters, problem constants, preconditioner blocks,
136    pivot arrays, grid constants, and processor indices */
137
138 typedef struct {
139     realtype *p;
140     realtype q4, om, dx, dy, hdco, haco, vdco;
141     realtype uext[NVARs*(MXSUB+2)*(MYSUB+2)];
142     long int my_pe, isubx, isuby, nvmsub, nvmsub2;
143     MPI_Comm comm;
144 } *UserData;
145
146 typedef struct {
147     void *f_data;
148     realtype **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
149     long int *pivot[MXSUB][MYSUB];
150 } *PreconData;
151
152
153 /* Functions Called by the CVODES Solver */
154
155 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
156
157 static int Precond(realtype tn, N_Vector u, N_Vector fu,
158                   booleantype jok, booleantype *jcurPtr,
159                   realtype gamma, void *P_data,
160                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);

```

```

161
162 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
163                 N_Vector r, N_Vector z,
164                 realtype gamma, realtype delta,
165                 int lr, void *P_data, N_Vector vtemp);
166
167 /* Private Helper Functions */
168
169 static void ProcessArgs(int argc, char *argv[], int my_pe,
170                       booleantype *sensi, int *sensi_meth, booleantype *err_con);
171 static void WrongArgs(int my_pe, char *name);
172
173 static PreconData AllocPreconData(UserData data);
174 static void FreePreconData(PreconData pdata);
175 static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
176 static void SetInitialProfiles(N_Vector u, UserData data);
177
178 static void BSend(MPI_Comm comm, int my_pe, long int isubx,
179                  long int isuby, long int dsize,
180                  long int dsizey, realtype udata[]);
181 static void BRecvPost(MPI_Comm comm, MPI_Request request[], int my_pe,
182                      long int isubx, long int isuby,
183                      long int dsize, long int dsizey,
184                      realtype uext[], realtype buffer[]);
185 static void BRecvWait(MPI_Request request[], long int isubx, long int isuby,
186                      long int dsize, realtype uext[], realtype buffer[]);
187 static void ucomm(realtype t, N_Vector u, UserData data);
188 static void fcalc(realtype t, realtype udata[], realtype dudata[], UserData data);
189
190 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
191                       realtype t, N_Vector u);
192 static void PrintOutputS(int my_pe, MPI_Comm comm, N_Vector *uS);
193 static void PrintFinalStats(void *cnode_mem, booleantype sensi);
194 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
195
196 /*
197  *-----
198  * MAIN PROGRAM
199  *-----
200  */
201
202 int main(int argc, char *argv[])
203 {
204     realtype abstol, reltol, t, tout;
205     N_Vector u;
206     UserData data;
207     PreconData pdata;
208     void *cnode_mem;
209     int iout, flag, my_pe, npes;
210     long int neq, local_N;
211     MPI_Comm comm;
212
213     realtype *pbar;
214     int is, *plist;

```

```

215 N_Vector *uS;
216 booleantype sensi, err_con;
217 int sensi_meth;
218
219 u = NULL;
220 data = NULL;
221 predata = NULL;
222 ccode_mem = NULL;
223 pbar = NULL;
224 plist = NULL;
225 uS = NULL;
226
227 /* Set problem size neq */
228 neq = Nvars*MX*MY;
229
230 /* Get processor number and total number of pe's */
231 MPI_Init(&argc, &argv);
232 comm = MPI_COMM_WORLD;
233 MPI_Comm_size(comm, &npes);
234 MPI_Comm_rank(comm, &my_pe);
235
236 if (npes != NPEX*NPEY) {
237     if (my_pe == 0)
238         fprintf(stderr,
239             "\nMPI_ERROR(0): npes = %d is not equal to NPEX*NPEY = %d\n\n",
240             npes, NPEX*NPEY);
241     MPI_Finalize();
242     return(1);
243 }
244
245 /* Process arguments */
246 ProcessArgs(argc, argv, my_pe, &sensi, &sensi_meth, &err_con);
247
248 /* Set local length */
249 local_N = Nvars*MXSUB*MYSUB;
250
251 /* Allocate and load user data block; allocate preconditioner block */
252 data = (UserData) malloc(sizeof *data);
253 data->p = NULL;
254 if (check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
255 data->p = (realttype *) malloc(NP*sizeof(realttype));
256 if (check_flag((void *)data->p, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
257 InitUserData(my_pe, comm, data);
258 predata = AllocPreconData (data);
259 if (check_flag((void *)predata, "AllocPreconData", 2, my_pe)) MPI_Abort(comm, 1);
260
261 /* Allocate u, and set initial values and tolerances */
262 u = N_VNew_Parallel(comm, local_N, neq);
263 if (check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
264 SetInitialProfiles(u, data);
265 abstol = ATOL; reltol = RTOL;
266
267 /* Create CVOIDS object, set optional input, allocate memory */
268 ccode_mem = CCodeCreate(CV_BDF, CV_NEWTON);

```

```

269 if (check_flag((void *)cnode_mem, "CNodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
270
271 flag = CNodeSetFdata(cnode_mem, data);
272 if (check_flag(&flag, "CNodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
273
274 flag = CNodeSetMaxNumSteps(cnode_mem, 2000);
275 if (check_flag(&flag, "CNodeSetMaxNumSteps", 1, my_pe)) MPI_Abort(comm, 1);
276
277 flag = CNodeMalloc(cnode_mem, f, T0, u, CV_SS, &reltol, &abstol);
278 if (check_flag(&flag, "CNodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
279
280 /* Attach linear solver CVSPGMR */
281 flag = CVSPgmr(cnode_mem, PREC_LEFT, 0);
282 if (check_flag(&flag, "CVSPgmr", 1, my_pe)) MPI_Abort(comm, 1);
283
284 flag = CVSPgmrSetPrecSetupFn(cnode_mem, Precond);
285 if (check_flag(&flag, "CVSPgmrSetPrecSetupFn", 1, my_pe)) MPI_Abort(comm, 1);
286
287 flag = CVSPgmrSetPrecSolveFn(cnode_mem, PSolve);
288 if (check_flag(&flag, "CVSPgmrSetPrecSolveFn", 1, my_pe)) MPI_Abort(comm, 1);
289
290 flag = CVSPgmrSetPrecData(cnode_mem, predata);
291 if (check_flag(&flag, "CVSPgmrSetPrecData", 1, my_pe)) MPI_Abort(comm, 1);
292
293 if(my_pe == 0)
294     printf("\n2-species diurnal advection-diffusion problem\n");
295
296 /* Sensitivity-related settings */
297 if( sensi) {
298
299     pbar = (realtype *) malloc(NP*sizeof(realtype));
300     if (check_flag((void *)pbar, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
301     for (is=0; is<NP; is++) pbar[is] = data->p[is];
302     plist = (int *) malloc(NS * sizeof(int));
303     if (check_flag((void *)plist, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
304     for (is=0; is<NS; is++) plist[is] = is+1;
305
306     uS = N_VNewVectorArray_Parallel(NS, comm, local_N, neq);
307     if (check_flag((void *)uS, "N_VNewVectorArray_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
308     for (is = 0; is < NS; is++)
309         N_VConst(ZERO,uS[is]);
310
311     flag = CNodeSetSensErrCon(cnode_mem, err_con);
312     if (check_flag(&flag, "CNodeSetSensErrCon", 1, my_pe)) MPI_Abort(comm, 1);
313
314     flag = CNodeSetSensRho(cnode_mem, ZERO);
315     if (check_flag(&flag, "CNodeSetSensRho", 1, my_pe)) MPI_Abort(comm, 1);
316
317     flag = CNodeSetSensPbar(cnode_mem, pbar);
318     if (check_flag(&flag, "CNodeSetSensPbar", 1, my_pe)) MPI_Abort(comm, 1);
319
320     flag = CNodeSensMalloc(cnode_mem, NS, sensi_meth, data->p, plist, uS);
321     if (check_flag(&flag, "CNodeSensMalloc", 1, my_pe)) MPI_Abort(comm, 1);
322

```

```

323     if(my_pe == 0) {
324         printf("Sensitivity: YES ");
325         if(sensi_meth == CV_SIMULTANEOUS)
326             printf("( SIMULTANEOUS +");
327         else
328             if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
329             else printf("( STAGGERED1 +");
330         if(err_con) printf(" FULL ERROR CONTROL ");
331         else printf(" PARTIAL ERROR CONTROL ");
332     }
333
334 } else {
335
336     if(my_pe == 0) printf("Sensitivity: NO ");
337
338 }
339
340 if (my_pe == 0) {
341     printf("\n\n");
342     printf("=====\n");
343     printf("      T      Q      H      NST      Bottom left  Top right \n");
344     printf("=====\n");
345 }
346
347 /* In loop over output points, call CVode, print results, test for error */
348 for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
349     flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
350     if (check_flag(&flag, "CVode", 1, my_pe)) break;
351     PrintOutput(cvode_mem, my_pe, comm, t, u);
352     if (sensi) {
353         flag = CVodeGetSens(cvode_mem, t, uS);
354         if (check_flag(&flag, "CVodeGetSens", 1, my_pe)) break;
355         PrintOutputS(my_pe, comm, uS);
356     }
357     if (my_pe == 0)
358         printf("-----\n");
359 }
360
361 /* Print final statistics */
362 if (my_pe == 0) PrintFinalStats(cvode_mem, sensi);
363
364 /* Free memory */
365 N_VDestroy_Parallel(u);
366 if (sensi) {
367     N_VDestroyVectorArray_Parallel(uS, NS);
368     free(plist);
369     free(pbar);
370 }
371 free(data->p);
372 free(data);
373 FreePreconData(predata);
374 CVodeFree(cvode_mem);
375
376 MPI_Finalize();

```

```

377
378     return(0);
379 }
380
381 /*
382 *-----
383 * FUNCTIONS CALLED BY CVOIDS
384 *-----
385 */
386
387 /*
388 * f routine. Evaluate f(t,y). First call ucomm to do communication of
389 * subgrid boundary data into uext. Then calculate f by a call to fcalc.
390 */
391
392 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
393 {
394     realtype *udata, *dudata;
395     UserData data;
396
397     udata = NV_DATA_P(u);
398     dudata = NV_DATA_P(udot);
399     data = (UserData) f_data;
400
401     /* Call ucomm to do inter-processor communicaiton */
402     ucomm (t, u, data);
403
404     /* Call fcalc to calculate all right-hand sides */
405     fcalc (t, udata, dudata, data);
406 }
407
408 /*
409 * Preconditioner setup routine. Generate and preprocess P.
410 */
411
412 static int Precond(realtype tn, N_Vector u, N_Vector fu,
413                   boolean_type jok, boolean_type *jcurPtr,
414                   realtype gamma, void *P_data,
415                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
416 {
417     realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
418     realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
419     int ier;
420     long int nvmxsub, *(*pivot)[MYSUB], offset;
421     int lx, ly, jx, jy, isubx, isuby;
422     realtype *udata, **a, **j;
423     PreconData predata;
424     UserData data;
425     realtype Q1, Q2, C3, A3, A4, KH, VEL, KV0;
426
427     /* Make local copies of pointers in P_data, pointer to u's data,
428        and PE index pair */
429     predata = (PreconData) P_data;
430     data = (UserData) (predata->f_data);

```



```

431 P = predata->P;
432 Jbd = predata->Jbd;
433 pivot = predata->pivot;
434 udata = NV_DATA_P(u);
435 isubx = data->isubx; isuby = data->isuby;
436 nvmxsub = data->nvmxsub;
437
438 /* Load problem coefficients and parameters */
439 Q1 = data->p[0];
440 Q2 = data->p[1];
441 C3 = data->p[2];
442 A3 = data->p[3];
443 A4 = data->p[4];
444 KH = data->p[5];
445 VEL = data->p[6];
446 KVO = data->p[7];
447
448 if (jok) { /* jok = TRUE: Copy Jbd to P */
449
450     for (ly = 0; ly < MYSUB; ly++)
451         for (lx = 0; lx < MXSUB; lx++)
452             dencopy(Jbd[lx][ly], P[lx][ly], NVARs);
453     *jcurPtr = FALSE;
454
455 } else { /* jok = FALSE: Generate Jbd from scratch and copy to P */
456
457     /* Make local copies of problem variables, for efficiency */
458     q4coef = data->q4;
459     dely = data->dy;
460     verdco = data->vdco;
461     hordco = data->hdco;
462
463     /* Compute 2x2 diagonal Jacobian blocks (using q4 values
464        computed on the last f call). Load into P. */
465     for (ly = 0; ly < MYSUB; ly++) {
466         jy = ly + isuby*MYSUB;
467         ydn = YMIN + (jy - RCONST(0.5))*dely;
468         yup = ydn + dely;
469         cydn = verdco*exp(RCONST(0.2)*ydn);
470         cyup = verdco*exp(RCONST(0.2)*yup);
471         diag = -(cydn + cyup + RCONST(2.0)*hordco);
472         for (lx = 0; lx < MXSUB; lx++) {
473             jx = lx + isubx*MXSUB;
474             offset = lx*NVARs + ly*nvmxsub;
475             c1 = udata[offset];
476             c2 = udata[offset+1];
477             j = Jbd[lx][ly];
478             a = P[lx][ly];
479             IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
480             IJth(j,1,2) = -Q2*c1 + q4coef;
481             IJth(j,2,1) = Q1*C3 - Q2*c2;
482             IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
483             dencopy(j, a, NVARs);
484         }

```

```

485     }
486
487     *jcurPtr = TRUE;
488
489 }
490
491 /* Scale by -gamma */
492 for (ly = 0; ly < MYSUB; ly++)
493     for (lx = 0; lx < MXSUB; lx++)
494         denscale(-gamma, P[lx][ly], NVARs);
495
496 /* Add identity matrix and do LU decompositions on blocks in place */
497 for (lx = 0; lx < MXSUB; lx++) {
498     for (ly = 0; ly < MYSUB; ly++) {
499         denaddI(P[lx][ly], NVARs);
500         ier = gefa(P[lx][ly], NVARs, pivot[lx][ly]);
501         if (ier != 0) return(1);
502     }
503 }
504
505 return(0);
506 }
507
508 /*
509  * Preconditioner solve routine
510  */
511
512 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
513                  N_Vector r, N_Vector z,
514                  realtype gamma, realtype delta,
515                  int lr, void *P_data, N_Vector vtemp)
516 {
517     realtype **(*P)[MYSUB];
518     long int nvmxsub, *(*pivot)[MYSUB];
519     int lx, ly;
520     realtype *zdata, *v;
521     PreconData predata;
522     UserData data;
523
524     /* Extract the P and pivot arrays from P_data */
525     predata = (PreconData) P_data;
526     data = (UserData) (predata->f_data);
527     P = predata->P;
528     pivot = predata->pivot;
529
530     /* Solve the block-diagonal system Px = r using LU factors stored
531        in P and pivot data in pivot, and return the solution in z.
532        First copy vector r to z. */
533     NV_Scale(RCONST(1.0), r, z);
534
535     nvmxsub = data->nvmxsub;
536     zdata = NV_DATA_P(z);
537
538     for (lx = 0; lx < MXSUB; lx++) {

```

```

539     for (ly = 0; ly < MYSUB; ly++) {
540         v = &(zdata[lx*NVARs + ly*nvmxsub]);
541         gesl(P[lx][ly], NVARs, pivot[lx][ly], v);
542     }
543 }
544
545     return(0);
546 }
547
548 /*
549 *-----
550 * PRIVATE FUNCTIONS
551 *-----
552 */
553
554 /*
555  * Process and verify arguments to pvfxx.
556  */
557
558 static void ProcessArgs(int argc, char *argv[], int my_pe,
559                         booleantype *sensi, int *sensi_meth, booleantype *err_con)
560 {
561     *sensi = FALSE;
562     *sensi_meth = -1;
563     *err_con = FALSE;
564
565     if (argc < 2) WrongArgs(my_pe, argv[0]);
566
567     if (strcmp(argv[1], "-nosensi") == 0)
568         *sensi = FALSE;
569     else if (strcmp(argv[1], "-sensi") == 0)
570         *sensi = TRUE;
571     else
572         WrongArgs(my_pe, argv[0]);
573
574     if (*sensi) {
575
576         if (argc != 4)
577             WrongArgs(my_pe, argv[0]);
578
579         if (strcmp(argv[2], "sim") == 0)
580             *sensi_meth = CV_SIMULTANEOUS;
581         else if (strcmp(argv[2], "stg") == 0)
582             *sensi_meth = CV_STAGGERED;
583         else if (strcmp(argv[2], "stg1") == 0)
584             *sensi_meth = CV_STAGGERED1;
585         else
586             WrongArgs(my_pe, argv[0]);
587
588         if (strcmp(argv[3], "t") == 0)
589             *err_con = TRUE;
590         else if (strcmp(argv[3], "f") == 0)
591             *err_con = FALSE;
592         else

```

```

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

```

```

647     }
648
649     free(pdata);
650 }
651
652 /*
653  * Set user data.
654  */
655
656 static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
657 {
658     long int isubx, isuby;
659     realtype KH, VEL, KV0;
660
661     /* Set problem parameters */
662     data->p[0] = RCONST(1.63e-16);      /* Q1 coeffs. q1, q2, c3 */
663     data->p[1] = RCONST(4.66e-16);      /* Q2 */
664     data->p[2] = RCONST(3.7e16);        /* C3 */
665     data->p[3] = RCONST(22.62);         /* A3 coeff. in expression for q3(t) */
666     data->p[4] = RCONST(7.601);        /* A4 coeff. in expression for q4(t) */
667     KH = data->p[5] = RCONST(4.0e-6);   /* KH horizontal diffusivity Kh */
668     VEL = data->p[6] = RCONST(0.001);   /* VEL advection velocity V */
669     KV0 = data->p[7] = RCONST(1.0e-8);  /* KV0 coeff. in Kv(z) */
670
671     /* Set problem constants */
672     data->om = PI/HALFDAY;
673     data->dx = (XMAX-XMIN)/((realtype)(MX-1));
674     data->dy = (YMAX-YMIN)/((realtype)(MY-1));
675     data->hdco = KH/SQR(data->dx);
676     data->haco = VEL/(RCONST(2.0)*data->dx);
677     data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;
678
679     /* Set machine-related constants */
680     data->comm = comm;
681     data->my_pe = my_pe;
682
683     /* isubx and isuby are the PE grid indices corresponding to my_pe */
684     isuby = my_pe/NPEX;
685     isubx = my_pe - isuby*NPEX;
686     data->isubx = isubx;
687     data->isuby = isuby;
688
689     /* Set the sizes of a boundary x-line in u and uext */
690     data->nvmxsub = NVAR*MXSUB;
691     data->nvmxsub2 = NVAR*(MXSUB+2);
692 }
693
694 /*
695  * Set initial conditions in u.
696  */
697
698 static void SetInitialProfiles(N_Vector u, UserData data)
699 {
700     long int isubx, isuby, lx, ly, jx, jy, offset;

```

```

701  realtype dx, dy, x, y, cx, cy, xmid, ymid;
702  realtype *udata;
703
704  /* Set pointer to data array in vector u */
705  udata = NV_DATA_P(u);
706
707  /* Get mesh spacings, and subgrid indices for this PE */
708  dx = data->dx;      dy = data->dy;
709  isubx = data->isubx; isuby = data->isuby;
710
711  /* Load initial profiles of c1 and c2 into local u vector.
712  Here lx and ly are local mesh point indices on the local subgrid,
713  and jx and jy are the global mesh point indices. */
714  offset = 0;
715  xmid = RCONST(0.5)*(XMIN + XMAX);
716  ymid = RCONST(0.5)*(YMIN + YMAX);
717  for (ly = 0; ly < MYSUB; ly++) {
718      jy = ly + isuby*MYSUB;
719      y = YMIN + jy*dy;
720      cy = SQR(RCONST(0.1)*(y - ymid));
721      cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
722      for (lx = 0; lx < MXSUB; lx++) {
723          jx = lx + isubx*MXSUB;
724          x = XMIN + jx*dx;
725          cx = SQR(RCONST(0.1)*(x - xmid));
726          cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
727          udata[offset] = C1_SCALE*cx*cy;
728          udata[offset+1] = C2_SCALE*cx*cy;
729          offset = offset + 2;
730      }
731  }
732 }
733
734 /*
735  * Routine to send boundary data to neighboring PEs.
736  */
737
738 static void BSend(MPI_Comm comm, int my_pe, long int isubx,
739                  long int isuby, long int dsizex, long int dsizey,
740                  realtype udata[])
741 {
742     int i, ly;
743     long int offsetu, offsetbuf;
744     realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
745
746     /* If isuby > 0, send data from bottom x-line of u */
747     if (isuby != 0)
748         MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
749
750     /* If isuby < NPEY-1, send data from top x-line of u */
751     if (isuby != NPEY-1) {
752         offsetu = (MYSUB-1)*dsizex;
753         MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
754     }

```

```

755
756 /* If isubx > 0, send data from left y-line of u (via bufleft) */
757 if (isubx != 0) {
758     for (ly = 0; ly < MYSUB; ly++) {
759         offsetbuf = ly*NVARs;
760         offsetu = ly*dsizex;
761         for (i = 0; i < NVARs; i++)
762             bufleft[offsetbuf+i] = udata[offsetu+i];
763     }
764     MPI_Send(&bufleft[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
765 }
766
767 /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
768 if (isubx != NPEX-1) {
769     for (ly = 0; ly < MYSUB; ly++) {
770         offsetbuf = ly*NVARs;
771         offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARs;
772         for (i = 0; i < NVARs; i++)
773             bufright[offsetbuf+i] = udata[offsetu+i];
774     }
775     MPI_Send(&bufright[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
776 }
777 }
778
779 /*
780  * Routine to start receiving boundary data from neighboring PEs.
781  * Notes:
782  * 1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
783  *    passed to both the BRecvPost and BRecvWait functions, and should not
784  *    be manipulated between the two calls.
785  * 2) request should have 4 entries, and should be passed in both calls also.
786  */
787
788 static void BRecvPost(MPI_Comm comm, MPI_Request request[], int my_pe,
789                     long int isubx, long int isuby,
790                     long int dsizex, long int dsizex,
791                     realtype uext[], realtype buffer[])
792 {
793     long int offsetue;
794
795     /* Have bufleft and bufright use the same buffer */
796     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
797
798     /* If isuby > 0, receive data for bottom x-line of uext */
799     if (isuby != 0)
800         MPI_Irecv(&uext[NVARs], dsizex, PVEC_REAL_MPI_TYPE,
801                 my_pe-NPEX, 0, comm, &request[0]);
802
803     /* If isuby < NPEY-1, receive data for top x-line of uext */
804     if (isuby != NPEY-1) {
805         offsetue = NVARs*(1 + (MYSUB+1)*(MXSUB+2));
806         MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
807                 my_pe+NPEX, 0, comm, &request[1]);
808     }

```

```

809
810 /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
811 if (isubx != 0) {
812     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
813             my_pe-1, 0, comm, &request[2]);
814 }
815
816 /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
817 if (isubx != NPEX-1) {
818     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
819             my_pe+1, 0, comm, &request[3]);
820 }
821 }
822
823 /*
824  * Routine to finish receiving boundary data from neighboring PEs.
825  * Notes:
826  * 1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
827  *    passed to both the BRecvPost and BRecvWait functions, and should not
828  *    be manipulated between the two calls.
829  * 2) request should have 4 entries, and should be passed in both calls also.
830  */
831
832 static void BRecvWait(MPI_Request request[], long int isubx, long int isuby,
833                     long int dsizey, realtype uext[], realtype buffer[])
834 {
835     int i, ly;
836     long int dsizey2, offsetue, offsetbuf;
837     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
838     MPI_Status status;
839
840     dsizey2 = dsizey + 2*NVARs;
841
842     /* If isuby > 0, receive data for bottom x-line of uext */
843     if (isuby != 0)
844         MPI_Wait(&request[0], &status);
845
846     /* If isuby < NPEY-1, receive data for top x-line of uext */
847     if (isuby != NPEY-1)
848         MPI_Wait(&request[1], &status);
849
850     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
851     if (isubx != 0) {
852         MPI_Wait(&request[2], &status);
853
854         /* Copy the buffer to uext */
855         for (ly = 0; ly < MYSUB; ly++) {
856             offsetbuf = ly*NVARs;
857             offsetue = (ly+1)*dsizey2;
858             for (i = 0; i < NVARs; i++)
859                 uext[offsetue+i] = bufleft[offsetbuf+i];
860         }
861     }
862 }

```



```

863  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
864  if (isubx != NPEX-1) {
865      MPI_Wait(&request[3], &status);
866
867      /* Copy the buffer to uext */
868      for (ly = 0; ly < MYSUB; ly++) {
869          offsetbuf = ly*NVAR;
870          offsetue = (ly+2)*dsizex2 - NVAR;
871          for (i = 0; i < NVAR; i++)
872              uext[offsetue+i] = bufright[offsetbuf+i];
873      }
874  }
875
876  }
877
878  /*
879   * ucomm routine. This routine performs all communication
880   * between processors of data needed to calculate f.
881   */
882
883  static void ucomm(realtype t, N_Vector u, UserData data)
884  {
885      realtype *udata, *uext, buffer[2*NVAR*MYSUB];
886      MPI_Comm comm;
887      int my_pe;
888      long int isubx, isuby, nvxsub, nvmysub;
889      MPI_Request request[4];
890
891      udata = NV_DATA_P(u);
892
893      /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
894      comm = data->comm; my_pe = data->my_pe;
895      isubx = data->isubx; isuby = data->isuby;
896      nvxsub = data->nvxsub;
897      nvmysub = NVAR*MYSUB;
898      uext = data->uext;
899
900      /* Start receiving boundary data from neighboring PEs */
901      BRecvPost(comm, request, my_pe, isubx, isuby, nvxsub, nvmysub, uext, buffer);
902
903      /* Send data from boundary of local grid to neighboring PEs */
904      BSend(comm, my_pe, isubx, isuby, nvxsub, nvmysub, udata);
905
906      /* Finish receiving boundary data from neighboring PEs */
907      BRecvWait(request, isubx, isuby, nvxsub, uext, buffer);
908  }
909
910  /*
911   * fcalc routine. Compute f(t,y). This routine assumes that communication
912   * between processors of data needed to calculate f has already been done,
913   * and this data is in the work array uext.
914   */
915
916  static void fcalc(realtype t, realtype udata[], realtype dudata[], UserData data)

```

```

917 {
918     realtype *uext;
919     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
920     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
921     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
922     realtype q4coef, dely, verdco, hordco, horaco;
923     int i, lx, ly, jx, jy;
924     long int isubx, isuby, nvmxsub, nvmxsub2, offsetu, offsetue;
925     realtype Q1, Q2, C3, A3, A4, KH, VEL, KVO;
926
927     /* Get subgrid indices, data sizes, extended work array uext */
928     isubx = data->isubx;    isuby = data->isuby;
929     nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
930     uext = data->uext;
931
932     /* Load problem coefficients and parameters */
933     Q1 = data->p[0];
934     Q2 = data->p[1];
935     C3 = data->p[2];
936     A3 = data->p[3];
937     A4 = data->p[4];
938     KH = data->p[5];
939     VEL = data->p[6];
940     KVO = data->p[7];
941
942     /* Copy local segment of u vector into the working extended array uext */
943     offsetu = 0;
944     offsetue = nvmxsub2 + NVARs;
945     for (ly = 0; ly < MYSUB; ly++) {
946         for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
947         offsetu = offsetu + nvmxsub;
948         offsetue = offsetue + nvmxsub2;
949     }
950
951     /* To facilitate homogeneous Neumann boundary conditions, when this is
952     a boundary PE, copy data from the first interior mesh line of u to uext */
953
954     /* If isuby = 0, copy x-line 2 of u to uext */
955     if (isuby == 0) {
956         for (i = 0; i < nvmxsub; i++) uext[NVARs+i] = udata[nvmxsub+i];
957     }
958
959     /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
960     if (isuby == NPEY-1) {
961         offsetu = (MYSUB-2)*nvmxsub;
962         offsetue = (MYSUB+1)*nvmxsub2 + NVARs;
963         for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
964     }
965
966     /* If isubx = 0, copy y-line 2 of u to uext */
967     if (isubx == 0) {
968         for (ly = 0; ly < MYSUB; ly++) {
969             offsetu = ly*nvmxsub + NVARs;
970             offsetue = (ly+1)*nvmxsub2;

```

```

971     for (i = 0; i < NVARs; i++) uext[offsetue+i] = udata[offsetu+i];
972 }
973 }
974
975 /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
976 if (isubx == NPEX-1) {
977     for (ly = 0; ly < MYSUB; ly++) {
978         offsetu = (ly+1)*nvmxsub - 2*NVARs;
979         offsetue = (ly+2)*nvmxsub2 - NVARs;
980         for (i = 0; i < NVARs; i++) uext[offsetue+i] = udata[offsetu+i];
981     }
982 }
983
984 /* Make local copies of problem variables, for efficiency */
985 dely = data->dy;
986 verdco = data->vdco;
987 hordco = data->hdco;
988 horaco = data->haco;
989
990 /* Set diurnal rate coefficients as functions of t, and save q4 in
991 data block for use by preconditioner evaluation routine */
992 s = sin((data->om)*t);
993 if (s > ZERO) {
994     q3 = exp(-A3/s);
995     q4coef = exp(-A4/s);
996 } else {
997     q3 = ZERO;
998     q4coef = ZERO;
999 }
1000 data->q4 = q4coef;
1001
1002 /* Loop over all grid points in local subgrid */
1003 for (ly = 0; ly < MYSUB; ly++) {
1004     jy = ly + isuby*MYSUB;
1005
1006     /* Set vertical diffusion coefficients at jy +/- 1/2 */
1007     ydn = YMIN + (jy - .5)*dely;
1008     yup = ydn + dely;
1009     cydn = verdco*exp(RCONST(0.2)*ydn);
1010     cyup = verdco*exp(RCONST(0.2)*yup);
1011     for (lx = 0; lx < MXSUB; lx++) {
1012         jx = lx + isubx*MXSUB;
1013
1014         /* Extract c1 and c2, and set kinetic rate terms */
1015         offsetue = (lx+1)*NVARs + (ly+1)*nvmxsub2;
1016         c1 = uext[offsetue];
1017         c2 = uext[offsetue+1];
1018         qq1 = Q1*c1*C3;
1019         qq2 = Q2*c1*c2;
1020         qq3 = q3*C3;
1021         qq4 = q4coef*c2;
1022         rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
1023         rkin2 = qq1 - qq2 - qq4;
1024

```

```

1025     /* Set vertical diffusion terms */
1026     c1dn = uext[offsetue-nvmxsub2];
1027     c2dn = uext[offsetue-nvmxsub2+1];
1028     c1up = uext[offsetue+nvmxsub2];
1029     c2up = uext[offsetue+nvmxsub2+1];
1030     vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
1031     vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
1032
1033     /* Set horizontal diffusion and advection terms */
1034     c1lt = uext[offsetue-2];
1035     c2lt = uext[offsetue-1];
1036     c1rt = uext[offsetue+2];
1037     c2rt = uext[offsetue+3];
1038     hord1 = hordco*(c1rt - 2.0*c1 + c1lt);
1039     hord2 = hordco*(c2rt - 2.0*c2 + c2lt);
1040     horad1 = horaco*(c1rt - c1lt);
1041     horad2 = horaco*(c2rt - c2lt);
1042
1043     /* Load all terms into dudata */
1044     offsetu = lx*NVARs + ly*nvmxsub;
1045     dudata[offsetu] = vertd1 + hord1 + horad1 + rkin1;
1046     dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
1047 }
1048 }
1049
1050 }
1051
1052 /*
1053  * Print current t, step count, order, stepsize, and sampled c1,c2 values.
1054  */
1055
1056 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
1057                        realtype t, N_Vector u)
1058 {
1059     long int nst;
1060     int qu, flag;
1061     realtype hu, *udata, tempu[2];
1062     long int npelast, i0, i1;
1063     MPI_Status status;
1064
1065     npelast = NPEX*NPEY - 1;
1066     udata = NV_DATA_P(u);
1067
1068     /* Send c at top right mesh point to PE 0 */
1069     if (my_pe == npelast) {
1070         i0 = NVARs*MXSUB*MYSUB - 2;
1071         i1 = i0 + 1;
1072         if (npelast != 0)
1073             MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
1074         else {
1075             tempu[0] = udata[i0];
1076             tempu[1] = udata[i1];
1077         }
1078     }

```

```

1079
1080 /* On PE 0, receive c at top right, then print performance data
1081    and sampled solution values */
1082 if (my_pe == 0) {
1083
1084     if (npelast != 0)
1085         MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
1086
1087     flag = CNodeGetNumSteps(cvode_mem, &nst);
1088     check_flag(&flag, "CNodeGetNumSteps", 1, my_pe);
1089     flag = CNodeGetLastOrder(cvode_mem, &qu);
1090     check_flag(&flag, "CNodeGetLastOrder", 1, my_pe);
1091     flag = CNodeGetLastStep(cvode_mem, &hu);
1092     check_flag(&flag, "CNodeGetLastStep", 1, my_pe);
1093
1094     #if defined(SUNDIALS_EXTENDED_PRECISION)
1095         printf("%8.3Le %2d %8.3Le %5ld\n", t, qu, hu, nst);
1096     #elif defined(SUNDIALS_DOUBLE_PRECISION)
1097         printf("%8.3le %2d %8.3le %5ld\n", t, qu, hu, nst);
1098     #else
1099         printf("%8.3e %2d %8.3e %5ld\n", t, qu, hu, nst);
1100     #endif
1101
1102     printf("                                Solution                ");
1103     #if defined(SUNDIALS_EXTENDED_PRECISION)
1104         printf("%12.4Le %12.4Le \n", udata[0], tempu[0]);
1105     #elif defined(SUNDIALS_DOUBLE_PRECISION)
1106         printf("%12.4le %12.4le \n", udata[0], tempu[0]);
1107     #else
1108         printf("%12.4e %12.4e \n", udata[0], tempu[0]);
1109     #endif
1110
1111     printf("                                ");
1112
1113     #if defined(SUNDIALS_EXTENDED_PRECISION)
1114         printf("%12.4Le %12.4Le \n", udata[1], tempu[1]);
1115     #elif defined(SUNDIALS_DOUBLE_PRECISION)
1116         printf("%12.4le %12.4le \n", udata[1], tempu[1]);
1117     #else
1118         printf("%12.4e %12.4e \n", udata[1], tempu[1]);
1119     #endif
1120
1121 }
1122
1123 }
1124
1125 /*
1126  * Print sampled sensitivity values.
1127  */
1128
1129 static void PrintOutputS(int my_pe, MPI_Comm comm, N_Vector *uS)
1130 {
1131     realtype *sdata, temps[2];
1132     long int npelast, i0, i1;

```

```

1133 MPI_Status status;
1134
1135 npelast = NPEX*NPEY - 1;
1136
1137 sdata = NV_DATA_P(uS[0]);
1138
1139 /* Send s1 at top right mesh point to PE 0 */
1140 if (my_pe == npelast) {
1141     i0 = NVAR*MXSUB*MYSUB - 2;
1142     i1 = i0 + 1;
1143     if (npelast != 0)
1144         MPI_Send(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
1145     else {
1146         temps[0] = sdata[i0];
1147         temps[1] = sdata[i1];
1148     }
1149 }
1150
1151 /* On PE 0, receive s1 at top right, then print sampled sensitivity values */
1152 if (my_pe == 0) {
1153     if (npelast != 0)
1154         MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
1155     printf("-----\n");
1156     printf("Sensitivity 1 ");
1157 #if defined(SUNDIALS_EXTENDED_PRECISION)
1158     printf("%12.4Le %12.4Le \n", sdata[0], temps[0]);
1159 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1160     printf("%12.4le %12.4le \n", sdata[0], temps[0]);
1161 #else
1162     printf("%12.4e %12.4e \n", sdata[0], temps[0]);
1163 #endif
1164     printf(" ");
1165 #if defined(SUNDIALS_EXTENDED_PRECISION)
1166     printf("%12.4Le %12.4Le \n", sdata[1], temps[1]);
1167 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1168     printf("%12.4le %12.4le \n", sdata[1], temps[1]);
1169 #else
1170     printf("%12.4e %12.4e \n", sdata[1], temps[1]);
1171 #endif
1172 }
1173
1174 sdata = NV_DATA_P(uS[1]);
1175
1176 /* Send s2 at top right mesh point to PE 0 */
1177 if (my_pe == npelast) {
1178     i0 = NVAR*MXSUB*MYSUB - 2;
1179     i1 = i0 + 1;
1180     if (npelast != 0)
1181         MPI_Send(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
1182     else {
1183         temps[0] = sdata[i0];
1184         temps[1] = sdata[i1];
1185     }
1186 }

```

```

1187
1188 /* On PE 0, receive s2 at top right, then print sampled sensitivity values */
1189 if (my_pe == 0) {
1190     if (npelast != 0)
1191         MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
1192     printf("-----\n");
1193     printf("Sensitivity 2 ");
1194 #if defined(SUNDIALS_EXTENDED_PRECISION)
1195     printf("%12.4Le %12.4Le \n", sdata[0], temps[0]);
1196 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1197     printf("%12.4le %12.4le \n", sdata[0], temps[0]);
1198 #else
1199     printf("%12.4e %12.4e \n", sdata[0], temps[0]);
1200 #endif
1201     printf(" ");
1202 #if defined(SUNDIALS_EXTENDED_PRECISION)
1203     printf("%12.4Le %12.4Le \n", sdata[1], temps[1]);
1204 #elif defined(SUNDIALS_DOUBLE_PRECISION)
1205     printf("%12.4le %12.4le \n", sdata[1], temps[1]);
1206 #else
1207     printf("%12.4e %12.4e \n", sdata[1], temps[1]);
1208 #endif
1209 }
1210 }
1211
1212 /*
1213  * Print final statistics from the CVODES memory.
1214  */
1215
1216 static void PrintFinalStats(void *cvode_mem, booleantype sensi)
1217 {
1218     long int nst;
1219     long int nfe, nsetups, nni, ncfn, netf;
1220     long int nfSe, nfeS, nsetupsS, nniS, ncfnS, netfS;
1221     int flag;
1222
1223     flag = CVodeGetNumSteps(cvode_mem, &nst);
1224     check_flag(&flag, "CVodeGetNumSteps", 1, 0);
1225     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
1226     check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
1227     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
1228     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
1229     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
1230     check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
1231     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
1232     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
1233     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
1234     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
1235
1236     if (sensi) {
1237         flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
1238         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1, 0);
1239         flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
1240         check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1, 0);

```

```

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

```



```
1295
1296  /* Check if function returned NULL pointer - no memory allocated */
1297  else if (opt == 2 && flagvalue == NULL) {
1298      fprintf(stderr,
1299          "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
1300          id, funcname);
1301      return(1); }
1302
1303  return(0);
1304 }
```

D listing of cvadx.c

```

1  /*
2  * -----
3  * $Revision: 1.18 $
4  * $Date: 2004/11/15 19:01:35 $
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 sundials/cvodes/LICENSE.
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  #include "cvodes.h"
58  #include "cvodea.h"
59  #include "cvsdense.h"
60  #include "nvector_serial.h"
61  #include "sundialstypes.h"
62
63  /* Accessor macros */
64
65  #define Ith(v,i)    NV_Ith_S(v,i-1)      /* i-th vector component i= 1..NEQ */
66  #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j = 1..NEQ */
67
68  /* Problem Constants */
69
70  #define NEQ        3          /* number of equations */
71
72  #define RTOL       RCONST(1e-4) /* scalar relative tolerance */
73
74  #define ATOL1      RCONST(1e-8) /* vector absolute tolerance components */
75  #define ATOL2      RCONST(1e-14)
76  #define ATOL3      RCONST(1e-6)
77
78  #define ATOLL      RCONST(1e-5) /* absolute tolerance for adjoint vars. */
79  #define ATOLq      RCONST(1e-6) /* absolute tolerance for quadratures */
80
81  #define T0         RCONST(0.0) /* initial time */
82  #define TOUT       RCONST(4e7) /* final time */
83
84  #define TB1        RCONST(4e7) /* starting point for adjoint problem */
85  #define TB2        RCONST(50.0) /* starting point for adjoint problem */
86
87  #define STEPS      150          /* number of steps between check points */
88
89  #define NP         3          /* number of problem parameters */
90
91  #define ZERO       RCONST(0.0)
92
93
94  /* Type : UserData */
95
96  typedef struct {
97      realtype p[3];
98  } *UserData;
99
100 /* Prototypes of user-supplied functions */
101
102 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
103 static void Jac(long int N, DenseMat J, realtype t,
104               N_Vector y, N_Vector fy, void *jac_data,
105               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
106 static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data);

```

```

107
108 static void fB(realtype t, N_Vector y,
109               N_Vector yB, N_Vector yBdot, void *f_dataB);
110 static void JacB(long int NB, DenseMat JB, realtype t,
111                 N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
112                 N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B);
113 static void fQB(realtype t, N_Vector y, N_Vector yB,
114                 N_Vector qBdot, void *fQ_dataB);
115
116
117 /* Prototypes of private functions */
118
119 static void PrintOutput(N_Vector yB, N_Vector qB);
120 static int check_flag(void *flagvalue, char *funcname, int opt);
121
122 /*
123  *-----
124  * MAIN PROGRAM
125  *-----
126  */
127
128 int main(int argc, char *argv[])
129 {
130     UserData data;
131
132     void *cvadj_mem;
133     void *ccode_mem;
134
135     realtype reltol, abstolQ;
136     N_Vector y, q, abstol;
137
138     realtype reltolB, abstolB, abstolQB;
139     N_Vector yB, qB;
140
141     realtype time;
142     int flag, ncheck;
143
144     data = NULL;
145     cvadj_mem = ccode_mem = NULL;
146     y = abstol = yB = qB = NULL;
147
148     /* Print problem description */
149     printf("\n\n Adjoint Sensitivity Example for Chemical Kinetics\n");
150     printf(" ----- \n\n");
151     printf("ODE: dy1/dt = -p1*y1 + p2*y2*y3\n");
152     printf("      dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2\n");
153     printf("      dy3/dt =  p3*(y2)^2\n");
154     printf("Find dG/dp for\n");
155     printf("      G = int_t0^tB0 g(t,p,y) dt\n");
156     printf("      g(t,p,y) = y3\n\n\n");
157
158
159     /* User data structure */
160     data = (UserData) malloc(sizeof *data);

```

```

161     if (check_flag((void *)data, "malloc", 2)) return(1);
162     data->p[0] = RCONST(0.04);
163     data->p[1] = RCONST(1.0e4);
164     data->p[2] = RCONST(3.0e7);
165
166     /* Initialize y */
167     y = N_VNew_Serial(NEQ);
168     if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
169     Ith(y,1) = RCONST(1.0);
170     Ith(y,2) = ZERO;
171     Ith(y,3) = ZERO;
172
173     /* Initialize q */
174     q = N_VNew_Serial(1);
175     if (check_flag((void *)q, "N_VNew_Serial", 0)) return(1);
176     Ith(q,1) = ZERO;
177
178     /* Set the scalar relative tolerance reltol */
179     reltol = RTOL;
180
181     /* Set the vector absolute tolerance abstol */
182     abstol = N_VNew_Serial(NEQ);
183     if (check_flag((void *)abstol, "N_VNew_Serial", 0)) return(1);
184     Ith(abstol,1) = ATOL1;
185     Ith(abstol,2) = ATOL2;
186     Ith(abstol,3) = ATOL3;
187
188     /* Set the scalar absolute tolerance abstolQ */
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     ccode_mem = CCodeCreate(CV_BDF, CV_NEWTON);
195     if (check_flag((void *)ccode_mem, "CCodeCreate", 0)) return(1);
196
197     flag = CCodeSetFdata(ccode_mem, data);
198     if (check_flag(&flag, "CCodeSetFdata", 1)) return(1);
199
200     flag = CCodeMalloc(ccode_mem, f, T0, y, CV_SV, &reltol, abstol);
201     if (check_flag(&flag, "CCodeMalloc", 1)) return(1);
202
203     flag = CVDense(ccode_mem, NEQ);
204     if (check_flag(&flag, "CVDense", 1)) return(1);
205     flag = CVDenseSetJacFn(ccode_mem, Jac);
206     if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
207     flag = CVDenseSetJacData(ccode_mem, data);
208     if (check_flag(&flag, "CVDenseSetJacData", 1)) return(1);
209
210     flag = CCodeSetQuadFdata(ccode_mem, data);
211     if (check_flag(&flag, "CCodeSetQuadFdata", 1)) return(1);
212     flag = CCodeSetQuadErrCon(ccode_mem, TRUE);
213     if (check_flag(&flag, "CCodeSetQuadErrCon", 1)) return(1);
214     flag = CCodeSetQuadTolerances(ccode_mem, CV_SS, &reltol, &abstolQ);

```

```

215     if (check_flag(&flag, "CVodeSetQuadTolerances", 1)) return(1);
216     flag = CVodeQuadMalloc(cvode_mem, fQ, q);
217     if (check_flag(&flag, "CVodeQuadMalloc", 1)) return(1);
218
219     /* Allocate global memory */
220     printf("Allocate global memory\n");
221     cvadj_mem = CVadjMalloc(cvode_mem, STEPS);
222     if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0)) return(1);
223
224     /* Perform forward run */
225     printf("Forward integration ... ");
226
227     flag = CVodeF(cvadj_mem, TOUT, y, &time, CV_NORMAL, &ncheck);
228     if (check_flag(&flag, "CVodeF", 1)) return(1);
229
230     flag = CVodeGetQuad(cvode_mem, TOUT, q);
231     if (check_flag(&flag, "CVodeGetQuad", 1)) return(1);
232
233     #if defined(SUNDIALS_EXTENDED_PRECISION)
234         printf("G: %12.4Le \n", Ith(q,1));
235     #elif defined(SUNDIALS_DOUBLE_PRECISION)
236         printf("G: %12.4le \n", Ith(q,1));
237     #else
238         printf("G: %12.4e \n", Ith(q,1));
239     #endif
240
241     /* Test check point linked list */
242     printf("\nList of Check Points (ncheck = %d)\n", ncheck);
243     CVadjGetCheckPointsList(cvadj_mem);
244
245     /* Initialize yB */
246     yB = N_VNew_Serial(NEQ);
247     if (check_flag((void *)yB, "N_VNew_Serial", 0)) return(1);
248     Ith(yB,1) = ZERO;
249     Ith(yB,2) = ZERO;
250     Ith(yB,3) = ZERO;
251
252     /* Initialize qB */
253     qB = N_VNew_Serial(NP);
254     if (check_flag((void *)qB, "N_VNew", 0)) return(1);
255     Ith(qB,1) = ZERO;
256     Ith(qB,2) = ZERO;
257     Ith(qB,3) = ZERO;
258
259     /* Set the scalar relative tolerance reltolB */
260     reltolB = RTOL;
261
262     /* Set the scalar absolute tolerance abstolB */
263     abstolB = ATOLl;
264
265     /* Set the scalar absolute tolerance abstolQB */
266     abstolQB = ATOLq;
267
268     /* Create and allocate CVODES memory for backward run */

```

```

269 printf("\nCreate and allocate CVODES memory for backward run\n");
270 flag = CVodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
271 if (check_flag(&flag, "CVodeCreateB", 1)) return(1);
272 flag = CVodeSetFdataB(cvadj_mem, data);
273 if (check_flag(&flag, "CVodeSetFdataB", 1)) return(1);
274 flag = CVodeMallocB(cvadj_mem, fB, TB1, yB, CV_SS, &reltolB, &abstolB);
275 if (check_flag(&flag, "CVodeMallocB", 1)) return(1);
276
277 flag = CVDenseB(cvadj_mem, NEQ);
278 if (check_flag(&flag, "CVDenseB", 1)) return(1);
279 flag = CVDenseSetJacFnB(cvadj_mem, JacB);
280 if (check_flag(&flag, "CVDenseSetJacFnB", 1)) return(1);
281 flag = CVDenseSetJacDataB(cvadj_mem, data);
282 if (check_flag(&flag, "CVDenseSetJacDataB", 1)) return(1);
283
284 flag = CVodeSetQuadFdataB(cvadj_mem, data);
285 if (check_flag(&flag, "CVodeSetQuadFdataB", 1)) return(1);
286 flag = CVodeSetQuadErrConB(cvadj_mem, TRUE);
287 if (check_flag(&flag, "CVodeSetQuadErrConB", 1)) return(1);
288 flag = CVodeSetQuadTolerancesB(cvadj_mem, CV_SS, &reltolB, &abstolQB);
289 if (check_flag(&flag, "CVodeSetQuadTolerancesB", 1)) return(1);
290 flag = CVodeQuadMallocB(cvadj_mem, fQB, qB);
291 if (check_flag(&flag, "CVodeQuadMallocB", 1)) return(1);
292
293 /* Backward Integration */
294 printf("Integrate backwards\n");
295 flag = CVodeB(cvadj_mem, T0, yB, &time, CV_NORMAL);
296 if (check_flag(&flag, "CVodeB", 1)) return(1);
297
298 flag = CVodeGetQuadB(cvadj_mem, qB);
299 if (check_flag(&flag, "CVodeGetQuadB", 1)) return(1);
300
301 PrintOutput(yB, qB);
302
303 /* Reinitialize backward phase (new tB0) */
304 Ith(yB,1) = ZERO;
305 Ith(yB,2) = ZERO;
306 Ith(yB,3) = ZERO;
307
308 Ith(qB,1) = ZERO;
309 Ith(qB,2) = ZERO;
310 Ith(qB,3) = ZERO;
311
312 printf("Re-initialize CVODES memory for backward run\n");
313 flag = CVodeReInitB(cvadj_mem, fB, TB2, yB, CV_SS, &reltolB, &abstolB);
314 if (check_flag(&flag, "CVodeReInitB", 1)) return(1);
315 flag = CVodeQuadReInitB(cvadj_mem, fQB, qB);
316 if (check_flag(&flag, "CVodeQuadReInitB", 1)) return(1);
317
318 /* Backward Integration */
319 printf("Integrate backwards\n");
320 flag = CVodeB(cvadj_mem, T0, yB, &time, CV_NORMAL);
321 if (check_flag(&flag, "CVodeB", 1)) return(1);
322

```

```

323     flag = CVodeGetQuadB(cvadj_mem, qB);
324     if (check_flag(&flag, "CVodeGetQuadB", 1)) return(1);
325
326     PrintOutput(yB, qB);
327
328     /* Free memory */
329     printf("Free memory\n\n");
330     CVodeFree(cvode_mem);
331     N_VDestroy_Serial(abstol);
332     N_VDestroy_Serial(y);
333     N_VDestroy_Serial(q);
334     N_VDestroy_Serial(yB);
335     N_VDestroy_Serial(qB);
336     CVadjFree(cvadj_mem);
337     free(data);
338
339     return(0);
340
341 }
342
343 /*
344  *-----
345  * FUNCTIONS CALLED BY CVODES
346  *-----
347  */
348
349 /*
350  * f routine. Compute f(t,y).
351  */
352
353 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
354 {
355     realtype y1, y2, y3, yd1, yd3;
356     UserData data;
357     realtype p1, p2, p3;
358
359     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
360     data = (UserData) f_data;
361     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
362
363     yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
364     yd3 = Ith(ydot,3) = p3*y2*y2;
365     Ith(ydot,2) = -yd1 - yd3;
366 }
367
368 /*
369  * Jacobian routine. Compute J(t,y).
370  */
371
372 static void Jac(long int N, DenseMat J, realtype t,
373                N_Vector y, N_Vector fy, void *jac_data,
374                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
375 {
376     realtype y1, y2, y3;

```



```

377   UserData data;
378   realtype p1, p2, p3;
379
380   y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
381   data = (UserData) jac_data;
382   p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
383
384   IJth(J,1,1) = -p1;   IJth(J,1,2) = p2*y3;           IJth(J,1,3) = p2*y2;
385   IJth(J,2,1) =  p1;   IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
386                       IJth(J,3,2) = 2*p3*y2;
387 }
388
389 /*
390  * fQ routine. Compute fQ(t,y).
391  */
392
393 static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
394 {
395
396   Ith(qdot,1) = Ith(y,3);
397
398 }
399
400 /*
401  * fB routine. Compute fB(t,y,yB).
402  */
403
404 static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot, void *f_dataB)
405 {
406   UserData data;
407   realtype y1, y2, y3;
408   realtype p1, p2, p3;
409   realtype l1, l2, l3;
410   realtype l21, l32, y23;
411
412   data = (UserData) f_dataB;
413
414   /* The p vector */
415   p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
416
417   /* The y vector */
418   y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
419
420   /* The lambda vector */
421   l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);
422
423   /* Temporary variables */
424   l21 = l2-l1;
425   l32 = l3-l2;
426   y23 = y2*y3;
427
428   /* Load yBdot */
429   Ith(yBdot,1) = - p1*l21;
430   Ith(yBdot,2) = p2*y3*l21 - RCONST(2.0)*p3*y2*l32;

```

```

431     Ith(yBdot,3) = p2*y2*121 - RCONST(1.0);
432 }
433
434 /*
435  * JacB routine. Compute JB(t,y,yB).
436 */
437
438 static void JacB(long int NB, DenseMat JB, realtype t,
439                 N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
440                 N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B)
441 {
442     UserData data;
443     realtype y1, y2, y3;
444     realtype p1, p2, p3;
445
446     data = (UserData) jac_dataB;
447
448     /* The p vector */
449     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
450
451     /* The y vector */
452     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
453
454     /* Load JB */
455     IJth(JB,1,1) = p1;      IJth(JB,1,2) = -p1;
456     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;
457     IJth(JB,3,1) = -p2*y2; IJth(JB,3,2) = p2*y2;
458 }
459
460 /*
461  * fQB routine. Compute integrand for quadratures
462 */
463
464 static void fQB(realtype t, N_Vector y, N_Vector yB,
465                N_Vector qBdot, void *fQ_dataB)
466 {
467     UserData data;
468     realtype y1, y2, y3;
469     realtype p1, p2, p3;
470     realtype l1, l2, l3;
471     realtype l21, l32, y23;
472
473     data = (UserData) fQ_dataB;
474
475     /* The p vector */
476     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
477
478     /* The y vector */
479     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
480
481     /* The lambda vector */
482     l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);
483
484     /* Temporary variables */

```

```

485     l21 = l2-l1;
486     l32 = l3-l2;
487     y23 = y2*y3;
488
489     Ith(qBdot,1) = y1*l21;
490     Ith(qBdot,2) = - y23*l21;
491     Ith(qBdot,3) = y2*y2*l32;
492 }
493
494 /*
495  *-----
496  * PRIVATE FUNCTIONS
497  *-----
498  */
499
500 /*
501  * Print results after backward integration
502  */
503
504 static void PrintOutput(N_Vector yB, N_Vector qB)
505 {
506     printf("-----\n");
507 #if defined(SUNDIALS_EXTENDED_PRECISION)
508     printf("tB0:          %12.4Le\n",TB1);
509     printf("dG/dp:          %12.4Le %12.4Le %12.4Le\n",
510           -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
511     printf("lambda(t0): %12.4Le %12.4Le %12.4Le\n",
512           Ith(yB,1), Ith(yB,2), Ith(yB,3));
513 #elif defined(SUNDIALS_DOUBLE_PRECISION)
514     printf("tB0:          %12.4le\n",TB1);
515     printf("dG/dp:          %12.4le %12.4le %12.4le\n",
516           -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
517     printf("lambda(t0): %12.4le %12.4le %12.4le\n",
518           Ith(yB,1), Ith(yB,2), Ith(yB,3));
519 #else
520     printf("tB0:          %12.4e\n",TB1);
521     printf("dG/dp:          %12.4e %12.4e %12.4e\n",
522           -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
523     printf("lambda(t0): %12.4e %12.4e %12.4e\n",
524           Ith(yB,1), Ith(yB,2), Ith(yB,3));
525 #endif
526     printf("-----\n\n");
527 }
528
529 /*
530  * Check function return value.
531  *   opt == 0 means SUNDIALS function allocates memory so check if
532  *   returned NULL pointer
533  *   opt == 1 means SUNDIALS function returns a flag so check if
534  *   flag >= 0
535  *   opt == 2 means function allocates memory so check if returned
536  *   NULL pointer
537  */
538

```

```

539 static int check_flag(void *flagvalue, char *funcname, int opt)
540 {
541     int *errflag;
542
543     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
544     if (opt == 0 && flagvalue == NULL) {
545         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
546             funcname);
547         return(1); }
548
549     /* Check if flag < 0 */
550     else if (opt == 1) {
551         errflag = flagvalue;
552         if (*errflag < 0) {
553             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
554                 funcname, *errflag);
555             return(1); }}
556
557     /* Check if function returned NULL pointer - no memory allocated */
558     else if (opt == 2 && flagvalue == NULL) {
559         fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
560             funcname);
561         return(1); }
562
563     return(0);
564 }

```

E Listing of pvanx.c

```

1  /*
2  * -----
3  * $Revision: 1.17 $
4  * $Date: 2004/11/23 21:35:07 $
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 <= x <= 2, and the time interval 0 <= t <= 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 <= N <= MX.
42 * -----
43 */
44
45 #include <stdio.h>
46 #include <stdlib.h>
47 #include <math.h>
48 #include "mpi.h"
49 #include "cvodes.h"
50 #include "cvodea.h"
51 #include "nvector_parallel.h"
52 #include "sundialstypes.h"

```

```

53
54
55 /* Problem Constants */
56
57 #define XMAX RCONST(2.0) /* domain boundary */
58 #define MX 20 /* mesh dimension */
59 #define NEQ MX /* number of equations */
60 #define ATOL RCONST(1.e-5) /* scalar absolute tolerance */
61 #define TO RCONST(0.0) /* initial time */
62 #define TOUT RCONST(2.5) /* output time increment */
63
64 /* Adjoint Problem Constants */
65
66 #define NP 2 /* number of parameters */
67 #define STEPS 200 /* steps between check points */
68
69 #define ZERO RCONST(0.0)
70 #define ONE RCONST(1.0)
71 #define TWO RCONST(2.0)
72
73 /* Type : UserData */
74
75 typedef struct {
76     realtype p[2]; /* model parameters */
77     realtype dx; /* spatial discretization grid */
78     realtype hdcoef, hacoef; /* diffusion and advection coefficients */
79     long int local_N;
80     long int npes, my_pe; /* total number of processes and current ID */
81     long int nperpe, nrem;
82     MPI_Comm comm; /* MPI communicator */
83     realtype *z1, *z2; /* work space */
84 } *UserData;
85
86 /* Prototypes of user-supplied functions */
87
88 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
89 static void fB(realtype t, N_Vector u,
90               N_Vector uB, N_Vector uBdot, void *f_dataB);
91
92 /* Prototypes of private functions */
93
94 static void SetIC(N_Vector u, realtype dx, long int my_length, long int my_base);
95 static void SetICback(N_Vector uB, long int my_base);
96 static realtype Xintgr(realtype *z, long int l, realtype dx);
97 static realtype Compute_g(N_Vector u, UserData data);
98 static void PrintOutput(realtype g_val, N_Vector uB, UserData data);
99 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
100
101 /*
102  *-----
103  * MAIN PROGRAM
104  *-----
105  */
106

```

```

107 int main(int argc, char *argv[])
108 {
109     UserData data;
110
111     void *cvarj_mem;
112     void *cvarj_mem;
113
114     N_Vector u;
115     realtype reltol, abstol;
116
117     N_Vector uB;
118
119     realtype dx, t, g_val;
120     int flag, my_pe, nprocs, npes, ncheck;
121     long int local_N=0, nperpe, nrem, my_base=0;
122
123     MPI_Comm comm;
124
125     data = NULL;
126     cvarj_mem = cvarj_mem = NULL;
127     u = uB = NULL;
128
129     /*-----
130      Initialize MPI and get total number of pe's, and my_pe
131      -----*/
132     MPI_Init(&argc, &argv);
133     comm = MPI_COMM_WORLD;
134     MPI_Comm_size(comm, &nprocs);
135     MPI_Comm_rank(comm, &my_pe);
136
137     npes = nprocs - 1; /* pe's dedicated to PDE integration */
138
139     if ( npes <= 0 ) {
140         if (my_pe == npes)
141             fprintf(stderr, "\nMPI_ERROR(%d): number of processes must be >= 2\n\n",
142                 my_pe);
143         MPI_Finalize();
144         return(1);
145     }
146
147     /*-----
148      Set local vector length
149      -----*/
150     nperpe = NEQ/npes;
151     nrem = NEQ - npes*nperpe;
152     if (my_pe < npes) {
153
154         /* PDE vars. distributed to this proccess */
155         local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
156         my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;
157
158     } else {
159
160         /* Make last process inactive for forward phase */

```

```

161     local_N = 0;
162
163 }
164
165 /*-----
166     Allocate and load user data structure
167     -----*/
168 data = (UserData) malloc(sizeof *data);
169 if (check_flag((void *)data , "malloc", 2, my_pe)) MPI_Abort(comm, 1);
170 data->p[0] = ONE;
171 data->p[1] = RCONST(0.5);
172 dx = data->dx = XMAX/((realtype)(MX+1));
173 data->hdcoef = data->p[0]/(dx*dx);
174 data->hacoef = data->p[1]/(TWO*dx);
175 data->comm = comm;
176 data->npes = npes;
177 data->my_pe = my_pe;
178 data->nperpe = nperpe;
179 data->nrem = nrem;
180 data->local_N = local_N;
181
182 /*-----
183     Forward integration phase
184     -----*/
185
186 /* Set relative and absolute tolerances for forward phase */
187 reltol = ZERO;
188 abstol = ATOL;
189
190 /* Allocate and initialize forward variables */
191 u = N_VNew_Parallel(comm, local_N, NEQ);
192 if (check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
193 SetIC(u, dx, local_N, my_base);
194
195 /* Allocate CVODES memory for forward integration */
196 ccode_mem = CCodeCreate(CV_ADAMS, CV_FUNCTIONAL);
197 if (check_flag((void *)ccode_mem, "CCodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
198
199 flag = CCodeSetFdata(ccode_mem, data);
200 if (check_flag(&flag, "CCodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
201
202 flag = CCodeMalloc(ccode_mem, f, T0, u, CV_SS, &reltol, &abstol);
203 if (check_flag(&flag, "CCodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
204
205 /* Allocate combined forward/backward memory */
206 cvadj_mem = CVadjMalloc(ccode_mem, STEPS);
207 if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0, my_pe)) MPI_Abort(comm, 1);
208
209 /* Integrate to TOUT and collect check point information */
210 flag = CCodeF(cvadj_mem, TOUT, u, &t, CV_NORMAL, &ncheck);
211 if (check_flag(&flag, "CCodeF", 1, my_pe)) MPI_Abort(comm, 1);
212
213 if(my_pe == npes) {
214     printf("(PE# %d) Number of check points: %d\n",my_pe, ncheck);

```



```

215     CVadjGetCheckPointsList(cvadj_mem);
216 }
217
218 /*-----
219     Compute and value of g(t_f)
220     -----*/
221 g_val = Compute_g(u, data);
222
223 /*-----
224     Backward integration phase
225     -----*/
226
227 if (my_pe == npes) {
228
229     /* Activate last process for integration of the quadrature equations */
230     local_N = NP;
231
232 } else {
233
234     /* Allocate work space */
235     data->z1 = (realtype *)malloc(local_N*sizeof(realtype));
236     if (check_flag((void *)data->z1, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
237     data->z2 = (realtype *)malloc(local_N*sizeof(realtype));
238     if (check_flag((void *)data->z2, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
239
240 }
241
242 /* Allocate and initialize backward variables */
243 uB = N_VNew_Parallel(comm, local_N, NEQ+NP);
244 if (check_flag((void *)uB, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
245 SetICback(uB, my_base);
246
247 /* Allocate CVODES memory for the backward integration */
248 flag = CNodeCreateB(cvadj_mem, CV_ADAMS, CV_FUNCTIONAL);
249 if (check_flag(&flag, "CNodeCreateB", 1, my_pe)) MPI_Abort(comm, 1);
250 flag = CNodeSetFdataB(cvadj_mem, data);
251 if (check_flag(&flag, "CNodeSetFdataB", 1, my_pe)) MPI_Abort(comm, 1);
252 flag = CNodeMallocB(cvadj_mem, fB, TOUT, uB, CV_SS, &reltol, &abstol);
253 if (check_flag(&flag, "CNodeMallocB", 1, my_pe)) MPI_Abort(comm, 1);
254
255 /* Integrate to T0 */
256 flag = CNodeB(cvadj_mem, T0, uB, &t, CV_NORMAL);
257 if (check_flag(&flag, "CNodeB", 1, my_pe)) MPI_Abort(comm, 1);
258
259 /* Print results (adjoint states and quadrature variables) */
260 PrintOutput(g_val, uB, data);
261
262
263 /* Free memory */
264 N_VDestroy_Parallel(u);
265 N_VDestroy_Parallel(uB);
266 CNodeFree(cnode_mem);
267 CVadjFree(cvadj_mem);
268 if (my_pe != npes) {

```

```

269     free(data->z1);
270     free(data->z2);
271 }
272 free(data);
273
274 MPI_Finalize();
275
276 return(0);
277 }
278
279 /*
280 *-----
281 * FUNCTIONS CALLED BY CVOIDS
282 *-----
283 */
284
285 /*
286 * f routine. Compute f(t,u) for forward phase.
287 */
288
289 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
290 {
291     realtype uLeft, uRight, ui, ult, urt;
292     realtype hordc, horac, hdiff, hadv;
293     realtype *udata, *dudata;
294     long int i, my_length;
295     int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
296     UserData data;
297     MPI_Status status;
298     MPI_Comm comm;
299
300     /* Extract MPI info. from data */
301     data = (UserData) f_data;
302     comm = data->comm;
303     npes = data->npes;
304     my_pe = data->my_pe;
305
306     /* If this process is inactive, return now */
307     if (my_pe == npes) return;
308
309     /* Extract problem constants from data */
310     hordc = data->hdcoef;
311     horac = data->hacoef;
312
313     /* Find related processes */
314     my_pe_m1 = my_pe - 1;
315     my_pe_p1 = my_pe + 1;
316     last_pe = npes - 1;
317
318     /* Obtain local arrays */
319     udata = NV_DATA_P(u);
320     dudata = NV_DATA_P(udot);
321     my_length = NV_LOCLENGTH_P(u);
322     my_last = my_length - 1;

```

```

323
324 /* Pass needed data to processes before and after current process. */
325 if (my_pe != 0)
326     MPI_Send(&udata[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
327 if (my_pe != last_pe)
328     MPI_Send(&udata[my_length-1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
329
330 /* Receive needed data from processes before and after current process. */
331 if (my_pe != 0)
332     MPI_Recv(&uLeft, 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
333 else uLeft = ZERO;
334 if (my_pe != last_pe)
335     MPI_Recv(&uRight, 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
336             &status);
337 else uRight = ZERO;
338
339 /* Loop over all grid points in current process. */
340 for (i=0; i<my_length; i++) {
341
342     /* Extract u at x_i and two neighboring points */
343     ui = udata[i];
344     ult = (i==0) ? uLeft: udata[i-1];
345     urt = (i==my_length-1) ? uRight : udata[i+1];
346
347     /* Set diffusion and advection terms and load into udot */
348     hdiff = hordc*(ult - TWO*ui + urt);
349     hadv = horac*(urt - ult);
350     dudata[i] = hdiff + hadv;
351 }
352 }
353
354 /*
355  * fB routine. Compute right hand side of backward problem
356  */
357
358 static void fB(realtype t, N_Vector u,
359               N_Vector uB, N_Vector uBdot, void *f_dataB)
360 {
361     realtype *uBdata, *duBdata, *udata;
362     realtype uBLeft, uBRight, uBi, uBlt, uBrt;
363     realtype uLeft, uRight, ui, ult, urt;
364     realtype dx, hordc, horac, hdiff, hadv;
365     realtype *z1, *z2, intgr1, intgr2;
366     long int i, my_length;
367     int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
368     UserData data;
369     realtype data_in[2], data_out[2];
370     MPI_Status status;
371     MPI_Comm comm;
372
373     /* Extract MPI info. from data */
374     data = (UserData) f_dataB;
375     comm = data->comm;
376     npes = data->npes;

```

```

377 my_pe = data->my_pe;
378
379 if (my_pe == npes) { /* This process performs the quadratures */
380
381     /* Obtain local arrays */
382     duBdata = NV_DATA_P(uBdot);
383     my_length = NV_LOCLENGTH_P(uB);
384
385     /* Loop over all other processes and load right hand side of quadrature eqs. */
386     duBdata[0] = ZERO;
387     duBdata[1] = ZERO;
388     for (i=0; i<npes; i++) {
389         MPI_Recv(&intgr1, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
390         duBdata[0] += intgr1;
391         MPI_Recv(&intgr2, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
392         duBdata[1] += intgr2;
393     }
394
395 } else { /* This process integrates part of the PDE */
396
397     /* Extract problem constants and work arrays from data */
398     dx      = data->dx;
399     hordc = data->hdcoef;
400     horac = data->haccoef;
401     z1     = data->z1;
402     z2     = data->z2;
403
404     /* Obtain local arrays */
405     uBdata = NV_DATA_P(uB);
406     duBdata = NV_DATA_P(uBdot);
407     udata = NV_DATA_P(u);
408     my_length = NV_LOCLENGTH_P(uB);
409
410     /* Compute related parameters. */
411     my_pe_m1 = my_pe - 1;
412     my_pe_p1 = my_pe + 1;
413     last_pe  = npes - 1;
414     my_last  = my_length - 1;
415
416     /* Pass needed data to processes before and after current process. */
417     if (my_pe != 0) {
418         data_out[0] = udata[0];
419         data_out[1] = uBdata[0];
420
421         MPI_Send(data_out, 2, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
422     }
423     if (my_pe != last_pe) {
424         data_out[0] = udata[my_length-1];
425         data_out[1] = uBdata[my_length-1];
426
427         MPI_Send(data_out, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
428     }
429
430     /* Receive needed data from processes before and after current process. */

```

```

431     if (my_pe != 0) {
432         MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
433
434         uLeft = data_in[0];
435         uBLeft = data_in[1];
436     } else {
437         uLeft = ZERO;
438         uBLeft = ZERO;
439     }
440     if (my_pe != last_pe) {
441         MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm, &status);
442
443         uRight = data_in[0];
444         uBRight = data_in[1];
445     } else {
446         uRight = ZERO;
447         uBRight = ZERO;
448     }
449
450     /* Loop over all grid points in current process. */
451     for (i=0; i<my_length; i++) {
452
453         /* Extract uB at x_i and two neighboring points */
454         uBi = uBdata[i];
455         uBlt = (i==0) ? uBLeft: uBdata[i-1];
456         uBrt = (i==my_length-1) ? uBRight : uBdata[i+1];
457
458         /* Set diffusion and advection terms and load into udot */
459         hdiff = hordc*(uBlt - TWO*uBi + uBrt);
460         hadv = horac*(uBrt - uBlt);
461         duBdata[i] = - hdiff + hadv;
462
463         /* Extract u at x_i and two neighboring points */
464         ui = udata[i];
465         ult = (i==0) ? uLeft: udata[i-1];
466         urt = (i==my_length-1) ? uRight : udata[i+1];
467
468         /* Load integrands of the two space integrals */
469         z1[i] = uBdata[i]*(ult - TWO*ui + urt)/(dx*dx);
470         z2[i] = uBdata[i]*(urt - ult)/(TWO*dx);
471     }
472
473     /* Compute local integrals */
474     intgr1 = Xintgr(z1, my_length, dx);
475     intgr2 = Xintgr(z2, my_length, dx);
476
477     /* Send local integrals to 'quadrature' process */
478     MPI_Send(&intgr1, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
479     MPI_Send(&intgr2, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
480
481 }
482
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("dgdg(tf)\n [ 1]: %8Le\n [ 2]: %8Le\n\n", -uBdata[0], -uBdata[1]);
612         #elif defined(SUNDIALS_DOUBLE_PRECISION)
613             printf("\ng(tf) = %8le\n\n", g_val);
614             printf("dgdg(tf)\n [ 1]: %8le\n [ 2]: %8le\n\n", -uBdata[0], -uBdata[1]);
615         #else
616             printf("\ng(tf) = %8e\n\n", g_val);
617             printf("dgdg(tf)\n [ 1]: %8e\n [ 2]: %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(" [%2ld]: %8Le\n", i+1, mu[i]);
635         #elif defined(SUNDIALS_DOUBLE_PRECISION)
636             for (i=0; i<NEQ; i++)
637                 printf(" [%2ld]: %8le\n", i+1, mu[i]);
638         #else
639             for (i=0; i<NEQ; i++)
640                 printf(" [%2ld]: %8e\n", i+1, mu[i]);
641         #endif
642
643         free(mu);
644
645     } else {
646

```



```

647     MPI_Send(uBdata, local_N, PVEC_REAL_MPI_TYPE, npes, 0, comm);
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 = 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 pvakx.c

```
1  /*
2  * -----
3  * $Revision: 1.11 $
4  * $Date: 2004/11/22 20:50:33 $
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 #include "mpi.h"
17 #include "cvodes.h"
18 #include "cvodea.h"
19 #include "cvspgmr.h"
20 #include "cvbbdpre.h"
21 #include "nvector_parallel.h"
22 #include "sundialstypes.h"
23 #include "sundialsmath.h"
24
25 /*
26 *-----
27 * Constants
28 *-----
29 */
30
31 #ifdef USE3D
32 #define DIM 3
33 #else
34 #define DIM 2
35 #endif
36
37 /* Domain definition */
38
39 #define XMIN RCONST(0.0)
40 #define XMAX RCONST(20.0)
41 #define MX 20 /* no. of divisions in x dir. */
42 #define NPX 2 /* no. of procs. in x dir. */
43
44 #define YMIN RCONST(0.0)
45 #define YMAX RCONST(20.0)
46 #define MY 40 /* no. of divisions in y dir. */
47 #define NPY 2 /* no. of procs. in y dir. */
48
49 #ifdef USE3D
50 #define ZMIN RCONST(0.0)
51 #define ZMAX RCONST(20.0)
52 #define MZ 20 /* no. of divisions in z dir. */
```

```

53 #define NPZ 1      /* no. of procs. in z dir.    */
54 #endif
55
56 /* Parameters for source Gaussians */
57
58 #define G1_AMPL  RCONST(1.0)
59 #define G1_SIGMA RCONST(1.7)
60 #define G1_X     RCONST(4.0)
61 #define G1_Y     RCONST(8.0)
62 #ifndef USE3D
63 #define G1_Z     RCONST(8.0)
64 #endif
65
66 #define G2_AMPL  RCONST(0.8)
67 #define G2_SIGMA RCONST(3.0)
68 #define G2_X     RCONST(16.0)
69 #define G2_Y     RCONST(12.0)
70 #ifndef USE3D
71 #define G2_Z     RCONST(12.0)
72 #endif
73
74 #define G_MIN    RCONST(1.0e-5)
75
76 /* Diffusion coeff., max. velocity, domain width in y dir. */
77
78 #define DIFF_COEF RCONST(1.0)
79 #define V_MAX     RCONST(1.0)
80 #define L         (YMAX-YMIN)/RCONST(2.0)
81 #define V_COEFF   V_MAX/L/L
82
83 /* Initial and final times */
84
85 #define ti       RCONST(0.0)
86 #define tf       RCONST(10.0)
87
88 /* Integration tolerances */
89
90 #define RTOL     RCONST(1.0e-8) /* states */
91 #define ATOL     RCONST(1.0e-6)
92
93 #define RTOL_Q   RCONST(1.0e-8) /* forward quadrature */
94 #define ATOL_Q   RCONST(1.0e-6)
95
96 #define RTOL_B   RCONST(1.0e-8) /* adjoint variables */
97 #define ATOL_B   RCONST(1.0e-6)
98
99 #define RTOL_QB  RCONST(1.0e-8) /* backward quadratures */
100 #define ATOL_QB RCONST(1.0e-6)
101
102 /* Steps between check points */
103
104 #define STEPS 200
105
106 #define ZERO RCONST(0.0)

```

```

107 #define ONE   RCONST(1.0)
108 #define TWO   RCONST(2.0)
109
110 /*
111  *-----
112  * Macros
113  *-----
114  */
115
116 #define FOR_DIM for(dim=0; dim<DIM; dim++)
117
118 /* IJth:      (i[0],i[1],i[2])-th vector component          */
119 /* IJth_ext:  (i[0],i[1],i[2])-th vector component in the extended array */
120
121 #ifndef USE3D
122 #define IJth(y,i)      ( y[(i[0])+(1_m[0]*((i[1])+(i[2])*1_m[1]))] )
123 #define IJth_ext(y,i) ( y[(i[0]+1)+((1_m[0]+2)*((i[1]+1)+(i[2]+1)*(1_m[1]+2)))] )
124 #else
125 #define IJth(y,i)      (y[i[0]+(i[1])*1_m[0]])
126 #define IJth_ext(y,i) (y[ (i[0]+1) + (i[1]+1) * (1_m[0]+2)])
127 #endif
128
129 /*
130  *-----
131  * Type definition: ProblemData
132  *-----
133  */
134
135 typedef struct {
136     /* Domain */
137     realtype xmin[DIM]; /* "left" boundaries */
138     realtype xmax[DIM]; /* "right" boundaries */
139     int m[DIM];          /* number of grid points */
140     realtype dx[DIM];    /* grid spacing */
141     realtype dOmega;     /* differential volume */
142
143     /* Parallel stuff */
144     MPI_Comm comm;       /* MPI communicator */
145     int myId;            /* process id */
146     int npes;            /* total number of processes */
147     int num_procs[DIM];  /* number of processes in each direction */
148     int nbr_left[DIM];   /* MPI ID of "left" neighbor */
149     int nbr_right[DIM];  /* MPI ID of "right" neighbor */
150     int m_start[DIM];    /* "left" index in the global domain */
151     int l_m[DIM];        /* number of local grid points */
152     realtype *y_ext;     /* extended data array */
153     realtype *buf_send;  /* Send buffer */
154     realtype *buf_recv;  /* Receive buffer */
155     int buf_size;        /* Buffer size */
156
157     /* Source */
158     N_Vector p;          /* Source parameters */
159
160 } *ProblemData;

```

```

161
162 /*
163 *-----
164 * Interface functions to CVODES
165 *-----
166 */
167
168 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
169 static void f_local(long int Nlocal, realtype t, N_Vector y,
170                    N_Vector ydot, void *f_data);
171
172 static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data);
173
174
175 static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
176               void *f_dataB);
177 static void fB_local(long int NlocalB, realtype t,
178                     N_Vector y, N_Vector yB, N_Vector yBdot,
179                     void *f_dataB);
180
181 static void fQB(realtype t, N_Vector y, N_Vector yB,
182                 N_Vector qBdot, void *fQ_dataB);
183
184 /*
185 *-----
186 * Private functions
187 *-----
188 */
189
190 static void SetData(ProblemData d, MPI_Comm comm, int npes, int myId,
191                    long int *neq, long int *l_neq);
192 static void SetSource(ProblemData d);
193 static void f_comm( long int Nlocal, realtype t, N_Vector y, void *f_data);
194 static void Load_yext(realtype *src, ProblemData d);
195 static void PrintHeader();
196 static void PrintFinalStats(void *cvsde_mem);
197 static void OutputGradient(int myId, N_Vector qB, ProblemData d);
198
199 /*
200 *-----
201 * Main program
202 *-----
203 */
204
205 int main(int argc, char *argv[])
206 {
207     ProblemData d;
208
209     MPI_Comm comm;
210     int npes, npes_needed;
211     int myId;
212
213     long int neq, l_neq;
214

```

```

215 void *cnode_mem;
216 N_Vector y, q;
217 realtype abstol, reltol, abstolQ, reltolQ;
218 void *bbdp_data;
219 int mudq, mldq, mukeep, mlkeep;
220
221 void *cvadj_mem;
222 void *cnode_memB;
223 N_Vector yB, qB;
224 realtype abstolB, reltolB, abstolQB, reltolQB;
225 int mudqB, mldqB, mukeepB, mlkeepB;
226
227 realtype tret, *qdata, G;
228
229 int ncheckpnt, flag;
230
231 booleantype output;
232
233 /* Initialize MPI and set Ids */
234 MPI_Init(&argc, &argv);
235 comm = MPI_COMM_WORLD;
236 MPI_Comm_rank(comm, &myId);
237
238 /* Check number of processes */
239 npes_needed = NPX * NPY;
240 #ifdef USE3D
241 npes_needed *= NPZ;
242 #endif
243 MPI_Comm_size(comm, &npes);
244 if (npes_needed != npes) {
245     if (myId == 0)
246         fprintf(stderr, "I need %d processes but I only got %d\n",
247                 npes_needed, npes);
248     MPI_Abort(comm, EXIT_FAILURE);
249 }
250
251 /* Test if matlab output is requested */
252 if (argc > 1) output = TRUE;
253 else output = FALSE;
254
255 /* Allocate and set problem data structure */
256 d = (ProblemData) malloc(sizeof *d);
257 SetData(d, comm, npes, myId, &neq, &l_neq);
258
259 if (myId == 0) PrintHeader();
260
261 /*-----
262 Forward integration phase
263 -----*/
264
265 /* Allocate space for y and set it with the I.C. */
266 y = N_VNew_Parallel(comm, l_neq, neq);
267 N_VConst(ZERO, y);
268

```

```

269  /* Allocate and initialize qB (local contributin to cost) */
270  q = N_VNew_Parallel(comm, 1, npes);
271  N_VConst(ZERO, q);
272
273  /* Create CVODES object, attach user data, and allocate space */
274  cvode_mem = CNodeCreate(CV_BDF, CV_NEWTON);
275  flag = CNodeSetFdata(cvode_mem, d);
276  abstol = ATOL;
277  reltol = RTOL;
278  flag = CNodeMalloc(cvode_mem, f, ti, y, CV_SS, &reltol, &abstol);
279
280  /* Attach preconditioner and linear solver modules */
281  mudq = mldq = d->l_m[0]+1;
282  mukeep = mlkeep = 2;
283  bbdp_data = (void *) CVBBDPrecAlloc(cvode_mem, l_neq, mudq, mldq,
284                                     mukeep, mlkeep, ZERO,
285                                     f_local, NULL);
286  flag = CVBBDSpgmr(cvode_mem, PREC_LEFT, 0, bbdp_data);
287
288  /* Initialize quadrature calculations */
289  abstolQ = ATOL_Q;
290  reltolQ = RTOL_Q;
291  flag = CNodeSetQuadFdata(cvode_mem, d);
292  flag = CNodeSetQuadErrCon(cvode_mem, TRUE);
293  flag = CNodeSetQuadTolerances(cvode_mem, CV_SS, &reltolQ, &abstolQ);
294  flag = CNodeQuadMalloc(cvode_mem, fQ, q);
295
296  /* Allocate space for the adjoint calculation */
297  cvadj_mem = CVadjMalloc(cvode_mem, STEPS);
298
299  /* Integrate forward in time while storing check points */
300  if (myId == 0) printf("Begin forward integration... ");
301  flag = CNodeF(cvadj_mem, tf, y, &tret, CV_NORMAL, &ncheckpnt);
302  if (myId == 0) printf("done. ");
303
304  /* Extract quadratures */
305  flag = CNodeGetQuad(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 = CVodeSetQuadFdataB(cvadj_mem, d);
349  flag = CVodeSetQuadErrConB(cvadj_mem, TRUE);
350  flag = CVodeSetQuadTolerancesB(cvadj_mem, CV_SS, &reltolQB, &abstolQB);
351  flag = CVodeQuadMallocB(cvadj_mem, fQB, qB);
352
353  /* Integrate backwards */
354  if (myId == 0) printf("Begin backward integration... ");
355  flag = CVodeB(cvadj_mem, ti, yB, &tret, CV_NORMAL);
356  if (myId == 0) printf("done.\n");
357
358  /* Print statistics for backward run */
359  if (myId == 0) {
360      cvode_memB = CVadjGetCVodeBmem(cvadj_mem);
361      PrintFinalStats(cvode_memB);
362  }
363
364  /* Extract quadratures */
365  flag = CVodeGetQuadB(cvadj_mem, qB);
366
367  /* Process 0 collects the gradient components and prints them */
368  if (output) {
369      OutputGradient(myId, qB, d);
370      if (myId == 0) printf("Wrote matlab file 'grad.m'.\n");
371  }
372
373  /* Free memory */
374  N_VDestroy_Parallel(y);
375  N_VDestroy_Parallel(q);
376  N_VDestroy_Parallel(qB);

```



```

377     N_VDestroy_Parallel(yB);
378
379     CVBBDPrecFree(bbdp_data);
380     CVadjFree(cvadj_mem);
381     CVodeFree(cvode_mem);
382
383     MPI_Finalize();
384
385     return(0);
386 }
387
388 /*
389 *-----
390 * SetData:
391 * Allocate space for the ProblemData structure.
392 * Set fields in the ProblemData structure.
393 * Return local and global problem dimensions.
394 *
395 * SetSource:
396 * Instantiates the source parameters for a combination of two
397 * Gaussian sources.
398 *-----
399 */
400
401 static void SetData(ProblemData d, MPI_Comm comm, int npes, int myId,
402                    long int *neq, long int *l_neq)
403 {
404     int n[DIM], nd[DIM];
405     int dim, size;
406
407     /* Set MPI communicator, id, and total number of processes */
408
409     d->comm = comm;
410     d->myId = myId;
411     d->npes = npes;
412
413     /* Set domain boundaries */
414
415     d->xmin[0] = XMIN;
416     d->xmax[0] = XMAX;
417     d->m[0]    = MX;
418
419     d->xmin[1] = YMIN;
420     d->xmax[1] = YMAX;
421     d->m[1]    = MY;
422
423     #ifdef USE3D
424     d->xmin[2] = ZMIN;
425     d->xmax[2] = ZMAX;
426     d->m[2]    = MZ;
427     #endif
428
429     /* Calculate grid spacing and differential volume */
430

```

```

431     d->dOmega = ONE;
432     FOR_DIM {
433         d->dx[dim] = (d->xmax[dim] - d->xmin[dim]) / d->m[dim];
434         d->m[dim] +=1;
435         d->dOmega *= d->dx[dim];
436     }
437
438     /* Set partitioning */
439
440     d->num_procs[0] = NPX;
441     n[0] = NPX;
442     nd[0] = d->m[0] / NPX;
443
444     d->num_procs[1] = NPY;
445     n[1] = NPY;
446     nd[1] = d->m[1] / NPY;
447
448     #ifdef USE3D
449     d->num_procs[2] = NPZ;
450     n[2] = NPZ;
451     nd[2] = d->m[2] / NPZ;
452     #endif
453
454     /* Compute the neighbors */
455
456     d->nbr_left[0] = (myId%n[0]) == 0           ? myId : myId-1;
457     d->nbr_right[0] = (myId%n[0]) == n[0]-1     ? myId : myId+1;
458
459     d->nbr_left[1] = (myId/n[0])%n[1] == 0      ? myId : myId-n[0];
460     d->nbr_right[1] = (myId/n[0])%n[1] == n[1]-1 ? myId : myId+n[0];
461
462     #ifdef USE3D
463     d->nbr_left[2] = (myId/n[0]/n[1])%n[2] == 0 ? myId : myId-n[0]*n[1];
464     d->nbr_right[2] = (myId/n[0]/n[1])%n[2] == n[2]-1 ? myId : myId+n[0]*n[1];
465     #endif
466
467     /* Compute the local subdomains
468         m_start: left border in global index space
469         l_m:      length of the subdomain */
470
471     d->m_start[0] = (myId%n[0])*nd[0];
472     d->l_m[0] = d->nbr_right[0] == myId ? d->m[0] - d->m_start[0] : nd[0];
473
474     d->m_start[1] = ((myId/n[0])%n[1])*nd[1];
475     d->l_m[1] = d->nbr_right[1] == myId ? d->m[1] - d->m_start[1] : nd[1];
476
477     #ifdef USE3D
478     d->m_start[2] = (myId/n[0]/n[1])*nd[2];
479     d->l_m[2] = d->nbr_right[2] == myId ? d->m[2] - d->m_start[2] : nd[2];
480     #endif
481
482     /* Allocate memory for the y_ext array
483         (local solution + data from neighbors) */
484

```

```

485     size = 1;
486     FOR_DIM size *= d->l_m[dim]+2;
487     d->y_ext = (realtype *) malloc( size*sizeof(realtype));
488
489     /* Initialize Buffer field.
490        Size of buffer is checked when needed */
491
492     d->buf_send = NULL;
493     d->buf_recv = NULL;
494     d->buf_size = 0;
495
496     /* Allocate space for the source parameters */
497
498     *neq = 1; *l_neq = 1;
499     FOR_DIM {*neq *= d->m[dim]; *l_neq *= d->l_m[dim];}
500     d->p = NVNew_Parallel(comm, *l_neq, *neq);
501
502     /* Initialize the parameters for a source with Gaussian profile */
503
504     SetSource(d);
505
506 }
507
508 static void SetSource(ProblemData d)
509 {
510     int *l_m, *m_start;
511     realtype *xmin, *xmax, *dx;
512     realtype x[DIM], g, *pdata;
513     int i[DIM];
514
515     l_m = d->l_m;
516     m_start = d->m_start;
517     xmin = d->xmin;
518     xmax = d->xmax;
519     dx = d->dx;
520
521
522     pdata = NV_DATA_P(d->p);
523
524     for(i[0]=0; i[0]<l_m[0]; i[0]++) {
525         x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
526         for(i[1]=0; i[1]<l_m[1]; i[1]++) {
527             x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
528 #ifndef USE3D
529             for(i[2]=0; i[2]<l_m[2]; i[2]++) {
530                 x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];
531
532                 g = G1_AMPL
533                     * exp( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
534                     * exp( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) )
535                     * exp( -SQR(G1_Z-x[2])/SQR(G1_SIGMA) );
536
537                 g += G2_AMPL
538                     * exp( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )

```

```

539         * exp( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) )
540         * exp( -SQR(G2_Z-x[2])/SQR(G2_SIGMA) );
541
542     if( g < G_MIN ) g = ZERO;
543
544     IJth(pdata, i) = g;
545 }
546 #else
547     g = G1_AMPL
548     * exp( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
549     * exp( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) );
550
551     g += G2_AMPL
552     * exp( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )
553     * exp( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) );
554
555     if( g < G_MIN ) g = ZERO;
556
557     IJth(pdata, i) = g;
558 #endif
559 }
560 }
561 }
562
563 /*
564 *-----
565 * f_comm:
566 * Function for inter-process communication
567 * Used both for the forward and backward phase.
568 *-----
569 */
570
571 static void f_comm(long int N_local, realtype t, N_Vector y, void *f_data)
572 {
573     int id, n[DIM], proc_cond[DIM], nbr[DIM][2];
574     ProblemData d;
575     realtype *yextdata, *ydata;
576     int l_m[DIM], dim;
577     int c, i[DIM], l[DIM-1];
578     realtype *buf_send, *buf_recv;
579     MPI_Status stat;
580     MPI_Comm comm;
581     int dir, size = 1, small = INT_MAX;
582
583     d = (ProblemData) f_data;
584     comm = d->comm;
585     id = d->myId;
586
587     /* extract data from domain*/
588     FOR_DIM {
589         n[dim] = d->num_procs[dim];
590         l_m[dim] = d->l_m[dim];
591     }
592     yextdata = d->y_ext;

```

```

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

```

```

647
648     if ( proc_cond[dim] ) {
649         /* Send buf_send and receive into buf_recv */
650         MPI_Send(buf_send, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm);
651         MPI_Recv(buf_recv, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm, &stat);
652     } else {
653         /* Receive into buf_recv and send buf_send */
654         MPI_Recv(buf_recv, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm, &stat);
655         MPI_Send(buf_send, c, PVEC_REAL_MPI_TYPE, nbr[dim][dir], 0, comm);
656     }
657
658     c=0;
659
660     /* Compute the index of the boundary (right or left) in yextdata */
661     i[dim] = (dir ^ proc_cond[dim]) ? l_m[dim] : -1;
662
663     /* Loop over all other dimensions and copy data into yextdata */
664 #ifdef USE3D
665     for(i[l[1]]=0; i[l[1]]<l_m[l[1]]; i[l[1]]++)
666 #endif
667     for(i[l[0]]=0; i[l[0]]<l_m[l[0]]; i[l[0]]++)
668         IJth_ext(yextdata, i) = buf_recv[c++];
669     }
670 } /* end loop over direction */
671 } /* end loop over dimension */
672 }
673
674 /*
675 *-----
676 * f and f_local:
677 * Forward phase ODE right-hand side
678 *-----
679 */
680
681 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
682 {
683     ProblemData d;
684     int l_neq=1;
685     int dim;
686
687     d = (ProblemData) f_data;
688     FOR_DIM l_neq *= d->l_m[dim];
689
690     /* Do all inter-processor communication */
691     f_comm(l_neq, t, y, f_data);
692
693     /* Compute right-hand side locally */
694     f_local(l_neq, t, y, ydot, f_data);
695 }
696
697 static void f_local(long int Nlocal, realtype t, N_Vector y,
698                   N_Vector ydot, void *f_data)
699 {
700     realtype *Ydata, *dydata, *pdata;

```

```

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

```

```

755
756      /* Source term*/
757      IJth(dydata, i) = IJth(pdata, i);
758
759      FOR_DIM {
760          i[dim]++;
761          cr[dim] = IJth_ext(Ydata, i);
762          i[dim]--;
763          cl[dim] = IJth_ext(Ydata, i);
764          i[dim]++;
765
766          /* Boundary conditions for the state variables */
767          if( i[dim]==l_m[dim]-1 && nbr_right[dim]==id)
768              cr[dim] = cl[dim];
769          else if( i[dim]==0 && nbr_left[dim]==id )
770              cl[dim] = cr[dim];
771
772          adv[dim] = v[dim] * (cr[dim]-cl[dim]) / (TWO*dx[dim]);
773          diff[dim] = DIFF_COEF * (cr[dim]-TWO*c+cl[dim]) / SQR(dx[dim]);
774
775          IJth(dydata, i) += (diff[dim] - adv[dim]);
776      }
777  }
778  }
779  #ifdef USE3D
780  }
781  #endif
782  }
783
784  /*
785  *-----
786  * fQ:
787  * Right-hand side of quadrature equations on forward integration.
788  * The only quadrature on this phase computes the local contribution
789  * to the function G.
790  *-----
791  */
792
793  static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
794  {
795      ProblemData d;
796      realtype *dqdata;
797
798      d = (ProblemData) fQ_data;
799
800      dqdata = NV_DATA_P(qdot);
801
802      dqdata[0] = N_VDotProd_Parallel(y,y);
803      dqdata[0] *= RCONST(0.5) * (d->dOmega);
804  }
805
806  /*
807  *-----
808  * fB and fB_local:

```



```

809  * Backward phase ODE right-hand side (the discretized adjoint PDE)
810  *-----
811  */
812
813  static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
814               void *f_dataB)
815  {
816      ProblemData d;
817      int l_neq=1;
818      int dim;
819
820      d = (ProblemData) f_dataB;
821      FOR_DIM l_neq *= d->l_m[dim];
822
823      /* Do all inter-processor communication */
824      f_comm(l_neq, t, yB, f_dataB);
825
826      /* Compute right-hand side locally */
827      fB_local(l_neq, t, y, yB, yBdot, f_dataB);
828  }
829
830  static void fB_local(long int NlocalB, realtype t,
831                     N_Vector y, N_Vector yB, N_Vector dyB,
832                     void *f_dataB)
833  {
834      realtype *YBdata, *dyBdata, *ydata;
835      realtype dx[DIM], c, v[DIM], cl[DIM], cr[DIM];
836      realtype adv[DIM], diff[DIM];
837      realtype xmin[DIM], xmax[DIM], x[DIM], x1;
838      int i[DIM], l_m[DIM], m_start[DIM], nbr_left[DIM], nbr_right[DIM], id;
839      ProblemData d;
840      int dim;
841
842      d = (ProblemData) f_dataB;
843
844      /* Extract stuff from data structure */
845      id = d->myId;
846      FOR_DIM {
847          xmin[dim]      = d->xmin[dim];
848          xmax[dim]      = d->xmax[dim];
849          l_m[dim]       = d->l_m[dim];
850          m_start[dim]   = d->m_start[dim];
851          dx[dim]        = d->dx[dim];
852          nbr_left[dim]  = d->nbr_left[dim];
853          nbr_right[dim] = d->nbr_right[dim];
854      }
855
856      dyBdata = NV_DATA_P(dyB);
857      ydata    = NV_DATA_P(y);
858
859      /* Copy local segment of yB to y_ext */
860      Load_yext(NV_DATA_P(yB), d);
861      YBdata = d->y_ext;
862

```

```

863  /* Velocity components in x1 and x2 directions (Poiseuille profile) */
864  v[1] = ZERO;
865  #ifdef USE3D
866  v[2] = ZERO;
867  #endif
868
869  /* local domain is [xmin+(m_start)*dx, xmin+(m_start+l_m-1)*dx] */
870  #ifdef USE3D
871  for(i[2]=0; i[2]<l_m[2]; i[2]++) {
872
873      x[2] = xmin[2] + (m_start[2]+i[2])*dx[2];
874  #endif
875
876  for(i[1]=0; i[1]<l_m[1]; i[1]++) {
877
878      x[1] = xmin[1] + (m_start[1]+i[1])*dx[1];
879
880      /* Velocity component in x0 direction (Poiseuille profile) */
881      x1 = x[1] - xmin[1] - L;
882      v[0] = V_COEFF * (L + x1) * (L - x1);
883
884      for(i[0]=0; i[0]<l_m[0]; i[0]++) {
885
886          x[0] = xmin[0] + (m_start[0]+i[0])*dx[0];
887
888          c = IJth_ext(YBdata, i);
889
890          /* Source term for adjoint PDE */
891          IJth(dyBdata, i) = -IJth(ydata, i);
892
893          FOR_DIM {
894
895              i[dim]+=1;
896              cr[dim] = IJth_ext(YBdata, i);
897              i[dim]-=2;
898              cl[dim] = IJth_ext(YBdata, i);
899              i[dim]+=1;
900
901              /* Boundary conditions for the adjoint variables */
902              if( i[dim]==l_m[dim]-1 && nbr_right[dim]==id)
903                  cr[dim] = cl[dim]-(TWO*dx[dim]*v[dim]/DIFF_COEF)*c;
904              else if( i[dim]==0 && nbr_left[dim]==id )
905                  cl[dim] = cr[dim]+(TWO*dx[dim]*v[dim]/DIFF_COEF)*c;
906
907              adv[dim] = v[dim] * (cr[dim]-cl[dim]) / (TWO*dx[dim]);
908              diff[dim] = DIFF_COEF * (cr[dim]-TWO*c+cl[dim]) / SQR(dx[dim]);
909
910              IJth(dyBdata, i) -= (diff[dim] + adv[dim]);
911          }
912      }
913  }
914  #ifdef USE3D
915  }
916  #endif

```

```

917 }
918
919 /*
920 *-----
921 * fQB:
922 * Right-hand side of quadrature equations on backward integration
923 * The i-th component of the gradient is nothing but int_t yB_i dt
924 *-----
925 */
926
927 static void fQB(realtype t, N_Vector y, N_Vector yB, N_Vector qBdot,
928               void *fQ_dataB)
929 {
930     ProblemData d;
931
932     d = (ProblemData) fQ_dataB;
933
934     N_VScale_Parallel(-(d->dOmega), yB, qBdot);
935 }
936
937 /*
938 *-----
939 * Load_yext:
940 * copies data from src (y or yB) into y_ext, which already contains
941 * data from neighboring processes.
942 *-----
943 */
944
945 static void Load_yext(realtype *src, ProblemData d)
946 {
947     int i[DIM], l_m[DIM], dim;
948
949     FOR_DIM l_m[dim] = d->l_m[dim];
950
951     /* copy local segment */
952 #ifdef USE3D
953     for (i[2]=0; i[2]<l_m[2]; i[2]++)
954 #endif
955         for(i[1]=0; i[1]<l_m[1]; i[1]++)
956             for(i[0]=0; i[0]<l_m[0]; i[0]++)
957                 IJth_ext(d->y_ext, i) = IJth(src, i);
958 }
959
960 /*
961 *-----
962 * PrintHeader:
963 * Print first lines of output (problem description)
964 *-----
965 */
966
967 static void PrintHeader()
968 {
969     printf("\nParallel Krylov adjoint sensitivity analysis example\n");
970     printf("%ldD Advection diffusion PDE with homogeneous Neumann B.C.\n",DIM);

```

```

971     printf("Computes gradient of G = int_t_Omega ( c_i^2 ) dt dOmega\n");
972     printf("with respect to the source values at each grid point.\n\n");
973
974     printf("Domain:\n");
975
976     #if defined(SUNDIALS_EXTENDED_PRECISION)
977         printf("    %Lf < x < %Lf    mx = %d  npe_x = %d \n",XMIN,XMAX,MX,NPX);
978         printf("    %Lf < y < %Lf    my = %d  npe_y = %d \n",YMIN,YMAX,MY,NPY);
979     #else
980         printf("    %f < x < %f    mx = %d  npe_x = %d \n",XMIN,XMAX,MX,NPX);
981         printf("    %f < y < %f    my = %d  npe_y = %d \n",YMIN,YMAX,MY,NPY);
982     #endif
983
984     #ifndef USE3D
985     #if defined(SUNDIALS_EXTENDED_PRECISION)
986         printf("    %Lf < z < %Lf    mz = %d  npe_z = %d \n",ZMIN,ZMAX,MZ,NPZ);
987     #else
988         printf("    %f < z < %f    mz = %d  npe_z = %d \n",ZMIN,ZMAX,MZ,NPZ);
989     #endif
990     #endif
991
992     printf("\n");
993 }
994
995 /*
996 *-----
997 * PrintFinalStats:
998 * Print final statistics contained in ccode_mem
999 *-----
1000 */
1001
1002 static void PrintFinalStats(void *ccode_mem)
1003 {
1004     long int lenrw, leniw ;
1005     long int lenrwSPGMR, leniwSPGMR;
1006     long int nst, nfe, nsetups, nni, ncnf, netf;
1007     long int nli, npe, nps, ncfl, nfeSPGMR;
1008     int flag;
1009
1010     flag = CVodeGetWorkspace(ccode_mem, &lenrw, &leniw);
1011     flag = CVodeGetNumSteps(ccode_mem, &nst);
1012     flag = CVodeGetNumRhsEvals(ccode_mem, &nfe);
1013     flag = CVodeGetNumLinSolvSetups(ccode_mem, &nsetups);
1014     flag = CVodeGetNumErrTestFails(ccode_mem, &netf);
1015     flag = CVodeGetNumNonlinSolvIters(ccode_mem, &nni);
1016     flag = CVodeGetNumNonlinSolvConvFails(ccode_mem, &ncnf);
1017
1018     flag = CVSpgmrGetWorkspace(ccode_mem, &lenrwSPGMR, &leniwSPGMR);
1019     flag = CVSpgmrGetNumLinIters(ccode_mem, &nli);
1020     flag = CVSpgmrGetNumPrecEvals(ccode_mem, &npe);
1021     flag = CVSpgmrGetNumPrecSolves(ccode_mem, &nps);
1022     flag = CVSpgmrGetNumConvFails(ccode_mem, &ncfl);
1023     flag = CVSpgmrGetNumRhsEvals(ccode_mem, &nfeSPGMR);
1024

```

```

1025     printf("\nFinal Statistics.. \n\n");
1026     printf("lenrw   = %6ld      leniw = %6ld\n", lenrw, leniw);
1027     printf("llrw    = %6ld      lliw  = %6ld\n", lenrwSPGMR, leniwSPGMR);
1028     printf("nst      = %6ld\n", nst);
1029     printf("nfe      = %6ld      nfel  = %6ld\n", nfe, nfeSPGMR);
1030     printf("nni      = %6ld      nli   = %6ld\n", nni, nli);
1031     printf("nsetups = %6ld      netf   = %6ld\n", nsetups, netf);
1032     printf("npe      = %6ld      nps    = %6ld\n", npe, nps);
1033     printf("ncfn     = %6ld      ncfl   = %6ld\n", ncfn, ncfl);
1034 }
1035
1036 /*
1037 *-----
1038 * OutputGradient:
1039 * Generate matlab m files for visualization
1040 * One file gradXXXX.m from each process + a driver grad.m
1041 *-----
1042 */
1043
1044 static void OutputGradient(int myId, N_Vector qB, ProblemData d)
1045 {
1046     FILE *fid;
1047     char filename[20];
1048     int *l_m, *m_start, i[DIM], ip;
1049     realtype *xmin, *xmax, *dx;
1050     realtype x[DIM], *pdata, p, *qBdata, g;
1051
1052     sprintf(filename, "grad%03d.m", myId);
1053     fid = fopen(filename, "w");
1054
1055     l_m = d->l_m;
1056     m_start = d->m_start;
1057     xmin = d->xmin;
1058     xmax = d->xmax;
1059     dx = d->dx;
1060
1061     qBdata = NV_DATA_P(qB);
1062     pdata = NV_DATA_P(d->p);
1063
1064     /* Write matlab files with solutions from each process */
1065
1066     for(i[0]=0; i[0]<l_m[0]; i[0]++) {
1067         x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
1068         for(i[1]=0; i[1]<l_m[1]; i[1]++) {
1069             x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
1070 #ifndef USE3D
1071             for(i[2]=0; i[2]<l_m[2]; i[2]++) {
1072                 x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];
1073                 g = IJth(qBdata, i);
1074                 p = IJth(pdata, i);
1075 #if defined(SUNDIALS_EXTENDED_PRECISION)
1076                 fprintf(fid, "x%d(%d,1) = %Le; \n", myId, i[0]+1, x[0]);
1077                 fprintf(fid, "y%d(%d,1) = %Le; \n", myId, i[1]+1, x[1]);
1078                 fprintf(fid, "z%d(%d,1) = %Le; \n", myId, i[2]+1, x[2]);

```

```

1079         fprintf(fid,"p%d(%d,%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, i[2]+1, p);
1080         fprintf(fid,"g%d(%d,%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, i[2]+1, g);
1081     #elif defined(SUNDIALS_DOUBLE_PRECISION)
1082         fprintf(fid,"x%d(%d,1) = %le; \n", myId, i[0]+1, x[0]);
1083         fprintf(fid,"y%d(%d,1) = %le; \n", myId, i[1]+1, x[1]);
1084         fprintf(fid,"z%d(%d,1) = %le; \n", myId, i[2]+1, x[2]);
1085         fprintf(fid,"p%d(%d,%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, i[2]+1, p);
1086         fprintf(fid,"g%d(%d,%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, i[2]+1, g);
1087     #else
1088         fprintf(fid,"x%d(%d,1) = %e; \n", myId, i[0]+1, x[0]);
1089         fprintf(fid,"y%d(%d,1) = %e; \n", myId, i[1]+1, x[1]);
1090         fprintf(fid,"z%d(%d,1) = %e; \n", myId, i[2]+1, x[2]);
1091         fprintf(fid,"p%d(%d,%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, i[2]+1, p);
1092         fprintf(fid,"g%d(%d,%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, i[2]+1, g);
1093     #endif
1094 }
1095 #else
1096     g = IJth(qBdata, i);
1097     p = IJth(pdata, i);
1098     #if defined(SUNDIALS_EXTENDED_PRECISION)
1099         fprintf(fid,"x%d(%d,1) = %Le; \n", myId, i[0]+1, x[0]);
1100         fprintf(fid,"y%d(%d,1) = %Le; \n", myId, i[1]+1, x[1]);
1101         fprintf(fid,"p%d(%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, p);
1102         fprintf(fid,"g%d(%d,%d) = %Le; \n", myId, i[1]+1, i[0]+1, g);
1103     #elif defined(SUNDIALS_DOUBLE_PRECISION)
1104         fprintf(fid,"x%d(%d,1) = %le; \n", myId, i[0]+1, x[0]);
1105         fprintf(fid,"y%d(%d,1) = %le; \n", myId, i[1]+1, x[1]);
1106         fprintf(fid,"p%d(%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, p);
1107         fprintf(fid,"g%d(%d,%d) = %le; \n", myId, i[1]+1, i[0]+1, g);
1108     #else
1109         fprintf(fid,"x%d(%d,1) = %e; \n", myId, i[0]+1, x[0]);
1110         fprintf(fid,"y%d(%d,1) = %e; \n", myId, i[1]+1, x[1]);
1111         fprintf(fid,"p%d(%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, p);
1112         fprintf(fid,"g%d(%d,%d) = %e; \n", myId, i[1]+1, i[0]+1, g);
1113     #endif
1114 #endif
1115 }
1116 }
1117 fclose(fid);
1118
1119 /* Write matlab driver */
1120
1121 if (myId == 0) {
1122
1123     fid = fopen("grad.m","w");
1124
1125     #ifndef USE3D
1126         fprintf(fid,"clear;\nfigure;\nhold on\n");
1127         fprintf(fid,"trans = 0.7;\n");
1128         fprintf(fid,"ecol = 'none';\n");
1129     #if defined(SUNDIALS_EXTENDED_PRECISION)
1130         fprintf(fid,"xp=[%Lf %Lf];\n",G1_X,G2_X);
1131         fprintf(fid,"yp=[%Lf %Lf];\n",G1_Y,G2_Y);
1132         fprintf(fid,"zp=[%Lf %Lf];\n",G1_Z,G2_Z);

```

```

1133 #else
1134     fprintf(fid,"xp=[%f %f];\n",G1_X,G2_X);
1135     fprintf(fid,"yp=[%f %f];\n",G1_Y,G2_Y);
1136     fprintf(fid,"zp=[%f %f];\n",G1_Z,G2_Z);
1137 #endif
1138     fprintf(fid,"ns = length(xp)*length(yp)*length(zp);\n");
1139
1140     for (ip=0; ip<d->npes; ip++) {
1141         fprintf(fid,"\ngrad%03d;\n",ip);
1142         fprintf(fid,"[X,Y,Z]=meshgrid(x%d,y%d,z%d);\n",ip,ip,ip);
1143         fprintf(fid,"s%d=slice(X,Y,Z,g%d,xp,yp,zp);\n",ip,ip);
1144         fprintf(fid,"for i = 1:ns\n");
1145         fprintf(fid,"    set(s%d(i),'FaceAlpha',trans);\n",ip);
1146         fprintf(fid,"    set(s%d(i),'EdgeColor',ecol);\n",ip);
1147         fprintf(fid,"end\n");
1148     }
1149
1150     fprintf(fid,"view(3)\n");
1151     fprintf(fid,"\nshading interp\naxis equal\n");
1152 #else
1153     fprintf(fid,"clear;\nfigure;\n");
1154     fprintf(fid,"trans = 0.7;\n");
1155     fprintf(fid,"ecol = 'none';\n");
1156
1157     for (ip=0; ip<d->npes; ip++) {
1158
1159         fprintf(fid,"\ngrad%03d;\n",ip);
1160
1161         fprintf(fid,"\nsubplot(1,2,1)\n");
1162         fprintf(fid,"s=surf(x%d,y%d,g%d);\n",ip,ip,ip);
1163         fprintf(fid,"set(s,'FaceAlpha',trans);\n");
1164         fprintf(fid,"set(s,'EdgeColor',ecol);\n");
1165         fprintf(fid,"hold on\n");
1166         fprintf(fid,"axis tight\n");
1167         fprintf(fid,"box on\n");
1168
1169         fprintf(fid,"\nsubplot(1,2,2)\n");
1170         fprintf(fid,"s=surf(x%d,y%d,p%d);\n",ip,ip,ip);
1171         fprintf(fid,"set(s,'CData',g%d);\n",ip);
1172         fprintf(fid,"set(s,'FaceAlpha',trans);\n");
1173         fprintf(fid,"set(s,'EdgeColor',ecol);\n");
1174         fprintf(fid,"hold on\n");
1175         fprintf(fid,"axis tight\n");
1176         fprintf(fid,"box on\n");
1177     }
1178 }
1179 #endif
1180     fclose(fid);
1181 }
1182 }

```


Approved for public release; further dissemination unlimited

