

# Example Programs for CVODES v2.2.0

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

April 2005

## **DISCLAIMER**

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

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Forward sensitivity analysis example problems</b>	<b>5</b>
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
<b>3</b>	<b>Adjoint sensitivity analysis example problems</b>	<b>25</b>
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
<b>4</b>	<b>Parallel tests</b>	<b>36</b>
	<b>References</b>	<b>38</b>
<b>A</b>	<b>Listing of <code>cvfnx.c</code></b>	<b>39</b>
<b>B</b>	<b>Listing of <code>cvfdx.c</code></b>	<b>49</b>
<b>C</b>	<b>Listing of <code>pvfkx.c</code></b>	<b>61</b>
<b>D</b>	<b>listing of <code>cvadx.c</code></b>	<b>86</b>
<b>E</b>	<b>Listing of <code>pvanx.c</code></b>	<b>97</b>
<b>F</b>	<b>Listing of <code>pvakx.c</code></b>	<b>110</b>



# 1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The first problem is the Van der Pol oscillator for which the Newton iteration cases use the following types of Jacobian approximations: (1) dense (user-supplied), (2) dense (difference quotient approximation), (3) diagonal approximation. The second

problem is a linear ODE with a banded lower triangular matrix derived from a 2-D advection PDE. In this case, the Newton iteration cases use the following types of Jacobian approximation: (1) banded (user-supplied), (2) banded (difference quotient approximation), (3) diagonal approximation.

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

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

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

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

Four different runs are made for this problem. The product 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. It also uses the user-supplied error weight function feature of CVODES.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

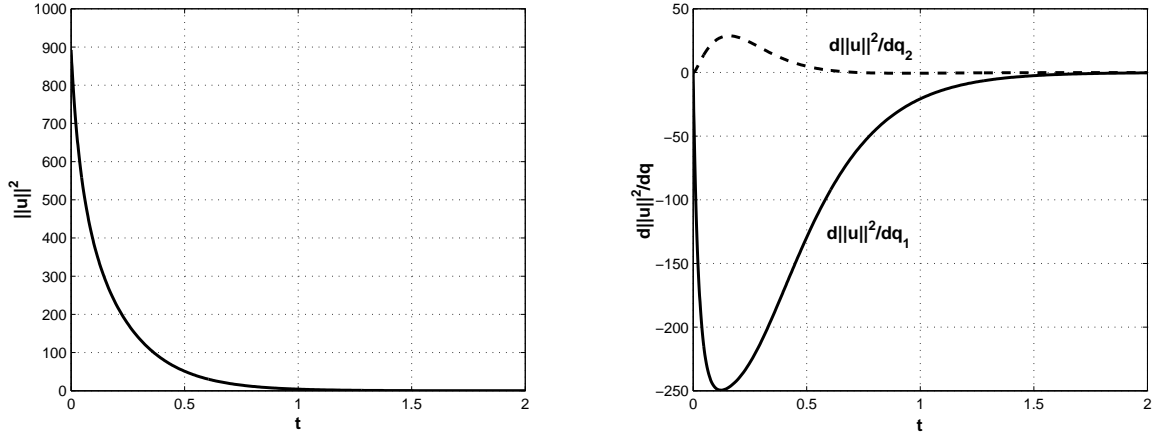


Figure 1: Results for the `cvfnx` example problem. The time evolution of the squared solution norm,  $\|u\|^2$ , is shown on the left. The figure on the right shows the evolution of the sensitivities of  $\|u\|^2$  with respect to the two problem parameters.

				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.

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

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

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

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

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

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

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

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

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

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

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

The user-supplied functions `f` for the right-hand side of the original ODEs and `Jac` for the system Jacobian are identical to those in `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. Similarly, the user-supplied function `ewt` is identical to that in `cvdx.c`. The user-supplied function `fS` computes the sensitivity right-hand side for the

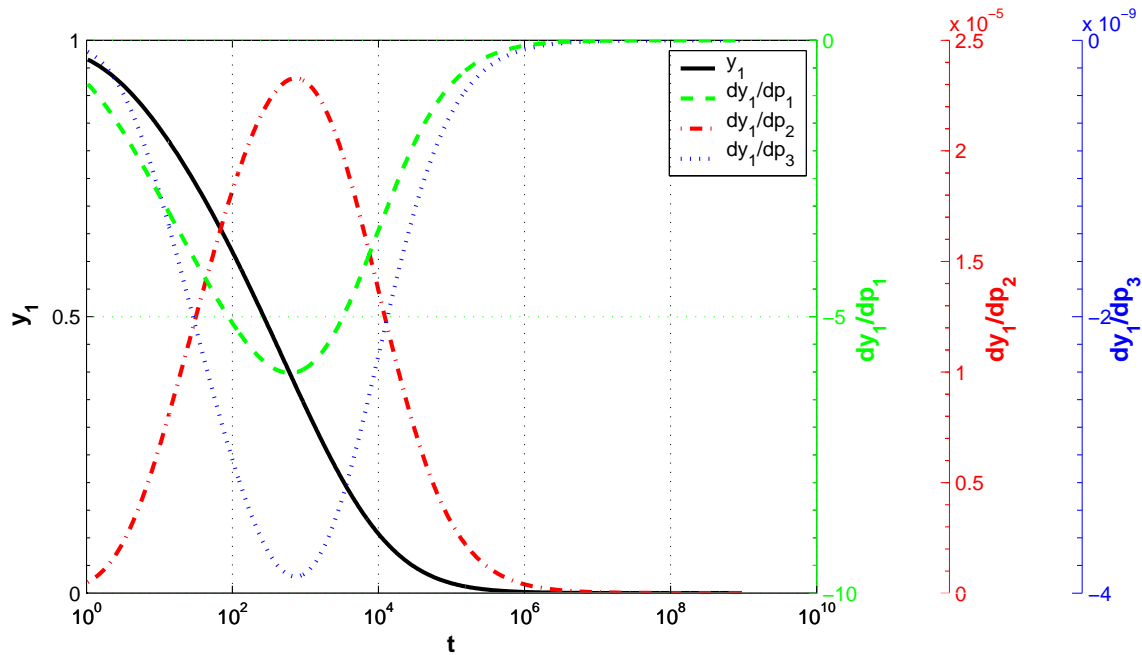


Figure 2: Results for the `cvfdx` example problem: time evolution of  $y_1$  and its sensitivities with respect to the three problem parameters.

$iS$ -th sensitivity equation.

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

```
% cvfdx -nosensi
```

if no sensitivity calculations are desired, or

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

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

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

cvfdx sample output						
3-species chemical kinetics problem						
Sensitivity: YES ( SIMULTANEOUS + FULL ERROR CONTROL )						
T	Q	H	NST	y1	y2	y3
4.000e-01	3	4.881e-02	115			
			Solution	9.8517e-01	3.3864e-05	1.4794e-02
			Sensitivity 1	-3.5595e-01	3.9025e-04	3.5556e-01



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

				Sensitivity 2	1.0425e-09	2.0859e-15	-1.0425e-09
				Sensitivity 3	-1.7366e-13	-6.9467e-19	1.7367e-13
-----							
4.000e+09	2	4.183e+08	836	Solution	5.1881e-07	2.0752e-12	1.0000e-00
				Sensitivity 1	-2.5907e-05	-5.1717e-11	2.5907e-05
				Sensitivity 2	1.0363e-10	2.0687e-16	-1.0363e-10
				Sensitivity 3	-1.7293e-14	-6.9174e-20	1.7293e-14
-----							
4.000e+10	2	3.799e+09	859	Solution	6.5181e-08	2.6072e-13	1.0000e-00
				Sensitivity 1	-2.4884e-06	-3.3032e-12	2.4884e-06
				Sensitivity 2	9.9534e-12	1.3213e-17	-9.9534e-12
				Sensitivity 3	-2.1727e-15	-8.6908e-21	2.1727e-15
-----							
Final Statistics							
nst	=	859					
nfe	=	1221					
netf	=	29	nsetups	=	142		
nni	=	1218	ncfn	=	4		
nfSe	=	3663	nfeS	=	0		
netfs	=	0	nsetupsS	=	0		
nniS	=	0	ncfnS	=	0		
njeD	=	24	nfeD	=	0		

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

----- cvfdx sample output -----							
3-species chemical kinetics problem							
Sensitivity: YES ( STAGGERED + PARTIAL ERROR CONTROL )							
=====							
T	Q	H	NST		y1	y2	y3
=====							
4.000e-01	3	1.205e-01	59	Solution	9.8517e-01	3.3863e-05	1.4797e-02
				Sensitivity 1	-3.5611e-01	3.9023e-04	3.5572e-01
				Sensitivity 2	9.4831e-08	-2.1325e-10	-9.4618e-08
				Sensitivity 3	-1.5733e-11	-5.2897e-13	1.6262e-11
-----							
4.000e+00	4	5.316e-01	74	Solution	9.0552e-01	2.2404e-05	9.4461e-02
				Sensitivity 1	-1.8761e+00	1.7922e-04	1.8760e+00
				Sensitivity 2	2.9612e-06	-5.8308e-10	-2.9606e-06
				Sensitivity 3	-4.9330e-10	-2.7624e-13	4.9357e-10
-----							

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

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

---

# Final Statistics

nst	=	579		
nfe	=	1379		
netf	=	25	nsetups	= 109
nni	=	797	ncfn	= 0
nfSe	=	2829	nfeS	= 0
netfs	=	0	nsetupsS	= 3
nniS	=	942	ncfnS	= 0
njeD	=	11	nfeD	= 0

### 2.3 An SPGMR parallel example with user preconditioner: 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  $\text{NPEX}$  processes in the  $x$  direction and  $\text{NPEY}$  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

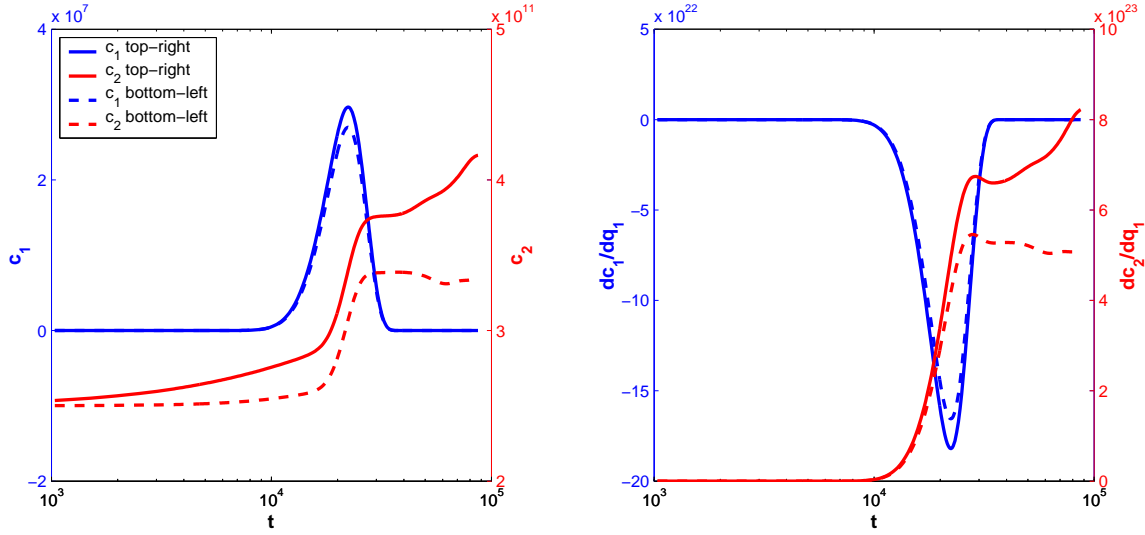


Figure 3: Results for the `pvfkn` example problem: time evolution of  $c_1$  and  $c_2$  at the bottom-left and top-right corners (left) and of their sensitivities with respect to  $q_1$ .

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	3.190e+01	408			
				Solution	1.0468e+04 2.5267e+11	1.1185e+04 2.6998e+11
				Sensitivity 1	-6.4201e+19 7.1178e+19	-6.8598e+19 7.6557e+19
				Sensitivity 2	-4.3853e+14 -2.4407e+18	-5.0065e+14 -2.7842e+18
1.440e+04	3	6.016e+01	722			
				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	4	1.654e+02	891			
				Solution	2.6650e+07 2.9928e+11	2.9308e+07 3.3134e+11
				Sensitivity 1	-1.6346e+23 3.8203e+23	-1.7976e+23 4.4991e+23
				Sensitivity 2	-7.6601e+18 -7.6459e+22	-9.4433e+18 -9.4502e+22
2.880e+04	2	3.838e+01	1163			
				Solution	8.7021e+06 3.3804e+11	9.6501e+06 3.7510e+11
				Sensitivity 1	-5.3375e+22 5.4487e+23	-5.9188e+22 6.7431e+23
				Sensitivity 2	-4.8855e+18 -1.7194e+23	-6.1040e+18 -2.1518e+23
3.600e+04	3	2.170e+01	1314			
				Solution	1.4040e+04 3.3868e+11	1.5609e+04 3.7652e+11
				Sensitivity 1	-8.6141e+19	-9.5762e+19

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



```

-----
7.920e+04  5  5.172e+02  1725
Solution      -4.1258e-15  -1.0944e-14
               3.3344e+11   4.1203e+11
-----
Sensitivity 1  -9.7653e+04  -2.5314e+05
               5.0731e+23   7.9960e+23
-----
Sensitivity 2   7.4987e-01   2.1107e+00
               -1.7747e+23  -2.7972e+23
-----

8.640e+04  5  5.172e+02  1739
Solution      1.7392e-18   2.4267e-18
               3.3518e+11   4.1625e+11
-----
Sensitivity 1  -2.0753e+03  -5.5491e+03
               5.1171e+23   8.2143e+23
-----
Sensitivity 2  -5.2491e-02  -1.6658e-01
               -1.7901e+23  -2.8736e+23
-----

Final Statistics

nst      = 1739

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

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

```

The output generated by `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	1.038e+02	308		
				Solution	8.7021e+06
					9.6500e+06
					3.3804e+11
					3.7510e+11
-----					
				Sensitivity 1	-5.3375e+22
					-5.9187e+22
					5.4487e+23
					6.7430e+23
-----					
				Sensitivity 2	-4.8855e+18
					-6.1040e+18
					-1.7194e+23
					-2.1518e+23
-----					
3.600e+04	4	7.257e+01	346		
				Solution	1.4040e+04
					1.5609e+04
					3.3868e+11
					3.7652e+11
-----					
				Sensitivity 1	-8.6140e+19
					-9.5761e+19
					5.2718e+23
					6.6029e+23
-----					
				Sensitivity 2	-8.4328e+15
					-1.0549e+16
					-1.8439e+23
					-2.3096e+23
-----					
4.320e+04	4	3.835e+02	407		
				Solution	-5.2385e-08
					5.9808e-07
					3.3823e+11
					3.8035e+11
-----					
				Sensitivity 1	8.8900e+08
					-2.8682e+09
					5.2753e+23
					6.7448e+23
-----					
				Sensitivity 2	2.6162e+07
					2.2623e+07
					-1.8454e+23
					-2.3595e+23
-----					
5.040e+04	5	4.386e+02	421		
				Solution	5.6769e-10
					4.8955e-09

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

-----			
Sensitivity 2			
	2.2251e+04	2.3887e+04	
	-1.7901e+23	-2.8736e+23	
-----			
Final Statistics			
nst	=	485	
nfe	=	1109	
netf	=	29	nsetups = 83
nni	=	621	ncfn = 0
nfSe	=	1226	nfeS = 2452
netfs	=	0	nsetupsS = 0
nniS	=	612	ncfnS = 0

### 3 Adjoint sensitivity analysis example problems

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

#### 3.1 A serial dense example: `cvadx`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Allocation for the memory block of the combined forward-backward problem is 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  $\lambda$  at the final time (0.0). A second serial vector `qB` of dimension `NP` is created and initialized to 0.0. This vector corresponds to the quadrature variables  $\xi$  whose values at  $t_0$  are the components of the gradient of  $G$  with respect to the problem parameters  $p$ . Following that, the program sets the relative and absolute tolerances for the backward integration.

The CVODES memory for the integration of the backward integration is created and allocated by the calls to the interface routines `CVodeCreateB` 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  $\xi$  should be included in the error test. Quadrature computation is initialized by calling `CVodeQuadMallocB` which specifies the right-hand side of the quadrature equations as `fQB`.

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

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

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

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

The output generated by `cvadx` is shown below.

```

----- cvadx sample output -----
Adjoint Sensitivity Example for Chemical Kinetics
-----

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

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

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

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

```

```
Re-initialize CVODES memory for backward run
Integrate backwards
```

```
-----
tB0:          4.0000e+07
dG/dp:        1.7341e+02  -5.0590e-04  8.4320e-08
lambda(t0):   8.4190e+00  1.6097e+01  1.6097e+01
-----
```

```
Free memory
```



### 3.2 A parallel nonstiff example: pvanx

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

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

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  $\delta u_0$  in  $u_0$ .

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

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

and

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

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

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

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

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

yielding

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

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

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

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

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

g(tf) = 2.129919e-02

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

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

```

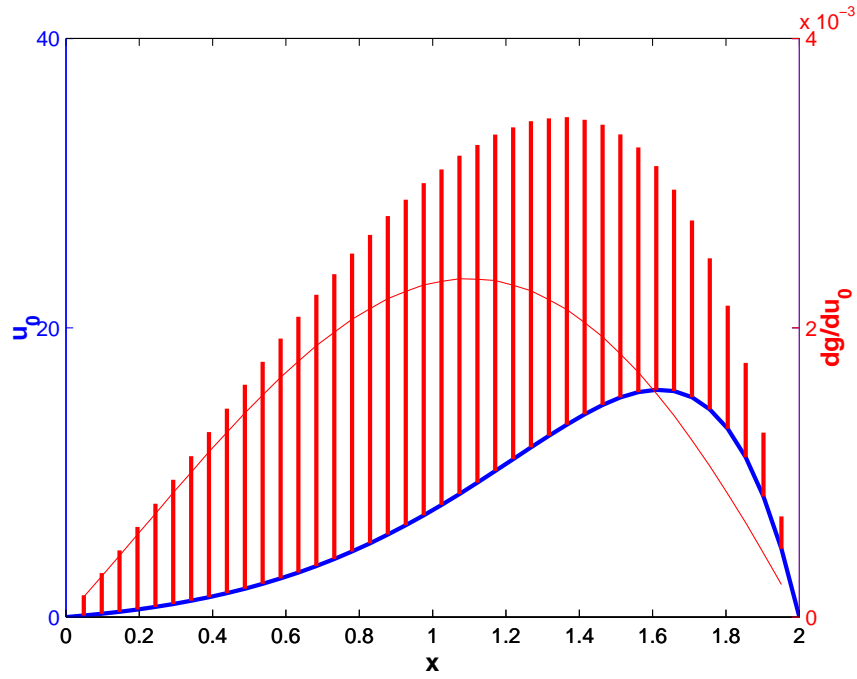


Figure 4: Results for the pvanx example problem. The gradient of  $g(t_f)$  with respect to the initial conditions  $u_0$  is shown superimposed over the values  $u_0$ .

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

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

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

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

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

Using adjoint sensitivity analysis, the gradient of

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

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  $\lambda$  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  $\Omega$  is discretized into  $m_i$  intervals, this leads to a system of ODEs of dimension  $N = \prod_1^d (m_i + 1)$ , with  $d = 2$ , or  $d = 3$ . The source term  $f$  is parameterized as a piecewise constant function and yielding  $N$  parameters in the problem. The nominal values of the source parameters 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

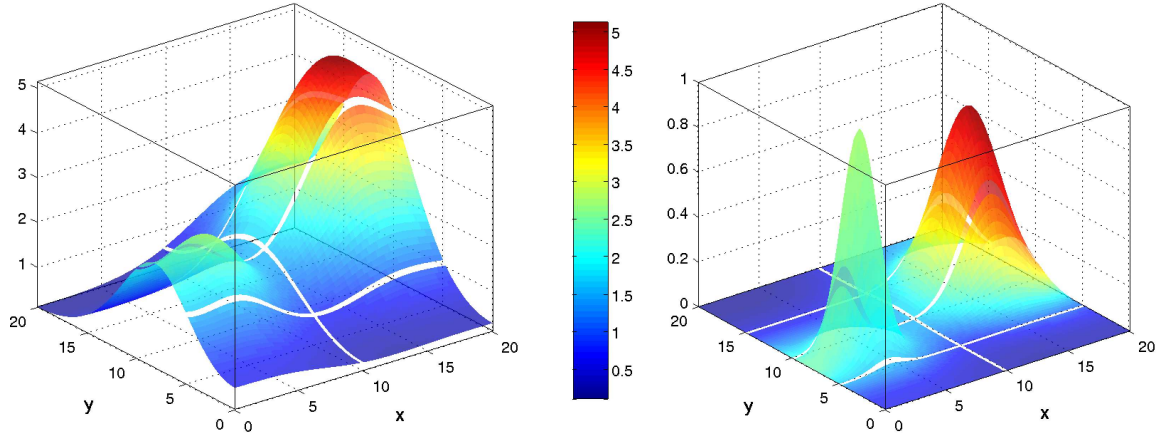


Figure 5: Results for the `pvakx` example problem in 2D. The gradient with respect to the source parameters is pictured on the left. On the right, the gradient was color coded and superimposed over the nominal value of the source parameters.

`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 `CVBBDSPrecAlloc` and `CVBBDSPrecAllocB`, 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  $80 \times 80$  grid and  $2 \times 4 = 8$  processes are shown in Fig. 5. Results in 3-D <sup>†</sup>, 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  =  8746      leniw =   212
llrw   =  8656      lliw  =    80

```

<sup>†</sup>The name of executable for the 3-D version is `pvakx3D`.

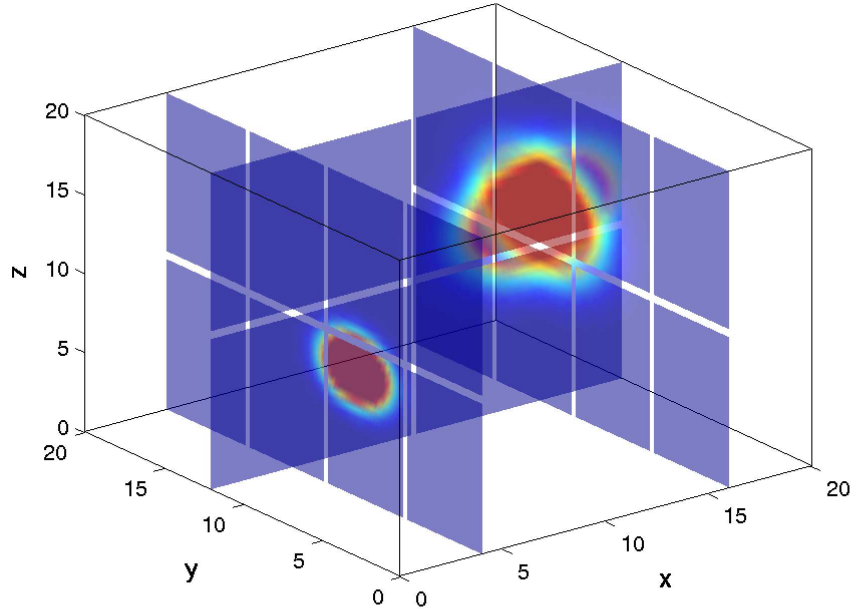


Figure 6: Results for the `pvakx` example problem in 3D. Nominal values of the source parameters.

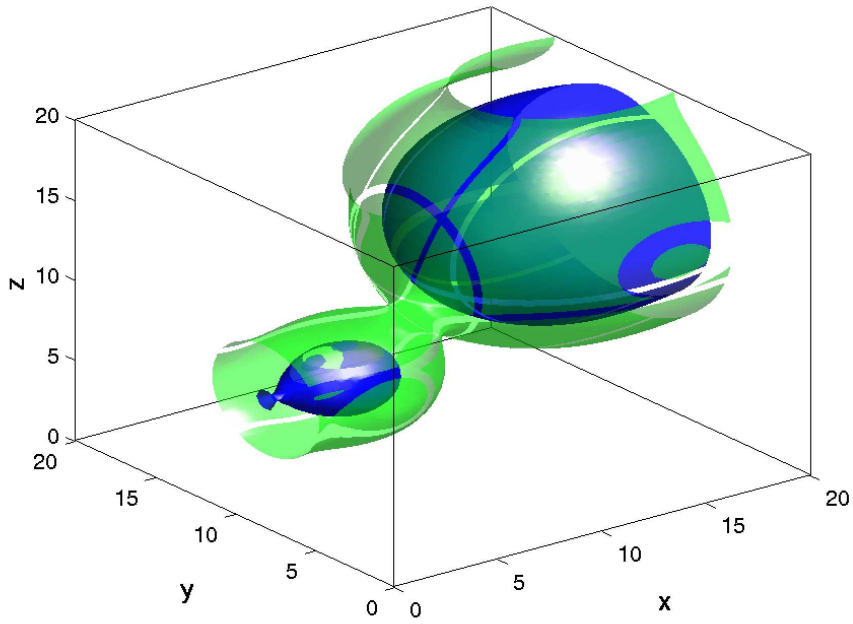


Figure 7: Results for the `pvakx` example problem in 3D. Two isosurfaces of the gradient with respect to the source parameters. They correspond to values of 0.25 (green) and 0.4 (blue).

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

Begin backward integration... done.

Final Statistics..

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

## 4 Parallel tests

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

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

where

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

$K_h$ ,  $K_v$ ,  $v$ ,  $q_1$ ,  $q_2$ , 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.



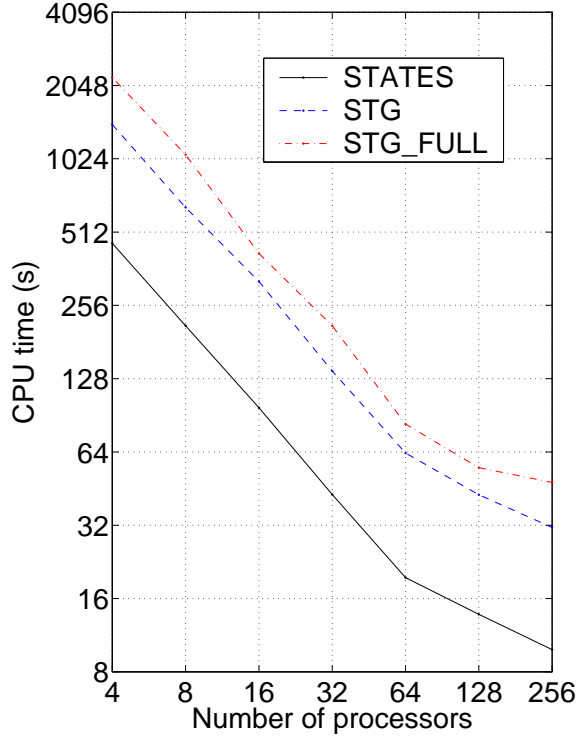


Figure 8: Speedup results for the integration of the state equations only (solid line and column 'STATES'), staggered sensitivity analysis without error control on the sensitivity variables (dashed line and column 'STG'), and staggered sensitivity analysis with full error control (dotted line and column '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: 1.17.2.2 $
4  * $Date: 2005/04/01 21:55:27 $
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 = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
143     if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);
144
145     flag = CVodeSetFdata(cvode_mem, data);
146     if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);
147
148     /* Allocate CVODES memory */
149     flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
150     if(check_flag(&flag, "CVodeMalloc", 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         plist = (int *) malloc(NS * sizeof(int));
158         if(check_flag((void *)plist, "malloc", 2)) return(1);
159         for(is=0; is<NS; is++) plist[is] = is+1;
160

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

## B Listing of cvfdx.c

```

1  /*
2  * -----
3  * $Revision: 1.21.2.4 $
4  * $Date: 2005/04/07 15:58:52 $
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 reatype */
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 #include "sundialsmath.h"      /* definition of ABS */
57
58 /* Accessor macros */
59
60 #define Ith(v,i)    NV_Ith_S(v,i-1)      /* i-th vector component i=1..NEQ */
61 #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j=1..NEQ */
62
63 /* Problem Constants */
64
65 #define NEQ    3                /* number of equations */
66 #define Y1     RCONST(1.0)      /* initial y components */
67 #define Y2     RCONST(0.0)
68 #define Y3     RCONST(0.0)
69 #define RTOL   RCONST(1e-4)     /* scalar relative tolerance */
70 #define ATOL1  RCONST(1e-8)     /* vector absolute tolerance components */
71 #define ATOL2  RCONST(1e-14)
72 #define ATOL3  RCONST(1e-6)
73 #define T0     RCONST(0.0)      /* initial time */
74 #define T1     RCONST(0.4)      /* first output time */
75 #define TMULT  RCONST(10.0)     /* output time factor */
76 #define NOUT   12               /* number of output times */
77
78 #define NP     3                /* number of problem parameters */
79 #define NS     3                /* number of sensitivities computed */
80
81 #define ZERO   RCONST(0.0)
82
83 /* Type : UserData */
84
85 typedef struct {
86     realtype p[3];              /* problem parameters */
87 } *UserData;
88
89 /* Prototypes of functions by CVODES */
90
91 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
92
93 static void Jac(long int N, DenseMat J, realtype t,
94               N_Vector y, N_Vector fy, void *jac_data,
95               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
96
97 static void fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
98               int iS, N_Vector yS, N_Vector ySdot,
99               void *fS_data, N_Vector tmp1, N_Vector tmp2);
100
101 static int ewt(N_Vector y, N_Vector w, void *e_data);
102
103 /* Prototypes of private functions */
104
105 static void ProcessArgs(int argc, char *argv[],
106                       boolean *sens, int *sens_meth,

```

```

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

```

```

161  /* Allocate space for CVODES */
162  flag = CNodeMalloc(cvode_mem, f, T0, y, CV_WF, 0.0, NULL);
163  if (check_flag(&flag, "CNodeMalloc", 1)) return(1);
164
165  /* Use private function to compute error weights */
166  flag = CNodeSetEwtFn(cvode_mem, ewt, NULL);
167  if (check_flag(&flag, "CNodeSetEwtFn", 1)) return(1);
168
169  /* Attach user data */
170  flag = CNodeSetFdata(cvode_mem, data);
171  if (check_flag(&flag, "CNodeSetFdata", 1)) return(1);
172
173  /* Attach linear solver */
174  flag = CVDense(cvode_mem, NEQ);
175  if (check_flag(&flag, "CVDense", 1)) return(1);
176
177  flag = CVDenseSetJacFn(cvode_mem, Jac, data);
178  if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
179
180  printf("\n3-species chemical kinetics problem\n");
181
182  /* Sensitivity-related settings */
183  if (sensi) {
184
185      pbar[0] = data->p[0];
186      pbar[1] = data->p[1];
187      pbar[2] = data->p[2];
188
189      yS = N_VNewVectorArray_Serial(NS, NEQ);
190      if (check_flag((void *)yS, "N_VNewVectorArray_Serial", 0)) return(1);
191      for (is=0;is<NS;is++) N_VConst(ZERO, yS[is]);
192
193      flag = CNodeSensMalloc(cvode_mem, NS, sensi_meth, yS);
194      if (check_flag(&flag, "CNodeSensMalloc", 1)) return(1);
195
196      flag = CNodeSetSensRhs1Fn(cvode_mem, fS);
197      if (check_flag(&flag, "CNodeSetSensRhs1Fn", 1)) return(1);
198      flag = CNodeSetSensErrCon(cvode_mem, err_con);
199      if (check_flag(&flag, "CNodeSetSensFdata", 1)) return(1);
200      flag = CNodeSetSensFdata(cvode_mem, data);
201      if (check_flag(&flag, "CNodeSetSensFdata", 1)) return(1);
202      flag = CNodeSetSensParams(cvode_mem, NULL, pbar, NULL);
203      if (check_flag(&flag, "CNodeSetSensParams", 1)) return(1);
204
205      printf("Sensitivity: YES ");
206      if(sensi_meth == CV_SIMULTANEOUS)
207          printf("( SIMULTANEOUS +");
208      else
209          if(sensi_meth == CV_STAGGERED) printf("( STAGGERED +");
210          else printf("( STAGGERED1 +");
211      if(err_con) printf(" FULL ERROR CONTROL )");
212      else printf(" PARTIAL ERROR CONTROL )");
213
214  } else {

```



```

215
216     printf("Sensitivity: NO ");
217
218 }
219
220 /* In loop over output points, call CVode, print results, test for error */
221
222 printf("\n\n");
223 printf("=====");
224 printf("=====\n");
225 printf("      T      Q      H      NST      y1");
226 printf("      y2      y3      \n");
227 printf("=====");
228 printf("=====\n");
229
230 for (iout=1, tout=T1; iout <= NOUT; iout++, tout *= TMULT) {
231
232     flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
233     if (check_flag(&flag, "CVode", 1)) break;
234
235     PrintOutput(cvode_mem, t, y);
236
237     if (sensi) {
238         flag = CVodeGetSens(cvode_mem, t, yS);
239         if (check_flag(&flag, "CVodeGetSens", 1)) break;
240         PrintOutputS(yS);
241     }
242     printf("-----");
243     printf("-----\n");
244
245 }
246
247 /* Print final statistics */
248 PrintFinalStats(cvode_mem, sensi);
249
250 /* Free memory */
251
252 N_VDestroy_Serial(y);          /* Free y vector */
253 if (sensi) {
254     N_VDestroyVectorArray_Serial(yS, NS); /* Free yS vector */
255 }
256 free(data);                    /* Free user data */
257 CVodeFree(cvode_mem);          /* Free CVODES memory */
258
259 return(0);
260 }
261
262 /*
263  *-----
264  * FUNCTIONS CALLED BY CVODES
265  *-----
266  */
267
268 /*

```

```

269  * f routine. Compute f(t,y).
270  */
271
272 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
273 {
274     realtype y1, y2, y3, yd1, yd3;
275     UserData data;
276     realtype p1, p2, p3;
277
278     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
279     data = (UserData) f_data;
280     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
281
282     yd1 = Ith(ydot,1) = -p1*y1 + p2*y2*y3;
283     yd3 = Ith(ydot,3) = p3*y2*y2;
284     Ith(ydot,2) = -yd1 - yd3;
285 }
286
287
288 /*
289  * Jacobian routine. Compute J(t,y).
290  */
291
292 static void Jac(long int N, DenseMat J, realtype t,
293                N_Vector y, N_Vector fy, void *jac_data,
294                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
295 {
296     realtype y1, y2, y3;
297     UserData data;
298     realtype p1, p2, p3;
299
300     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
301     data = (UserData) jac_data;
302     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
303
304     IJth(J,1,1) = -p1;  IJth(J,1,2) = p2*y3;          IJth(J,1,3) = p2*y2;
305     IJth(J,2,1) =  p1;  IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
306     IJth(J,3,2) = 2*p3*y2;
307 }
308
309 /*
310  * fS routine. Compute sensitivity r.h.s.
311  */
312
313 static void fS(int Ns, realtype t, N_Vector y, N_Vector ydot,
314               int iS, N_Vector yS, N_Vector ySdot,
315               void *fS_data, N_Vector tmp1, N_Vector tmp2)
316 {
317     UserData data;
318     realtype p1, p2, p3;
319     realtype y1, y2, y3;
320     realtype s1, s2, s3;
321     realtype sd1, sd2, sd3;
322

```

```

323 data = (UserData) fS_data;
324 p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
325
326 y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
327 s1 = Ith(yS,1); s2 = Ith(yS,2); s3 = Ith(yS,3);
328
329 sd1 = -p1*s1 + p2*y3*s2 + p2*y2*s3;
330 sd3 = 2*p3*y2*s2;
331 sd2 = -sd1-sd3;
332
333 switch (iS) {
334 case 0:
335     sd1 += -y1;
336     sd2 += y1;
337     break;
338 case 1:
339     sd1 += y2*y3;
340     sd2 += -y2*y3;
341     break;
342 case 2:
343     sd2 += -y2*y2;
344     sd3 += y2*y2;
345     break;
346 }
347
348 Ith(ySdot,1) = sd1;
349 Ith(ySdot,2) = sd2;
350 Ith(ySdot,3) = sd3;
351 }
352
353 /*
354  * EwtSet function. Computes the error weights at the current solution.
355  */
356
357 static int ewt(N_Vector y, N_Vector w, void *e_data)
358 {
359     int i;
360     realtype yy, ww, rtol, atol[3];
361
362     rtol = RTOL;
363     atol[0] = ATOL1;
364     atol[1] = ATOL2;
365     atol[2] = ATOL3;
366
367     for (i=1; i<=3; i++) {
368         yy = Ith(y,i);
369         ww = rtol * ABS(yy) + atol[i-1];
370         if (ww <= 0.0) return (-1);
371         Ith(w,i) = 1.0/ww;
372     }
373
374     return(0);
375 }
376

```

```

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

```

```

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

```

```

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

```

```

539     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
540     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
541
542     if (sensi) {
543         flag = CVodeGetNumSensRhsEvals(cvode_mem, &nfSe);
544         check_flag(&flag, "CVodeGetNumSensRhsEvals", 1);
545         flag = CVodeGetNumRhsEvalsSens(cvode_mem, &nfeS);
546         check_flag(&flag, "CVodeGetNumRhsEvalsSens", 1);
547         flag = CVodeGetNumSensLinSolvSetups(cvode_mem, &nsetupsS);
548         check_flag(&flag, "CVodeGetNumSensLinSolvSetups", 1);
549         flag = CVodeGetNumSensErrTestFails(cvode_mem, &netfS);
550         check_flag(&flag, "CVodeGetNumSensErrTestFails", 1);
551         flag = CVodeGetNumSensNonlinSolvIters(cvode_mem, &nniS);
552         check_flag(&flag, "CVodeGetNumSensNonlinSolvIters", 1);
553         flag = CVodeGetNumSensNonlinSolvConvFails(cvode_mem, &ncfnS);
554         check_flag(&flag, "CVodeGetNumSensNonlinSolvConvFails", 1);
555     }
556
557     flag = CVDenseGetNumJacEvals(cvode_mem, &njeD);
558     check_flag(&flag, "CVDenseGetNumJacEvals", 1);
559     flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeD);
560     check_flag(&flag, "CVDenseGetNumRhsEvals", 1);
561
562     printf("\nFinal Statistics\n\n");
563     printf("nst      = %5ld\n\n", nst);
564     printf("nfe      = %5ld\n", nfe);
565     printf("netf     = %5ld      nsetups = %5ld\n", netf, nsetups);
566     printf("nni      = %5ld      ncfn     = %5ld\n", nni, ncfn);
567
568     if(sensi) {
569         printf("\n");
570         printf("nfSe     = %5ld      nfeS     = %5ld\n", nfSe, nfeS);
571         printf("netfs    = %5ld      nsetupsS = %5ld\n", netfS, nsetupsS);
572         printf("nniS     = %5ld      ncfnS     = %5ld\n", nniS, ncfnS);
573     }
574
575     printf("\n");
576     printf("njeD     = %5ld      nfeD     = %5ld\n", njeD, nfeD);
577
578 }
579
580 /*
581  * Check function return value.
582  *   opt == 0 means SUNDIALS function allocates memory so check if
583  *   returned NULL pointer
584  *   opt == 1 means SUNDIALS function returns a flag so check if
585  *   flag >= 0
586  *   opt == 2 means function allocates memory so check if returned
587  *   NULL pointer
588  */
589
590 static int check_flag(void *flagvalue, char *funcname, int opt)
591 {
592     int *errflag;

```

```

593
594 /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
595 if (opt == 0 && flagvalue == NULL) {
596     fprintf(stderr,
597         "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
598         funcname);
599     return(1); }
600
601 /* Check if flag < 0 */
602 else if (opt == 1) {
603     errflag = (int *) flagvalue;
604     if (*errflag < 0) {
605         fprintf(stderr,
606             "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
607             funcname, *errflag);
608         return(1); }}
609
610 /* Check if function returned NULL pointer - no memory allocated */
611 else if (opt == 2 && flagvalue == NULL) {
612     fprintf(stderr,
613         "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
614         funcname);
615     return(1); }
616
617 return(0);
618 }

```



## C Listing of pvfkn.c

```

1  /*
2  * -----
3  * $Revision: 1.20.2.3 $
4  * $Date: 2005/04/06 23:34:05 $
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 predata;
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 = CVSpgmrSetPreconditioner(cnode_mem, Precond, PSolve, predata);
285 if (check_flag(&flag, "CVSpgmrSetPreconditioner", 1, my_pe)) MPI_Abort(comm, 1);
286
287 if(my_pe == 0)
288     printf("\n2-species diurnal advection-diffusion problem\n");
289
290 /* Sensitivity-related settings */
291 if( sensi) {
292
293     plist = (int *) malloc(NS * sizeof(int));
294     if (check_flag((void *)plist, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
295     for (is=0; is<NS; is++) plist[is] = is+1;
296
297     pbar = (realtype *) malloc(NS*sizeof(realtype));
298     if (check_flag((void *)pbar, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
299     for (is=0; is<NS; is++) pbar[is] = data->p[plist[is]-1];
300
301     uS = N_VNewVectorArray_Parallel(NS, comm, local_N, neq);
302     if (check_flag((void *)uS, "N_VNewVectorArray_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
303     for (is = 0; is < NS; is++)
304         N_VConst(ZERO, uS[is]);
305
306     flag = CNodeSensMalloc(cnode_mem, NS, sensi_meth, uS);
307     if (check_flag(&flag, "CNodeSensMalloc", 1, my_pe)) MPI_Abort(comm, 1);
308
309     flag = CNodeSetSensErrCon(cnode_mem, err_con);
310     if (check_flag(&flag, "CNodeSetSensErrCon", 1, my_pe)) MPI_Abort(comm, 1);
311
312     flag = CNodeSetSensRho(cnode_mem, ZERO);
313     if (check_flag(&flag, "CNodeSetSensRho", 1, my_pe)) MPI_Abort(comm, 1);
314
315     flag = CNodeSetSensParams(cnode_mem, data->p, pbar, plist);
316     if (check_flag(&flag, "CNodeSetSensParams", 1, my_pe)) MPI_Abort(comm, 1);
317
318     if(my_pe == 0) {
319         printf("Sensitivity: YES ");
320         if(sensi_meth == CV_SIMULTANEOUS)
321             printf("( SIMULTANEOUS +");
322         else

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

809     }
810
811     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
812     if (isubx != NPEX-1) {
813         MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
814                 my_pe+1, 0, comm, &request[3]);
815     }
816 }
817
818 /*
819  * Routine to finish receiving boundary data from neighboring PEs.
820  * Notes:
821  * 1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
822  *    passed to both the BRecvPost and BRecvWait functions, and should not
823  *    be manipulated between the two calls.
824  * 2) request should have 4 entries, and should be passed in both calls also.
825  */
826
827 static void BRecvWait(MPI_Request request[], long int isubx, long int isuby,
828                      long int dsizex, realtype uext[], realtype buffer[])
829 {
830     int i, ly;
831     long int dsizex2, offsetue, offsetbuf;
832     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
833     MPI_Status status;
834
835     dsizex2 = dsizex + 2*NVARs;
836
837     /* If isuby > 0, receive data for bottom x-line of uext */
838     if (isuby != 0)
839         MPI_Wait(&request[0], &status);
840
841     /* If isuby < NPEY-1, receive data for top x-line of uext */
842     if (isuby != NPEY-1)
843         MPI_Wait(&request[1], &status);
844
845     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
846     if (isubx != 0) {
847         MPI_Wait(&request[2], &status);
848
849         /* Copy the buffer to uext */
850         for (ly = 0; ly < MYSUB; ly++) {
851             offsetbuf = ly*NVARs;
852             offsetue = (ly+1)*dsizex2;
853             for (i = 0; i < NVARs; i++)
854                 uext[offsetue+i] = bufleft[offsetbuf+i];
855         }
856     }
857
858     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
859     if (isubx != NPEX-1) {
860         MPI_Wait(&request[3], &status);
861
862         /* Copy the buffer to uext */

```



```

863     for (ly = 0; ly < MYSUB; ly++) {
864         offsetbuf = ly*NVAR;
865         offsetue = (ly+2)*dsizex2 - NVAR;
866         for (i = 0; i < NVAR; i++)
867             uext[offsetue+i] = bufright[offsetbuf+i];
868     }
869 }
870
871 }
872
873 /*
874  * ucomm routine. This routine performs all communication
875  * between processors of data needed to calculate f.
876  */
877
878 static void ucomm(realtype t, N_Vector u, UserData data)
879 {
880     realtype *udata, *uext, buffer[2*NVAR*MYSUB];
881     MPI_Comm comm;
882     int my_pe;
883     long int isubx, isuby, nvxsub, nvmysub;
884     MPI_Request request[4];
885
886     udata = NV_DATA_P(u);
887
888     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
889     comm = data->comm; my_pe = data->my_pe;
890     isubx = data->isubx; isuby = data->isuby;
891     nvxsub = data->nvxsub;
892     nvmysub = NVAR*MYSUB;
893     uext = data->uext;
894
895     /* Start receiving boundary data from neighboring PEs */
896     BRecvPost(comm, request, my_pe, isubx, isuby, nvxsub, nvmysub, uext, buffer);
897
898     /* Send data from boundary of local grid to neighboring PEs */
899     BSend(comm, my_pe, isubx, isuby, nvxsub, nvmysub, udata);
900
901     /* Finish receiving boundary data from neighboring PEs */
902     BRecvWait(request, isubx, isuby, nvxsub, uext, buffer);
903 }
904
905 /*
906  * fcalc routine. Compute f(t,y). This routine assumes that communication
907  * between processors of data needed to calculate f has already been done,
908  * and this data is in the work array uext.
909  */
910
911 static void fcalc(realtype t, realtype udata[], realtype dudata[], UserData data)
912 {
913     realtype *uext;
914     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
915     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
916     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;

```

```

917  realtype q4coef, dely, verdco, hordco, horaco;
918  int i, lx, ly, jx, jy;
919  long int isubx, isuby, nvmsub, nvmsub2, offsetu, offsetue;
920  realtype Q1, Q2, C3, A3, A4, KH, VEL, KVO;
921
922  /* Get subgrid indices, data sizes, extended work array uext */
923  isubx = data->isubx;  isuby = data->isuby;
924  nvmsub = data->nvmsub; nvmsub2 = data->nvmsub2;
925  uext = data->uext;
926
927  /* Load problem coefficients and parameters */
928  Q1 = data->p[0];
929  Q2 = data->p[1];
930  C3 = data->p[2];
931  A3 = data->p[3];
932  A4 = data->p[4];
933  KH = data->p[5];
934  VEL = data->p[6];
935  KVO = data->p[7];
936
937  /* Copy local segment of u vector into the working extended array uext */
938  offsetu = 0;
939  offsetue = nvmsub2 + NVAR;
940  for (ly = 0; ly < MYSUB; ly++) {
941      for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
942      offsetu = offsetu + nvmsub;
943      offsetue = offsetue + nvmsub2;
944  }
945
946  /* To facilitate homogeneous Neumann boundary conditions, when this is
947  a boundary PE, copy data from the first interior mesh line of u to uext */
948
949  /* If isuby = 0, copy x-line 2 of u to uext */
950  if (isuby == 0) {
951      for (i = 0; i < nvmsub; i++) uext[NVAR+i] = udata[nvmsub+i];
952  }
953
954  /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
955  if (isuby == NPEY-1) {
956      offsetu = (MYSUB-2)*nvmsub;
957      offsetue = (MYSUB+1)*nvmsub2 + NVAR;
958      for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
959  }
960
961  /* If isubx = 0, copy y-line 2 of u to uext */
962  if (isubx == 0) {
963      for (ly = 0; ly < MYSUB; ly++) {
964          offsetu = ly*nvmsub + NVAR;
965          offsetue = (ly+1)*nvmsub2;
966          for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
967      }
968  }
969
970  /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */

```

```

971  if (isubx == NPEX-1) {
972      for (ly = 0; ly < MYSUB; ly++) {
973          offsetu = (ly+1)*nvmxsub - 2*NVAR;
974          offsetue = (ly+2)*nvmxsub2 - NVAR;
975          for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
976      }
977  }
978
979  /* Make local copies of problem variables, for efficiency */
980  dely = data->dy;
981  verdco = data->vdco;
982  hordco = data->hdco;
983  horaco = data->haco;
984
985  /* Set diurnal rate coefficients as functions of t, and save q4 in
986  data block for use by preconditioner evaluation routine */
987  s = sin((data->om)*t);
988  if (s > ZERO) {
989      q3 = exp(-A3/s);
990      q4coef = exp(-A4/s);
991  } else {
992      q3 = ZERO;
993      q4coef = ZERO;
994  }
995  data->q4 = q4coef;
996
997  /* Loop over all grid points in local subgrid */
998  for (ly = 0; ly < MYSUB; ly++) {
999      jy = ly + isub*MYSUB;
1000
1001      /* Set vertical diffusion coefficients at jy +/- 1/2 */
1002      ydn = YMIN + (jy - .5)*dely;
1003      yup = ydn + dely;
1004      cydn = verdco*exp(RCONST(0.2)*ydn);
1005      cyup = verdco*exp(RCONST(0.2)*yup);
1006      for (lx = 0; lx < MXSUB; lx++) {
1007          jx = lx + isub*MXSUB;
1008
1009          /* Extract c1 and c2, and set kinetic rate terms */
1010          offsetue = (lx+1)*NVAR + (ly+1)*nvmxsub2;
1011          c1 = uext[offsetue];
1012          c2 = uext[offsetue+1];
1013          qq1 = Q1*c1*C3;
1014          qq2 = Q2*c1*c2;
1015          qq3 = q3*C3;
1016          qq4 = q4coef*c2;
1017          rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
1018          rkin2 = qq1 - qq2 - qq4;
1019
1020          /* Set vertical diffusion terms */
1021          c1dn = uext[offsetue-nvmxsub2];
1022          c2dn = uext[offsetue-nvmxsub2+1];
1023          c1up = uext[offsetue+nvmxsub2];
1024          c2up = uext[offsetue+nvmxsub2+1];

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```
1295         id, funcname);
1296     return(1); }
1297
1298     return(0);
1299 }
```

## D listing of cvadx.c

```
1  /*
2  * -----
3  * $Revision: 1.18.2.7 $
4  * $Date: 2005/04/28 20:06:31 $
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  #include "sundialsmath.h"
63
64  /* Accessor macros */
65
66  #define Ith(v,i)    NV_Ith_S(v,i-1)      /* i-th vector component i= 1..NEQ */
67  #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* (i,j)-th matrix component i,j = 1..NEQ */
68
69  /* Problem Constants */
70
71  #define NEQ        3                /* number of equations */
72
73  #define RTOL        RCONST(1e-6) /* scalar relative tolerance */
74
75  #define ATOL1        RCONST(1e-8) /* vector absolute tolerance components */
76  #define ATOL2        RCONST(1e-14)
77  #define ATOL3        RCONST(1e-6)
78
79  #define ATOLL        RCONST(1e-5) /* absolute tolerance for adjoint vars. */
80  #define ATOLq        RCONST(1e-6) /* absolute tolerance for quadratures */
81
82  #define T0          RCONST(0.0) /* initial time */
83  #define TOUT        RCONST(4e7) /* final time */
84
85  #define TB1          RCONST(4e7) /* starting point for adjoint problem */
86  #define TB2          RCONST(50.0) /* starting point for adjoint problem */
87
88  #define STEPS        150          /* number of steps between check points */
89
90  #define NP           3            /* number of problem parameters */
91
92  #define ZERO          RCONST(0.0)
93
94
95  /* Type : UserData */
96
97  typedef struct {
98      realtype p[3];
99  } *UserData;
100
101  /* Prototypes of user-supplied functions */
102
103  static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
104  static void Jac(long int N, DenseMat J, realtype t,
105                  N_Vector y, N_Vector fy, void *jac_data,
106                  N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

```

```

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

```

```

161  /* User data structure */
162  data = (UserData) malloc(sizeof *data);
163  if (check_flag((void *)data, "malloc", 2)) return(1);
164  data->p[0] = RCONST(0.04);
165  data->p[1] = RCONST(1.0e4);
166  data->p[2] = RCONST(3.0e7);
167
168  /* Initialize y */
169  y = N_VNew_Serial(NEQ);
170  if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
171  Ith(y,1) = RCONST(1.0);
172  Ith(y,2) = ZERO;
173  Ith(y,3) = ZERO;
174
175  /* Initialize q */
176  q = N_VNew_Serial(1);
177  if (check_flag((void *)q, "N_VNew_Serial", 0)) return(1);
178  Ith(q,1) = ZERO;
179
180  /* Set the scalar relative and absolute tolerances reltolQ and abstolQ */
181  reltolQ = RTOL;
182  abstolQ = ATOLq;
183
184  /* Create and allocate CVODES memory for forward run */
185  printf("Create and allocate CVODES memory for forward runs\n");
186
187  cvode_mem = CNodeCreate(CV_BDF, CV_NEWTON);
188  if (check_flag((void *)cvode_mem, "CNodeCreate", 0)) return(1);
189
190  flag = CNodeMalloc(cvode_mem, f, T0, y, CV_WF, 0.0, NULL);
191  if (check_flag(&flag, "CNodeMalloc", 1)) return(1);
192
193  flag = CNodeSetEwtFn(cvode_mem, ewt, NULL);
194  if (check_flag(&flag, "CNodeSetEwtFn", 1)) return(1);
195
196  flag = CNodeSetFdata(cvode_mem, data);
197  if (check_flag(&flag, "CNodeSetFdata", 1)) return(1);
198
199  flag = CVDense(cvode_mem, NEQ);
200  if (check_flag(&flag, "CVDense", 1)) return(1);
201
202  flag = CVDenseSetJacFn(cvode_mem, Jac, data);
203  if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
204
205  flag = CNodeQuadMalloc(cvode_mem, fQ, q);
206  if (check_flag(&flag, "CNodeQuadMalloc", 1)) return(1);
207
208  flag = CNodeSetQuadFdata(cvode_mem, data);
209  if (check_flag(&flag, "CNodeSetQuadFdata", 1)) return(1);
210
211  flag = CNodeSetQuadErrCon(cvode_mem, TRUE, CV_SS, reltolQ, &abstolQ);
212  if (check_flag(&flag, "CNodeSetQuadErrCon", 1)) return(1);
213
214  /* Allocate global memory */

```

```

215     printf("Allocate global memory\n");
216
217     cvadj_mem = CVadjMalloc(cvode_mem, STEPS);
218     if (check_flag((void *)cvadj_mem, "CVadjMalloc", 0)) return(1);
219
220     /* Perform forward run */
221     printf("Forward integration ... ");
222
223     flag = CVodeF(cvadj_mem, TOUT, y, &time, CV_NORMAL, &ncheck);
224     if (check_flag(&flag, "CVodeF", 1)) return(1);
225
226     flag = CVodeGetQuad(cvode_mem, TOUT, q);
227     if (check_flag(&flag, "CVodeGetQuad", 1)) return(1);
228
229     #if defined(SUNDIALS_EXTENDED_PRECISION)
230         printf("done. ncheck = %d    G: %12.4Le \n", ncheck, Ith(q,1));
231     #elif defined(SUNDIALS_DOUBLE_PRECISION)
232         printf("done. ncheck = %d    G: %12.4le \n", ncheck, Ith(q,1));
233     #else
234         printf("done. ncheck = %d    G: %12.4e \n", ncheck, Ith(q,1));
235     #endif
236
237     /* Initialize yB */
238     yB = N_VNew_Serial(NEQ);
239     if (check_flag((void *)yB, "N_VNew_Serial", 0)) return(1);
240     Ith(yB,1) = ZERO;
241     Ith(yB,2) = ZERO;
242     Ith(yB,3) = ZERO;
243
244     /* Initialize qB */
245     qB = N_VNew_Serial(NP);
246     if (check_flag((void *)qB, "N_VNew", 0)) return(1);
247     Ith(qB,1) = ZERO;
248     Ith(qB,2) = ZERO;
249     Ith(qB,3) = ZERO;
250
251     /* Set the scalar relative tolerance reltolB */
252     reltolB = RTOL;
253
254     /* Set the scalar absolute tolerance abstolB */
255     abstolB = ATOLl;
256
257     /* Set the scalar absolute tolerance abstolQB */
258     abstolQB = ATOLq;
259
260     /* Create and allocate CVODES memory for backward run */
261     printf("\nCreate and allocate CVODES memory for backward run\n");
262
263     flag = CVodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
264     if (check_flag(&flag, "CVodeCreateB", 1)) return(1);
265
266     flag = CVodeMallocB(cvadj_mem, fB, TB1, yB, CV_SS, reltolB, &abstolB);
267     if (check_flag(&flag, "CVodeMallocB", 1)) return(1);
268

```

```

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

```

```

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

```



```

377
378 y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
379 data = (UserData) jac_data;
380 p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
381
382 IJth(J,1,1) = -p1; IJth(J,1,2) = p2*y3; IJth(J,1,3) = p2*y2;
383 IJth(J,2,1) = p1; IJth(J,2,2) = -p2*y3-2*p3*y2; IJth(J,2,3) = -p2*y2;
384 IJth(J,3,2) = 2*p3*y2;
385 }
386
387 /*
388  * fQ routine. Compute fQ(t,y).
389 */
390
391 static void fQ(realtype t, N_Vector y, N_Vector qdot, void *fQ_data)
392 {
393     Ith(qdot,1) = Ith(y,3);
394 }
395
396 /*
397  * EwtSet function. Computes the error weights at the current solution.
398 */
399
400 static int ewt(N_Vector y, N_Vector w, void *e_data)
401 {
402     int i;
403     realtype yy, ww, rtol, atol[3];
404
405     rtol = RTOL;
406     atol[0] = ATOL1;
407     atol[1] = ATOL2;
408     atol[2] = ATOL3;
409
410     for (i=1; i<=3; i++) {
411         yy = Ith(y,i);
412         ww = rtol * ABS(yy) + atol[i-1];
413         if (ww <= 0.0) return (-1);
414         Ith(w,i) = 1.0/ww;
415     }
416
417     return(0);
418 }
419
420 /*
421  * fB routine. Compute fB(t,y,yB).
422 */
423
424 static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot, void *f_dataB)
425 {
426     UserData data;
427     realtype y1, y2, y3;
428     realtype p1, p2, p3;
429     realtype l1, l2, l3;
430     realtype l21, l32, y23;

```

```

431
432 data = (UserData) f_dataB;
433
434 /* The p vector */
435 p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
436
437 /* The y vector */
438 y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
439
440 /* The lambda vector */
441 l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);
442
443 /* Temporary variables */
444 l21 = l2-l1;
445 l32 = l3-l2;
446 y23 = y2*y3;
447
448 /* Load yBdot */
449 Ith(yBdot,1) = - p1*l21;
450 Ith(yBdot,2) = p2*y3*l21 - RCONST(2.0)*p3*y2*l32;
451 Ith(yBdot,3) = p2*y2*l21 - RCONST(1.0);
452 }
453
454 /*
455  * JacB routine. Compute JB(t,y,yB).
456 */
457
458 static void JacB(long int NB, DenseMat JB, realtype t,
459                 N_Vector y, N_Vector yB, N_Vector fyB, void *jac_dataB,
460                 N_Vector tmp1B, N_Vector tmp2B, N_Vector tmp3B)
461 {
462     UserData data;
463     realtype y1, y2, y3;
464     realtype p1, p2, p3;
465
466     data = (UserData) jac_dataB;
467
468     /* The p vector */
469     p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
470
471     /* The y vector */
472     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
473
474     /* Load JB */
475     IJth(JB,1,1) = p1;      IJth(JB,1,2) = -p1;
476     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;
477     IJth(JB,3,1) = -p2*y2; IJth(JB,3,2) = p2*y2;
478 }
479
480 /*
481  * fQB routine. Compute integrand for quadratures
482 */
483
484 static void fQB(realtype t, N_Vector y, N_Vector yB,

```

```

485         N_Vector qBdot, void *fQ_dataB)
486     {
487         UserData data;
488         realtype y1, y2, y3;
489         realtype p1, p2, p3;
490         realtype l1, l2, l3;
491         realtype l21, l32, y23;
492
493         data = (UserData) fQ_dataB;
494
495         /* The p vector */
496         p1 = data->p[0]; p2 = data->p[1]; p3 = data->p[2];
497
498         /* The y vector */
499         y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
500
501         /* The lambda vector */
502         l1 = Ith(yB,1); l2 = Ith(yB,2); l3 = Ith(yB,3);
503
504         /* Temporary variables */
505         l21 = l2-l1;
506         l32 = l3-l2;
507         y23 = y2*y3;
508
509         Ith(qBdot,1) = y1*l21;
510         Ith(qBdot,2) = - y23*l21;
511         Ith(qBdot,3) = y2*y2*l32;
512     }
513
514     /*
515     *-----
516     * PRIVATE FUNCTIONS
517     *-----
518     */
519
520     /*
521     * Print results after backward integration
522     */
523
524     static void PrintOutput(N_Vector yB, N_Vector qB)
525     {
526         printf("-----\n");
527         #if defined(SUNDIALS_EXTENDED_PRECISION)
528             printf("tB0:          %12.4Le\n", TB1);
529             printf("dG/dp:          %12.4Le %12.4Le %12.4Le\n",
530                 -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
531             printf("lambda(t0): %12.4Le %12.4Le %12.4Le\n",
532                 Ith(yB,1), Ith(yB,2), Ith(yB,3));
533         #elif defined(SUNDIALS_DOUBLE_PRECISION)
534             printf("tB0:          %12.4le\n", TB1);
535             printf("dG/dp:          %12.4le %12.4le %12.4le\n",
536                 -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
537             printf("lambda(t0): %12.4le %12.4le %12.4le\n",
538                 Ith(yB,1), Ith(yB,2), Ith(yB,3));

```

```

539 #else
540     printf("tB0:          %12.4e\n",TB1);
541     printf("dG/dp:        %12.4e %12.4e %12.4e\n",
542           -Ith(qB,1), -Ith(qB,2), -Ith(qB,3));
543     printf("lambda(t0): %12.4e %12.4e %12.4e\n",
544           Ith(yB,1), Ith(yB,2), Ith(yB,3));
545 #endif
546     printf("-----\n\n");
547 }
548
549 /*
550  * Check function return value.
551  *   opt == 0 means SUNDIALS function allocates memory so check if
552  *       returned NULL pointer
553  *   opt == 1 means SUNDIALS function returns a flag so check if
554  *       flag >= 0
555  *   opt == 2 means function allocates memory so check if returned
556  *       NULL pointer
557  */
558
559 static int check_flag(void *flagvalue, char *funcname, int opt)
560 {
561     int *errflag;
562
563     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
564     if (opt == 0 && flagvalue == NULL) {
565         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
566             funcname);
567         return(1); }
568
569     /* Check if flag < 0 */
570     else if (opt == 1) {
571         errflag = (int *) flagvalue;
572         if (*errflag < 0) {
573             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
574                 funcname, *errflag);
575             return(1); }}
576
577     /* Check if function returned NULL pointer - no memory allocated */
578     else if (opt == 2 && flagvalue == NULL) {
579         fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
580             funcname);
581         return(1); }
582
583     return(0);
584 }

```

## E Listing of pvanx.c

```

1  /*
2  * -----
3  * $Revision: 1.17.2.3 $
4  * $Date: 2005/04/28 20:06:28 $
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
216 /*-----
217     Compute and value of g(t_f)
218     -----*/
219 g_val = Compute_g(u, data);
220
221 /*-----
222     Backward integration phase
223     -----*/
224
225 if (my_pe == npes) {
226
227     /* Activate last process for integration of the quadrature equations */
228     local_N = NP;
229
230 } else {
231
232     /* Allocate work space */
233     data->z1 = (realtype *)malloc(local_N*sizeof(realtype));
234     if (check_flag((void *)data->z1, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
235     data->z2 = (realtype *)malloc(local_N*sizeof(realtype));
236     if (check_flag((void *)data->z2, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
237
238 }
239
240 /* Allocate and initialize backward variables */
241 uB = N_VNew_Parallel(comm, local_N, NEQ+NP);
242 if (check_flag((void *)uB, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
243 SetICback(uB, my_base);
244
245 /* Allocate CVODES memory for the backward integration */
246 flag = CNodeCreateB(cvadj_mem, CV_ADAMS, CV_FUNCTIONAL);
247 if (check_flag(&flag, "CNodeCreateB", 1, my_pe)) MPI_Abort(comm, 1);
248 flag = CNodeSetFdataB(cvadj_mem, data);
249 if (check_flag(&flag, "CNodeSetFdataB", 1, my_pe)) MPI_Abort(comm, 1);
250 flag = CNodeMallocB(cvadj_mem, fB, TOUT, uB, CV_SS, reltol, &abstol);
251 if (check_flag(&flag, "CNodeMallocB", 1, my_pe)) MPI_Abort(comm, 1);
252
253 /* Integrate to T0 */
254 flag = CNodeB(cvadj_mem, T0, uB, &t, CV_NORMAL);
255 if (check_flag(&flag, "CNodeB", 1, my_pe)) MPI_Abort(comm, 1);
256
257 /* Print results (adjoint states and quadrature variables) */
258 PrintOutput(g_val, uB, data);
259
260
261 /* Free memory */
262 N_VDestroy_Parallel(u);
263 N_VDestroy_Parallel(uB);
264 CNodeFree(cvode_mem);
265 CVadjFree(cvadj_mem);
266 if (my_pe != npes) {
267     free(data->z1);
268     free(data->z2);

```

```

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

```

```

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

```

```

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

```

```

431
432     uLeft = data_in[0];
433     uBLeft = data_in[1];
434 } else {
435     uLeft = ZERO;
436     uBLeft = ZERO;
437 }
438 if (my_pe != last_pe) {
439     MPI_Recv(data_in, 2, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm, &status);
440
441     uRight = data_in[0];
442     uBRight = data_in[1];
443 } else {
444     uRight = ZERO;
445     uBRight = ZERO;
446 }
447
448 /* Loop over all grid points in current process. */
449 for (i=0; i<my_length; i++) {
450
451     /* Extract uB at x_i and two neighboring points */
452     uBi = uBdata[i];
453     uBlt = (i==0) ? uBLeft: uBdata[i-1];
454     uBrt = (i==my_length-1) ? uBRight : uBdata[i+1];
455
456     /* Set diffusion and advection terms and load into udot */
457     hdiff = hordc*(uBlt - TWO*uBi + uBrt);
458     hadv = horac*(uBrt - uBlt);
459     duBdata[i] = - hdiff + hadv;
460
461     /* Extract u at x_i and two neighboring points */
462     ui = udata[i];
463     ult = (i==0) ? uLeft: udata[i-1];
464     urt = (i==my_length-1) ? uRight : udata[i+1];
465
466     /* Load integrands of the two space integrals */
467     z1[i] = uBdata[i]*(ult - TWO*ui + urt)/(dx*dx);
468     z2[i] = uBdata[i]*(urt - ult)/(TWO*dx);
469 }
470
471 /* Compute local integrals */
472 intgr1 = Xintgr(z1, my_length, dx);
473 intgr2 = Xintgr(z2, my_length, dx);
474
475 /* Send local integrals to 'quadrature' process */
476 MPI_Send(&intgr1, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
477 MPI_Send(&intgr2, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
478
479 }
480
481 }
482
483 /*
484 *-----

```

```

485  * PRIVATE FUNCTIONS
486  *-----
487  */
488
489  /*
490  * Set initial conditions in u vector
491  */
492
493  static void SetIC(N_Vector u, realtype dx, long int my_length, long int my_base)
494  {
495      int i;
496      long int iglobal;
497      realtype x;
498      realtype *udata;
499
500      /* Set pointer to data array and get local length of u */
501      udata = NV_DATA_P(u);
502      my_length = NV_LOCLENGTH_P(u);
503
504      /* Load initial profile into u vector */
505      for (i=1; i<=my_length; i++) {
506          iglobal = my_base + i;
507          x = iglobal*dx;
508          udata[i-1] = x*(XMAX - x)*exp(TWO*x);
509      }
510  }
511
512  /*
513  * Set final conditions in uB vector
514  */
515
516  static void SetICback(N_Vector uB, long int my_base)
517  {
518      int i;
519      realtype *uBdata;
520      long int my_length;
521
522      /* Set pointer to data array and get local length of uB */
523      uBdata = NV_DATA_P(uB);
524      my_length = NV_LOCLENGTH_P(uB);
525
526      /* Set adjoint states to 1.0 and quadrature variables to 0.0 */
527      if (my_base == -1) for (i=0; i<my_length; i++) uBdata[i] = ZERO;
528      else                for (i=0; i<my_length; i++) uBdata[i] = ONE;
529  }
530
531  /*
532  * Compute local value of the space integral  $\int_x z(x) dx$ 
533  */
534
535  static realtype Xintgr(realtype *z, long int l, realtype dx)
536  {
537      realtype my_intgr;
538      long int i;

```

```

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

```

```

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

```



```

647     }
648
649 }
650
651 /*
652  * Check function return value.
653  *     opt == 0 means SUNDIALS function allocates memory so check if
654  *         returned NULL pointer
655  *     opt == 1 means SUNDIALS function returns a flag so check if
656  *         flag >= 0
657  *     opt == 2 means function allocates memory so check if returned
658  *         NULL pointer
659 */
660
661 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
662 {
663     int *errflag;
664
665     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
666     if (opt == 0 && flagvalue == NULL) {
667         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
668             id, funcname);
669         return(1); }
670
671     /* Check if flag < 0 */
672     else if (opt == 1) {
673         errflag = (int *) flagvalue;
674         if (*errflag < 0) {
675             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
676                 id, funcname, *errflag);
677             return(1); }}
678
679     /* Check if function returned NULL pointer - no memory allocated */
680     else if (opt == 2 && flagvalue == NULL) {
681         fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
682             id, funcname);
683         return(1); }
684
685     return(0);
686 }

```

## F Listing of pvakx.c

```
1  /*
2  * -----
3  * $Revision: 1.11.2.1 $
4  * $Date: 2005/04/01 21:55:24 $
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 = CVodeCreate(CV_BDF, CV_NEWTON);
275  flag = CVodeSetFdata(cvode_mem, d);
276  abstol = ATOL;
277  reltol = RTOL;
278  flag = CVodeMalloc(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 = CVodeQuadMalloc(cvode_mem, fQ, q);
292  flag = CVodeSetQuadFdata(cvode_mem, d);
293  flag = CVodeSetQuadErrCon(cvode_mem, TRUE, CV_SS, reltolQ, &abstolQ);
294
295  /* Allocate space for the adjoint calculation */
296  cvadj_mem = CVadjMalloc(cvode_mem, STEPS);
297
298  /* Integrate forward in time while storing check points */
299  if (myId == 0) printf("Begin forward integration... ");
300  flag = CVodeF(cvadj_mem, tf, y, &tret, CV_NORMAL, &ncheckpnt);
301  if (myId == 0) printf("done. ");
302
303  /* Extract quadratures */
304  flag = CVodeGetQuad(cvode_mem, tf, q);
305  qdata = NV_DATA_P(q);
306  MPI_Allreduce(&qdata[0], &G, 1, PVEC_REAL_MPI_TYPE, MPI_SUM, comm);
307  #if defined(SUNDIALS_EXTENDED_PRECISION)
308    if (myId == 0) printf("  G = %Le\n",G);
309  #elif defined(SUNDIALS_DOUBLE_PRECISION)
310    if (myId == 0) printf("  G = %le\n",G);
311  #else
312    if (myId == 0) printf("  G = %e\n",G);
313  #endif
314
315  /* Print statistics for forward run */
316  if (myId == 0) PrintFinalStats(cvode_mem);
317
318  /*-----
319    Backward integration phase
320    -----*/
321
322  /* Allocate and initialize yB */

```

```

323 yB = N_VNew_Parallel(comm, l_neq, neq);
324 N_VConst(ZERO, yB);
325
326 /* Allocate and initialize qB (gradient) */
327 qB = N_VNew_Parallel(comm, l_neq, neq);
328 N_VConst(ZERO, qB);
329
330 /* Create and allocate backward CVODE memory */
331 flag = CVodeCreateB(cvadj_mem, CV_BDF, CV_NEWTON);
332 flag = CVodeSetFdataB(cvadj_mem, d);
333 abstolB = ATOL_B;
334 reltolB = RTOL_B;
335 flag = CVodeMallocB(cvadj_mem, fB, tf, yB, CV_SS, reltolB, &abstolB);
336
337 /* Attach preconditioner and linear solver modules */
338 mudqB = mldqB = d->l_m[0]+1;
339 mukeepB = mlkeepB = 2;
340 flag = CVBBDPrecAllocB(cvadj_mem, l_neq, mudqB, mldqB,
341                        mukeepB, mlkeepB, ZERO, fB_local, NULL);
342 flag = CVBBDSpgmrB(cvadj_mem, PREC_LEFT, 0);
343
344 /* Initialize quadrature calculations */
345 abstolQB = ATOL_QB;
346 reltolQB = RTOL_QB;
347 flag = CVodeQuadMallocB(cvadj_mem, fQB, qB);
348 flag = CVodeSetQuadFdataB(cvadj_mem, d);
349 flag = CVodeSetQuadErrConB(cvadj_mem, TRUE, CV_SS, reltolQB, &abstolQB);
350
351 /* Integrate backwards */
352 if (myId == 0) printf("Begin backward integration... ");
353 flag = CVodeB(cvadj_mem, ti, yB, &tret, CV_NORMAL);
354 if (myId == 0) printf("done.\n");
355
356 /* Print statistics for backward run */
357 if (myId == 0) {
358     cvode_memB = CVadjGetCVodeBmem(cvadj_mem);
359     PrintFinalStats(cvode_memB);
360 }
361
362 /* Extract quadratures */
363 flag = CVodeGetQuadB(cvadj_mem, qB);
364
365 /* Process 0 collects the gradient components and prints them */
366 if (output) {
367     OutputGradient(myId, qB, d);
368     if (myId == 0) printf("Wrote matlab file 'grad.m'.\n");
369 }
370
371 /* Free memory */
372 N_VDestroy_Parallel(y);
373 N_VDestroy_Parallel(q);
374 N_VDestroy_Parallel(qB);
375 N_VDestroy_Parallel(yB);
376

```



```

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

```

```

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

```

```

485     d->y_ext = (realtype *) malloc( size*sizeof(realtype));
486
487     /* Initialize Buffer field.
488        Size of buffer is checked when needed */
489
490     d->buf_send = NULL;
491     d->buf_recv = NULL;
492     d->buf_size = 0;
493
494     /* Allocate space for the source parameters */
495
496     *neq = 1; *l_neq = 1;
497     FOR_DIM { *neq *= d->m[dim]; *l_neq *= d->l_m[dim]; }
498     d->p = NV_New_Parallel(comm, *l_neq, *neq);
499
500     /* Initialize the parameters for a source with Gaussian profile */
501
502     SetSource(d);
503
504 }
505
506 static void SetSource(ProblemData d)
507 {
508     int *l_m, *m_start;
509     realtype *xmin, *xmax, *dx;
510     realtype x[DIM], g, *pdata;
511     int i[DIM];
512
513     l_m = d->l_m;
514     m_start = d->m_start;
515     xmin = d->xmin;
516     xmax = d->xmax;
517     dx = d->dx;
518
519
520     pdata = NV_DATA_P(d->p);
521
522     for(i[0]=0; i[0]<l_m[0]; i[0]++) {
523         x[0] = xmin[0] + (m_start[0]+i[0]) * dx[0];
524         for(i[1]=0; i[1]<l_m[1]; i[1]++) {
525             x[1] = xmin[1] + (m_start[1]+i[1]) * dx[1];
526 #ifdef USE3D
527             for(i[2]=0; i[2]<l_m[2]; i[2]++) {
528                 x[2] = xmin[2] + (m_start[2]+i[2]) * dx[2];
529
530                 g = G1_AMPL
531                     * exp( -SQR(G1_X-x[0])/SQR(G1_SIGMA) )
532                     * exp( -SQR(G1_Y-x[1])/SQR(G1_SIGMA) )
533                     * exp( -SQR(G1_Z-x[2])/SQR(G1_SIGMA) );
534
535                 g += G2_AMPL
536                     * exp( -SQR(G2_X-x[0])/SQR(G2_SIGMA) )
537                     * exp( -SQR(G2_Y-x[1])/SQR(G2_SIGMA) )
538                     * exp( -SQR(G2_Z-x[2])/SQR(G2_SIGMA) );

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

809  */
810
811 static void fB(realtype t, N_Vector y, N_Vector yB, N_Vector yBdot,
812              void *f_dataB)
813 {
814     ProblemData d;
815     int l_neq=1;
816     int dim;
817
818     d = (ProblemData) f_dataB;
819     FOR_DIM l_neq *= d->l_m[dim];
820
821     /* Do all inter-processor communication */
822     f_comm(l_neq, t, yB, f_dataB);
823
824     /* Compute right-hand side locally */
825     fB_local(l_neq, t, y, yB, yBdot, f_dataB);
826 }
827
828 static void fB_local(long int NlocalB, realtype t,
829                    N_Vector y, N_Vector yB, N_Vector dyB,
830                    void *f_dataB)
831 {
832     realtype *YBdata, *dyBdata, *ydata;
833     realtype dx[DIM], c, v[DIM], cl[DIM], cr[DIM];
834     realtype adv[DIM], diff[DIM];
835     realtype xmin[DIM], xmax[DIM], x[DIM], x1;
836     int i[DIM], l_m[DIM], m_start[DIM], nbr_left[DIM], nbr_right[DIM], id;
837     ProblemData d;
838     int dim;
839
840     d = (ProblemData) f_dataB;
841
842     /* Extract stuff from data structure */
843     id = d->myId;
844     FOR_DIM {
845         xmin[dim]      = d->xmin[dim];
846         xmax[dim]      = d->xmax[dim];
847         l_m[dim]       = d->l_m[dim];
848         m_start[dim]   = d->m_start[dim];
849         dx[dim]        = d->dx[dim];
850         nbr_left[dim]  = d->nbr_left[dim];
851         nbr_right[dim] = d->nbr_right[dim];
852     }
853
854     dyBdata = NV_DATA_P(dyB);
855     ydata    = NV_DATA_P(y);
856
857     /* Copy local segment of yB to y_ext */
858     Load_yext(NV_DATA_P(yB), d);
859     YBdata = d->y_ext;
860
861     /* Velocity components in x1 and x2 directions (Poiseuille profile) */
862     v[1] = ZERO;

```

```

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

```

```

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

```

```

971
972     printf("Domain:\n");
973
974 #if defined(SUNDIALS_EXTENDED_PRECISION)
975     printf("    %Lf < x < %Lf    mx = %d    npe_x = %d \n",XMIN,XMAX,MX,NPX);
976     printf("    %Lf < y < %Lf    my = %d    npe_y = %d \n",YMIN,YMAX,MY,NPY);
977 #else
978     printf("    %f < x < %f    mx = %d    npe_x = %d \n",XMIN,XMAX,MX,NPX);
979     printf("    %f < y < %f    my = %d    npe_y = %d \n",YMIN,YMAX,MY,NPY);
980 #endif
981
982 #ifndef USE3D
983 #if defined(SUNDIALS_EXTENDED_PRECISION)
984     printf("    %Lf < z < %Lf    mz = %d    npe_z = %d \n",ZMIN,ZMAX,MZ,NPZ);
985 #else
986     printf("    %f < z < %f    mz = %d    npe_z = %d \n",ZMIN,ZMAX,MZ,NPZ);
987 #endif
988 #endif
989
990     printf("\n");
991 }
992
993 /*
994 *-----
995 * PrintFinalStats:
996 * Print final statistics contained in ccode_mem
997 *-----
998 */
999
1000 static void PrintFinalStats(void *ccode_mem)
1001 {
1002     long int lenrw, leniw ;
1003     long int lenrwSPGMR, leniwSPGMR;
1004     long int nst, nfe, nsetups, nni, ncnf, netf;
1005     long int nli, npe, nps, ncfl, nfeSPGMR;
1006     int flag;
1007
1008     flag = CVodeGetWorkspace(ccode_mem, &lenrw, &leniw);
1009     flag = CVodeGetNumSteps(ccode_mem, &nst);
1010     flag = CVodeGetNumRhsEvals(ccode_mem, &nfe);
1011     flag = CVodeGetNumLinSolvSetups(ccode_mem, &nsetups);
1012     flag = CVodeGetNumErrTestFails(ccode_mem, &netf);
1013     flag = CVodeGetNumNonlinSolvIters(ccode_mem, &nni);
1014     flag = CVodeGetNumNonlinSolvConvFails(ccode_mem, &ncnf);
1015
1016     flag = CVSpgmrGetWorkspace(ccode_mem, &lenrwSPGMR, &leniwSPGMR);
1017     flag = CVSpgmrGetNumLinIters(ccode_mem, &nli);
1018     flag = CVSpgmrGetNumPrecEvals(ccode_mem, &npe);
1019     flag = CVSpgmrGetNumPrecSolves(ccode_mem, &nps);
1020     flag = CVSpgmrGetNumConvFails(ccode_mem, &ncfl);
1021     flag = CVSpgmrGetNumRhsEvals(ccode_mem, &nfeSPGMR);
1022
1023     printf("\nFinal Statistics.. \n\n");
1024     printf("lenrw    = %6ld    leniw = %6ld\n", lenrw, leniw);

```

```

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

```

```

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

```

```

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

```

