

User Documentation for CVODES,  
An ODE Solver with Sensitivity Analysis Capabilities

Alan C. Hindmarsh      Radu Serban

June 25, 2002

#### 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 products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Historical Background . . . . .	1
1.2	Reading this User Guide . . . . .	2
<b>2</b>	<b>Mathematical Considerations</b>	<b>4</b>
2.1	IVP Solution . . . . .	4
2.2	Forward Sensitivity Analysis . . . . .	6
2.3	Adjoint Sensitivity Analysis . . . . .	10
2.4	BDF Stability Limit Detection . . . . .	12
<b>3</b>	<b>Code Organization</b>	<b>14</b>
3.1	SUNDIALS Organization . . . . .	14
3.2	CVODES Organization . . . . .	15
<b>4</b>	<b>Using CVODES for IVP Solution</b>	<b>19</b>
4.1	Header Files . . . . .	19
4.2	A Skeleton of the User's Main Program . . . . .	20
4.3	User-Callable Routines for IVP Solution . . . . .	22
4.3.1	CVODES Initialization Routine . . . . .	22
4.3.2	Linear Solver Specification Routines . . . . .	23
4.3.3	CVODES Solver Routine . . . . .	28
4.3.4	Optional Input/Output . . . . .	29
4.3.5	Interpolated Output Routines . . . . .	29
4.3.6	CVODES Reinitialization Routine . . . . .	31
4.3.7	Linear Solver Reinitialization Routines . . . . .	32
4.4	User-Supplied Routines for IVP Solution . . . . .	34
4.5	CVODES Preconditioner Modules . . . . .	38
4.5.1	A Serial Banded Preconditioner Module . . . . .	38
4.5.2	A Parallel Band-Block-Diagonal Preconditioner Module . . . . .	40
<b>5</b>	<b>Using CVODES for Forward Sensitivity Analysis</b>	<b>43</b>
5.1	A Skeleton of the User's Main Program . . . . .	43
5.2	User-Callable Routines for Forward Sensitivity Analysis . . . . .	45
5.2.1	Forward Sensitivity Initialization Routine . . . . .	45
5.2.2	Forward Sensitivity Extraction Routine . . . . .	47
5.2.3	Additional Optional Input/Output . . . . .	47
5.2.4	Additional Diagnostics Extraction Routine . . . . .	48
5.2.5	Interpolated Sensitivity Output Routines . . . . .	48
5.2.6	Forward Sensitivity Reinitialization Routine . . . . .	49
5.3	User-Supplied Routines for Forward Sensitivity Analysis . . . . .	49

<b>6</b>	<b>Using CVODES for Adjoint Sensitivity Analysis</b>	<b>52</b>
6.1	A Skeleton of the User's Main Program . . . . .	52
6.2	User-Callable Routines for Adjoint Sensitivity Analysis . . . . .	54
6.2.1	Adjoint Sensitivity Allocation Routine . . . . .	54
6.2.2	Forward Integration Routine . . . . .	54
6.2.3	Backward Problem Initialization Routine . . . . .	54
6.2.4	Linear Solver Initialization Routines for Backward Problem . . . . .	55
6.2.5	Backward Integration Routine . . . . .	56
6.2.6	Adjoint Sensitivity Deallocation Routine . . . . .	57
6.2.7	Check Point Listing Routine . . . . .	57
6.3	User-Supplied Routines for Adjoint Sensitivity Analysis . . . . .	57
6.4	Using the Banded Preconditioner Module for Adjoint Sensitivity Analysis . . . . .	60
<b>7</b>	<b>Example Problems for IVP Solution</b>	<b>62</b>
7.1	A Serial Sample Problem . . . . .	63
7.2	A Parallel Sample Program . . . . .	65
<b>8</b>	<b>Example Problems for Forward Sensitivity Analysis</b>	<b>69</b>
8.1	A Serial Sample Problem . . . . .	69
8.2	A Parallel Sample Program . . . . .	75
<b>9</b>	<b>Example Problems for Adjoint Sensitivity Analysis</b>	<b>82</b>
9.1	A Serial Sample Problem . . . . .	83
9.2	A Parallel Sample Program . . . . .	85
<b>10</b>	<b>Types realtype and integertype</b>	<b>89</b>
10.1	Description . . . . .	89
10.2	Changing Type realtype . . . . .	89
10.3	Changing Type integertype . . . . .	89
<b>11</b>	<b>Description of the NVECTOR Concept</b>	<b>91</b>
11.1	The NVECTOR_SERIAL Implementation of NVECTOR . . . . .	95
11.2	The NVECTOR_PARALLEL Implementation of NVECTOR . . . . .	97
11.3	NVECTOR Kernels Used by CVODES . . . . .	100
<b>12</b>	<b>Providing Alternate Linear Solver Modules</b>	<b>102</b>
<b>13</b>	<b>Generic Linear Solvers in SUNDIALS</b>	<b>105</b>
13.1	The DENSE Module . . . . .	105
13.2	The BAND Module . . . . .	108
13.3	The SPGMR Module . . . . .	111
	<b>References</b>	<b>112</b>
	<b>Index</b>	<b>113</b>

<b>A</b>	<b>Listings of CVODES IVP Solution Examples</b>	<b>118</b>
A.1	A Serial Sample Problem - <code>cvdx.c</code> . . . . .	118
A.2	A Parallel Sample Program - <code>pvkx.c</code> . . . . .	123
<b>B</b>	<b>Listings of CVODES Forward Sensitivity Examples</b>	<b>142</b>
B.1	A Serial Sample Problem - <code>cvfdx.c</code> . . . . .	142
B.2	A Parallel Sample Program - <code>pvfkx.c</code> . . . . .	150
<b>C</b>	<b>Listings of CVODES Adjoint Sensitivity Examples</b>	<b>172</b>
C.1	A Serial Sample Problem - <code>cvadx.c</code> . . . . .	172
C.2	A Parallel Sample Program - <code>pvanx.c</code> . . . . .	179

## List of Tables

1	List of files in the CVODES package . . . . .	18
2	Description of the optional integer input-output array <b>iopt</b> . . . . .	30
3	Description of the optional real input-output array <b>ropt</b> . . . . .	31
4	Additional optional integer output from forward sensitivity . . . . .	48
5	Description of the NVECTOR kernels . . . . .	93
6	List of vector kernels usage by CVODES code modules . . . . .	101

## List of Figures

1	Organization of the SUNDIALS suite . . . . .	14
2	Overall structure diagram of the CVODES package . . . . .	16
3	Diagram of the user program and CVODES package for integration of IVP . . . . .	21
4	Diagram of the storage for a matrix of type <b>BandMat</b> . . . . .	109





# 1 Introduction

CVODES is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. This suite consists of CVODE, KINSOL, and IDA, and variants of these. CVODES is a solver for stiff and nonstiff initial value problems for systems of ordinary differential equation (ODEs). In addition to solving stiff and nonstiff ODE systems, CVODES has sensitivity analysis capabilities, using either the forward or the adjoint methods.

## 1.1 Historical Background

FORTRAN solvers for ODE initial value problems are widespread and heavily used. Two solvers that have been written at LLNL in the past are VODE [1] and VODPK [3]. VODE is a general purpose solver that includes methods for stiff and nonstiff systems, and in the stiff case uses direct methods (full or banded) for the solution of the linear systems that arise at each implicit step. Externally, VODE is very similar to the well known solver LSODE [14]. VODPK is a variant of VODE that uses a preconditioned Krylov (iterative) method for the solution of the linear systems. VODPK is a powerful tool for large stiff systems because it combines established methods for stiff integration, nonlinear iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant source of stiffness, in the form of the user-supplied preconditioner matrix [2]. The capabilities of both VODE and VODPK have been combined in the C-language package CVODE [7, 8].

In the process of translating the VODE and VODPK algorithms into C, the overall CVODE organization has been changed considerably. One key feature of the CVODE organization is that the linear system solvers comprise a layer of code modules that is separated from the integration algorithm, allowing for easy modification and expansion of the linear solver array. A second key feature is a separate module devoted to vector operations; this facilitated the extension to multiprocessor environments with minimal impacts on the rest of the solver, resulting in PODE [4], the parallel variant of CVODE.

CVODES is written with a functionality that is a superset of that of the pair CVODE/PODE. Sensitivity analysis capabilities, both forward and adjoint, have been added to the main integrator. Enabling forward sensitivity computations in CVODES will result in the code integrating the so-called *sensitivity equations* simultaneously with the original IVP, yielding both the solution and its sensitivity with respect to parameters in the model. Adjoint sensitivity analysis, most useful when the gradients of relatively few functionals of the solution with respect to many parameters are sought, involves integration of the original IVP forward in time followed by the integration of the so-called *adjoint equations* backwards in time. CVODES provides the infrastructure needed to integrate any final-condition ODE dependent on the solution of the original IVP (in particular the adjoint system).

Development of CVODES was concurrent with a redesign of the vector operations module across the SUNDIALS suite. The key feature of the new NVECTOR module is that it is written in terms of abstract vector operations with the actual vector kernels attached by a particular implementation (such as serial or parallel) of NVECTOR. This allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file.

There are several motivations for choosing the C language for CVODE and later for CVODES. First, a general movement away from FORTRAN and toward C in scientific computing is apparent.

Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for CVODES because of the wider availability of C compilers, the potentially greater efficiency of C, and the greater ease of interfacing the solver to applications written in extended FORTRAN.

## 1.2 Reading this User Guide

This user guide is a combination of general usage instructions and specific example programs. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of CVODES. The most casual user, with an IVP problem only, can get by with reading §2.1, then §4 through §4.3.3 only, and looking at examples in §7 and App. A. In addition, to solve a forward sensitivity problem the user should read §2.2, followed by §5 through §5.2.2 only, and look at examples in §8 and App. B. In a different direction, a more expert user with an IVP problem may want to (a) use a package preconditioner (§4.5), (b) supply his/her own Jacobian or preconditioner routines (§4.4), (c) do multiple runs of problems of the same size (§4.3.6 and §4.3.7), (d) supply a new NVECTOR module (§11), or even (e) supply a different linear solver module (§3.2 and §13). An advanced user with a forward sensitivity problem may also want to (a) provide his/her own sensitivity equations right-hand side routine (§5.3), (b) perform multiple runs with the same number of sensitivity parameters (§5.2.6), or (c) extract additional diagnostic information (§5.2.2). A user with an adjoint sensitivity problem needs to understand the IVP solution approach at the desired level and also go through §2.3 for a short mathematical description of the adjoint approach, §6 for the usage of the adjoint module in CVODES, and the examples in §9 and App. C.

The structure of this document is as follows:

- In §2, we begin with short descriptions of the numerical methods implemented by CVODES for the solution of initial value problems for systems of ODEs and continue with an overview of the mathematical aspects of sensitivity analysis, both forward (§2.2) and adjoint (§2.3).
- The following section describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the CVODES solver (§3.2).
- In §4, we give an overview of the usage of CVODES, as well as a complete description of the user interface and of the user-defined routines for integration of IVP ODEs. Readers that are not interested in using CVODES for sensitivity analysis can then skip to the example programs.
- Section 5 describes the usage of CVODES for forward sensitivity analysis as an extension of its IVP integration capabilities. We begin with a skeleton of the user main program, with emphasis on the steps that are required in addition to those already described in §4. Following that we provide detailed descriptions of the user-callable interface routines specific to forward sensitivity analysis and of the additional optional user-defined routines.
- Section §6 describes the usage of CVODES for adjoint sensitivity analysis. We begin by describing the CVODES check-pointing implementation for interpolation of the original IVP solution during integration of the adjoint system backwards in time and with an overview of the user

main program. Following that we provide complete descriptions of the user-callable interface routines for adjoint sensitivity analysis as well as descriptions of the required additional user-defined routines.

- The subsequent sections contain sample programs that illustrate the usage of CVODES with different choices of the linear system solvers for integration of IVP ODEs (§7), forward sensitivity analysis (§8), and adjoint sensitivity analysis (§9) with code listings provided in Appendices A, B, and C, respectively. Each of these sections is self-contained and provides both serial and parallel examples. In each case, we give the program source, a step-by-step explanation of the program, and the output. Our intention is that these programs will enable the user to learn CVODES by example, and each can serve as a template for user programs.
- Section 10 describes the `realtype` and `integertype` type definitions used across the SUNDIALS solvers, as well as instructions on changing these definitions.
- Section 11 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, as well as details on the two NVECTOR implementations provided with SUNDIALS: a serial implementation (§11.1) and a parallel implementation, based on MPI (§11.2).
- Section 13 describes in details the generic linear solvers shared by all SUNDIALS solvers.
- Appendices A-C contain the code listings of the six CVODES sample programs described in detail in this document.

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as `CVodeMalloc`) within textual explanations appear in type-writer type style; fields in C structures (such as *content*) appear in italics; and packages or modules, such as CVDENSE, are written in all capitals.

## 2 Mathematical Considerations

CVODES solves initial-value problems (IVPs) for systems of ODEs. Such problems can be stated as

$$\begin{aligned}\dot{y} &= f(t, y) \\ y(t_0) &= y_0,\end{aligned}\tag{1}$$

where  $y \in \mathbf{R}^N$ ,  $\dot{y} = dy/dt$  and  $\mathbf{R}^N$  is the real  $N$ -dimensional vector space. That is, (1) represents a system of  $N$  ordinary differential equations and their initial conditions at some  $t_0$ . The dependent variable is  $y$  and the independent variable is  $t$ . The independent variable need not appear explicitly in the  $N$ -vector valued function  $f$ .

Additionally, if (1) depends on some parameters  $p \in \mathbf{R}^{N_p}$ , i.e.

$$\begin{aligned}\dot{y} &= f(t, y, p) \\ y(t_0) &= y_0(p),\end{aligned}\tag{2}$$

CVODES can also compute first order derivative information, performing either *forward sensitivity analysis* or *adjoint sensitivity analysis*. In the first case, CVODES computes the sensitivities of the solution with respect to the parameters  $p$ , while in the second case, CVODES computes the gradient of a *derived function* with respect to the parameters  $p$ .

### 2.1 IVP Solution

The IVP is solved by one of two numerical methods. These are the backward differentiation formula (BDF) and the Adams-Moulton formula. Both are implemented in a variable-stepsize, variable-order form. The BDF uses a fixed-leading-coefficient form. These formulas can both be represented by a linear multistep formula

$$\sum_{i=0}^{K_1} \alpha_{n,i} y_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}_{n-i} = 0\tag{3}$$

where the  $N$ -vector  $y_n$  is the computed approximation to  $y(t_n)$ , the exact solution of (1) at  $t_n$ . The stepsize is  $h_n = t_n - t_{n-1}$ . The coefficients  $\alpha_{n,i}$  and  $\beta_{n,i}$  are uniquely determined by the particular integration formula, the history of the stepsize, and the normalization  $\alpha_{n,0} = -1$ . The Adams-Moulton formula is recommended for nonstiff ODEs and is represented by (3) with  $K_1 = 1$  and  $K_2 = q - 1$ . The order of this formula is  $q$  and its values range from 1 through 12. For stiff ODEs, BDF should be selected and is represented by (3) with  $K_1 = q$  and  $K_2 = 0$ . For BDF, the order  $q$  may take on values from 1 through 5. In the case of either formula, the integration begins with  $q = 1$ , and after that  $q$  varies automatically and dynamically.

For either BDF or the Adams formula,  $\dot{y}_n$  denotes  $f(t_n, y_n)$ . That is, (3) is an implicit formula, and the nonlinear equation

$$\begin{aligned}G(y_n) &\equiv y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0 \\ a_n &= \sum_{i>0} (\alpha_{n,i} y_{n-i} + h_n \beta_{n,i} \dot{y}_{n-i})\end{aligned}\tag{4}$$

must be solved for  $y_n$  at each time step. For nonstiff problems, functional (or fixpoint) iteration is normally used and does not require the solution of a linear system of equations. For stiff problems,

a Newton iteration is used and for each iteration an underlying linear system must be solved. This linear system of equations has the form

$$M[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}), \quad (5)$$

where  $y_{n(m)}$  is the  $m$ th approximation to  $y_n$ , and  $M$  approximates  $\partial G/\partial y$ :

$$M \approx I - \gamma J, \quad J = \frac{\partial f}{\partial y}, \quad \gamma = h_n \beta_{n,0}. \quad (6)$$

At present, aside from a diagonal Jacobian approximation, the other options implemented in CVODES for solving the linear systems (5) are:

- a direct method with dense treatment of the Jacobian;
- a direct method with band treatment of the Jacobian;
- an iterative method SPGMR (scaled, preconditioned GMRES) [2], which is a Krylov subspace method. In most cases, performance of SPGMR is improved by user-supplied preconditioners. The user may precondition the system on the left, on the right, on both the left and right, or use no preconditioner.

In most cases of interest to the CVODES user, the technique of integration will involve BDF and the Newton method coupled with one of the linear solver modules.

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

$$\|E_n\|_{rms,w} < 1.$$

Here the weighted root-mean-square norm is defined by

$$\|E_n\|_{rms,w} = \left[ \sum_{i=1}^N \frac{1}{N} (w_i E_{n,i})^2 \right]^{1/2}, \quad (7)$$

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

$$w_i = \frac{1}{rtol|y_i| + atol_i}. \quad (8)$$

This permits an arbitrary combination of relative and absolute error control. The user-specified relative error tolerance is the scalar  $rtol$  and the user-specified absolute error tolerance is  $atol$  which may be an  $N$ -vector (as indicated above) or a scalar. The value for  $rtol$  indicates the number of digits of relative accuracy for a single time step. The specified value for  $atol_i$  indicates the values of the corresponding component of the solution vector which may be thought of as being zero, or at the noise level. In particular, if we set  $atol_i = rtol \times floor_i$  then  $floor_i$  represents the floor value for the  $i$ th component of the solution and is that magnitude of the component for which there is a crossover from relative error control to absolute error control. Since these tolerances define the allowed error per step, they should be chosen conservatively. Experience indicates that a conservative choice yields a more economical solution than error tolerances that are too large.

The error control mechanism in CVODES varies the stepsize and order in an attempt to take minimum number of steps while satisfying the local error test. The order control can be (optionally) modified with an algorithm that attempts to detect limitations resulting from BDF stability properties.

## 2.2 Forward Sensitivity Analysis

Typically, the governing equations of complex, large-scale models depend on various parameters, through the right-hand side vector and/or through the vector of initial conditions:

$$\begin{aligned}\dot{y} &= f(t, y, p) \\ y(t_0) &= y_0(p),\end{aligned}\tag{9}$$

where  $y \in \mathbf{R}^N$  and  $p \in \mathbf{R}^{N_p}$ . In addition to numerically solving the ODEs, it may be desirable to determine the sensitivity of the results with respect to the model parameters. Such sensitivity information can be used to estimate which parameters are most influential in affecting the behavior of the simulation or to evaluate optimization gradients (in the setting of dynamic optimization, parameter estimation, optimal control, etc.).

The *solution sensitivity* with respect to the model parameter  $p_i$  is defined as the vector:

$$s_i(t) = \frac{\partial y(t)}{\partial p_i}\tag{10}$$

and satisfies the following *forward sensitivity equations* (or in short *sensitivity equations*):

$$\begin{aligned}\dot{s}_i &= \frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i} \\ s_i(t_0) &= \frac{\partial y_0(p)}{\partial p_i},\end{aligned}\tag{11}$$

which are obtained by applying the chain rule of differentiation to the original ODEs (9). The initial sensitivity vector  $s_i(t_0)$  is either all zeros (if  $p_i$  occurs only in  $f$ ), or has nonzeros according to how  $y_0(p)$  depends on  $p_i$ .

When performing forward sensitivity analysis, CVODES carries out the time integration of the combined system, (9) and (11), by viewing it as an ODE system of size  $N(N_s + 1)$ , where  $N_s$  represents a subset of model parameters  $p_i$ , with respect to which sensitivities are desired ( $N_s \leq N_p$ ). However, major efficiency improvements can be obtained by taking advantage of the special form of the sensitivity equations as linearizations of the original ODEs. In particular, for stiff systems, in which case CVODES employs a Newton iteration, the original ODE system and all sensitivity systems share the same Jacobian matrix, and therefore the same iteration matrix  $M$  in (6).

The sensitivity equations are solved with the same linear multistep formula that was selected for the original ODEs and, if Newton iteration was selected, the same linear solver is used in the correction phase for both state and sensitivity variables. In addition, CVODES offers the option of including (*full error control*) or excluding (*partial error control*) the sensitivity variables from the local error test.

**Forward sensitivity methods.** In what follows we briefly describe three methods that have been proposed for the solution of the combined ODE and sensitivity system. Due to its inefficiency, especially for large-scale problems, the first approach is not implemented in CVODES.

- *Staggered Direct Method*

In this method [6], the nonlinear system (4) is first solved and, once an acceptable numerical solution is obtained, the sensitivity variables are found by the direct solution of

$$\dot{s}_i - \frac{\partial f}{\partial y} s_i = \frac{\partial f}{\partial p_i}, \quad (12)$$

where the BDF discretization is used to eliminate  $\dot{s}_i$ . Although the system matrix of the above linear system is based on the exact same information as the matrix  $M$  in (6), it must be updated and factored at every step of the integration as  $M$  is updated only occasionally. The computational cost associated with these matrix updates and factorizations makes this method unattractive when compared with the methods described below and is therefore not implemented in CVODES.

- *Simultaneous Corrector Method*

In this method [13], the BDF discretization is applied simultaneously to both the original equations (9) and the sensitivity systems (11) resulting in the following nonlinear system

$$\hat{G}(\hat{y}_n) \equiv \hat{y}_n - h_n \beta_{n,0} \hat{f}(t_n, \hat{y}_n) - \hat{a}_n = 0,$$

where  $\hat{y} = [y, \dots, s_i, \dots]$  and  $\hat{f} = [f(t, y, p), \dots, (\partial f / \partial y)(t, y, p) s_i + (\partial f / \partial p_i)(t, y, p), \dots]$  and  $\hat{a}_n$  are the terms in the BDF discretization that depend on the solution at previous integration steps. This combined nonlinear system can be solved as in (5) using a modified Newton method by solving the corrector equation

$$\hat{M}[\hat{y}_{n(m+1)} - \hat{y}_{n(m)}] = -\hat{G}(\hat{y}_{n(m)}) \quad (13)$$

at each iteration, where

$$\hat{M} = \begin{bmatrix} M & & & & \\ \gamma J_1 & M & & & \\ \gamma J_2 & 0 & M & & \\ \vdots & \vdots & \ddots & \ddots & \\ \gamma J_{N_s} & 0 & \dots & 0 & M \end{bmatrix},$$

$M$  is defined as in (6), and  $J_i = (\partial / \partial y) [(\partial f / \partial y) s_i + (\partial f / \partial p_i)]$ . It can be shown that a 2-step quadratic convergence can be attained by only using the block-diagonal portion of  $\hat{M}$  in the corrector equation (13). This results in a decoupling that allows the reuse of  $M$  without additional matrix factorizations. However, the products  $(\partial f / \partial y) s_i$  as well as the vectors  $\partial f / \partial p_i$  must still be reevaluated at each step of the iterative process (13) to update the sensitivity portions of the residual  $\hat{G}$ .

- *Staggered Corrector Method*

In the staggered corrector method [9], as in the staggered direct method, the nonlinear system (4) is solved first using the Newton iteration (5). Then, a separate Newton iteration is used to solve the sensitivity system (12):

$$M[s_{i,n(m+1)} - s_{i,n(m)}] = s_{i,n(m)} - \gamma \left( \frac{\partial f}{\partial y}(t_n, y_n, p) s_{i,n(m)} + \frac{\partial f}{\partial p_i}(t_n, y_n, p) \right) - a_{i,n}, \quad (14)$$

where  $a_{i,n} = \sum_{j>0} (\alpha_{n,j} s_{i,n-j} + h_n \beta_{n,j} \dot{s}_{i,n-j})$ . In other words, a modified-Newton iteration is used to solve a linear system. In this approach, the vectors  $\partial f / \partial p_i$  need be updated only once per integration step, after the state correction phase (5) has converged. Note also that Jacobian-related data can be reused at all iterations (14) to evaluate the products  $(\partial f / \partial y) s_i$ .

CVODES implements the simultaneous corrector method and two flavors of the staggered corrector method which differ only if the sensitivity variables are included in the error control test. In the *full error control* case, the first variant of the staggered corrector method requires the convergence of the iterations (14) for all  $N_s$  sensitivity systems and then performs the error test on the sensitivity variables. The second variant of the method will perform the error test for each sensitivity vector  $s_i$ ,  $i = 1, 2, \dots, N_s$  individually, as they pass the convergence test. Differences in performance between the two variants may therefore be noticed whenever one of the sensitivity vectors  $s_i$  fails a convergence or error test.

An important observation is that the staggered corrector method, combined with the SPGMR linear solver effectively results in a staggered direct method. Indeed, SPGMR requires only the action of the matrix  $M$  on a vector and this can be provided with the current Jacobian information. Therefore, the modified Newton procedure (14) will theoretically converge after one iteration.

**Selection of the absolute tolerances for sensitivity variables.** If the sensitivities are considered in the error test, CVODES provides an automated estimation of absolute tolerances for the sensitivity variables based on the absolute tolerance for the corresponding state variable. The selection of  $atol$  for the sensitivity variables is based on the observation that the sensitivity vector  $s_i$  will have units of  $[y]/[p_i]$ . With this, the absolute tolerance for the  $j$ -th component of the sensitivity vector  $s_i$  is set to

$$atol_{S_{i,j}} = \frac{atol_j}{|\bar{p}_i|},$$

where  $atol$  are the absolute tolerances for the state variables and  $\bar{p}$  is a vector of scaling factors that are dimensionally consistent with the model parameters  $p$  and give indication of their order of magnitude. Typically, if  $p_i \neq 0$ , then  $\bar{p}_i = p_i$ . The relative tolerance  $rtolS$  for sensitivity variables is set to be the same as for the state variables, i.e.  $rtolS = rtol$ . This choice of relative and absolute tolerances is equivalent to requiring that the weighted root-mean-square norm (7) of the sensitivity vector  $s_i$  with weights (8) based on  $s_i$  is the same as the weighted root-mean-square norm of the vector of scaled sensitivities  $\bar{s}_i = |\bar{p}_i| s_i$  with weights based on the state variables (the scaled sensitivities  $\bar{s}_i$  being dimensionally consistent with the state variables).

**Evaluation of the sensitivity right-hand side.** There are several methods for evaluating the right-hand side of the sensitivity systems  $[(\partial f / \partial y) s_i + (\partial f / \partial p_i)]$ : analytic evaluation, automatic differentiation, complex-step approximation, finite differences (or directional derivatives). CVODES provides all the software hooks for implementing interfaces to automatic differentiation or complex-step approximation and future versions will provide these capabilities. At the present time, besides the option for analytical sensitivity right hand sides (user-provided), CVODES can evaluate these quantities using various finite difference-based approximations. The first option applies central



finite differences to each term separately:

$$\frac{\partial f}{\partial y} s_i \approx \frac{f(t, y + \delta_y s_i, p) - f(t, y - \delta_y s_i, p)}{2 \delta_y} \quad (15)$$

$$\frac{\partial f}{\partial p_i} \approx \frac{f(t, y, p + \delta_i e_i) - f(t, y, p - \delta_i e_i)}{2 \delta_i}. \quad (15')$$

As is typical for finite differences, the proper choice of perturbations  $\delta_y$  and  $\delta_i$  is a delicate matter. CVODES uses  $\delta_y$  and  $\delta_i$  that take into account several problem-related features; namely, the relative ODE error tolerance  $rtol$ , the machine unit roundoff  $\epsilon$ , the scale factor  $\bar{p}_i$ , and the weighted root-mean-square norm of the sensitivity vector  $s_i$ . We then define

$$\delta_i = |\bar{p}_i| \sqrt{\max(rtol, \epsilon)}; \quad \delta_y = \frac{|\bar{p}_i|}{\max(1/\delta_i, \|s_i\|_{rms,w})}.$$

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

Another technique for estimating the scaled sensitivity derivatives via centered differences is by using directional derivatives:

$$\frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \delta s_i, p + \delta e_i) - f(t, y - \delta s_i, p - \delta e_i)}{2 \delta}, \quad (16)$$

in which

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

If  $\delta_i = \delta_y = \delta$ , a Taylor series analysis shows that the sum of (15)–(15') and (16) are equivalent to within  $O(\delta^2)$ . However, the latter approach is half as costly since it only requires two evaluations of  $f(t, y, p)$ . To take advantage of this savings, it may also be desirable to use the latter formula when  $\delta_i \approx \delta_y$ . CVODES accommodates this possibility by allowing the user to specify a threshold parameter  $\rho_{\max}$ . In particular, if  $\delta_i$  and  $\delta_y$  are within a factor of  $|\rho_{\max}|$  of each other then (16) is used to estimate the scaled sensitivity derivatives. Otherwise, the sum of (15)–(15') is used since  $\delta_i$  and  $\delta_y$  differ by a relatively large amount and the use of separate perturbations is prudent.

These procedures for choosing the perturbations  $(\delta_i, \delta_y, \delta)$  and switching  $(\rho_{\max})$  between finite difference and directional derivative formulas have also been implemented for first-order formulas. Forward finite differences can be applied to  $\frac{\partial f}{\partial y} s_i$  and  $\frac{\partial f}{\partial p_i}$  separately or the single directional derivative formula

$$\frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i} \approx \frac{f(t, y + \delta s_i, p + \delta e_i) - f(t, y, p)}{\delta}$$

can be used. In CVODES, the default value of  $\rho_{\max} = 0$  indicates the use of the second-order centered directional derivative formula (16) exclusively. Otherwise, the magnitude of  $\rho_{\max}$  and its sign (positive or negative) indicates whether this switching is done with regard to (centered or forward) finite differences, respectively.

### 2.3 Adjoint Sensitivity Analysis

In the *forward sensitivity approach* described in the previous section, obtaining sensitivities with respect to  $N_s$  parameters is roughly equivalent to solving an ODE system of size  $(1 + N_s)N$ . This can become prohibitively expensive, especially for large-scale problems, if sensitivities with respect to many parameters are desired. In this situation, the *adjoint sensitivity method* is a very attractive alternative, provided that we do not need the solution sensitivities  $s_i$ , but rather the gradients with respect to model parameters of a relatively few derived functionals of the solution. In other words, if  $y(t)$  is the solution of (9), we wish to evaluate the gradient  $\frac{dG}{dp}$  with respect to  $p$  of

$$G(p) = \int_{t_0}^{t_1} g(t, y, p) dt, \quad (17)$$

or, alternatively, the gradient  $\frac{dg}{dp}$  of the function  $g(t, x, p)$  at time  $t_1$ . The function  $g$  must be smooth enough that  $g_p$  and  $g_x$  exist and are bounded. In what follows, we only sketch the analysis for the sensitivity problem for both  $G$  and  $g$ . For details on the derivation see [5]. Introducing a Lagrange multiplier  $\lambda$ , we form the augmented objective function

$$I(p) = G(p) - \int_{t_0}^{t_1} \lambda^* (\dot{y} - f(t, y, p)) dt, \quad (18)$$

where  $*$  denotes the transpose conjugate. The gradient of  $G$  with respect to  $p$  is

$$\frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{t_1} (g_p + g_y s) dt - \int_{t_0}^{t_1} \lambda^* (\dot{s} - f_y s - f_p) dt, \quad (19)$$

where subscripts on functions such as  $f$  or  $g$  are used to denote partial derivatives and  $s = [s_1, \dots, s_{N_s}]$  is the matrix of solution sensitivities. Applying integration by parts to the term  $\lambda^* \dot{s}$  and selecting  $\lambda$  such that

$$\begin{aligned} \dot{\lambda} &= - \left( \frac{\partial f}{\partial y} \right)^* \lambda - \left( \frac{\partial g}{\partial y} \right)^* \\ \lambda(t_1) &= 0, \end{aligned} \quad (20)$$

the gradient of  $G$  with respect to  $p$  is nothing but

$$\frac{dG}{dp} = \lambda^*(t_0) s(t_0) + \int_{t_0}^{t_1} (g_p + \lambda^* f_p) dt. \quad (21)$$

The gradient of  $g(t_1, y, p)$  with respect to  $p$  can be then obtained by using the Leibnitz differentiation rule. Indeed, from (17),

$$\frac{dg}{dp}(t_1) = \frac{d}{dt_1} \frac{dG}{dp}$$

and therefore, taking into account that  $dG/dp$  in (21) depends on  $t_1$  both through the upper integration limit and through  $\lambda$  and that  $\lambda(t_1) = 0$ ,

$$\frac{dg}{dp}(t_1) = \mu^*(t_0) s(t_0) + g_p(t_1) + \int_{t_0}^{t_1} \mu^* f_p dt, \quad (22)$$

where  $\mu$  is the sensitivity of  $\lambda$  with respect to the final integration limit and thus satisfies the following equation, obtained by taking the total derivative with respect to  $t_1$  of (20):

$$\begin{aligned}\dot{\mu} &= - \left( \frac{\partial f}{\partial y} \right)^* \mu \\ \mu(t_1) &= \left( \frac{\partial g}{\partial y} \right)^* .\end{aligned}\tag{23}$$

The final condition on  $\mu(t_1)$  follows from  $(\partial\lambda/\partial t) + (\partial\lambda/\partial t_1) = 0$  at  $t_1$ , and therefore,  $\mu(t_1) = -\dot{\lambda}(t_1)$ .

The first thing to notice about the adjoint system (20) is that there is no explicit specification of the parameters  $p$ ; this implies that, once the solution  $\lambda$  is found, the formula (21) can then be used to find the gradient of  $G$  with respect to any of the parameters  $p$ . The same holds true for the system (23) and the formula (22) for gradients of  $g(t_1, y, p)$ . The second important remark is that the adjoint systems (20) and (23) are terminal value problems which depend on the solution  $y(t)$  of the original IVP (9). Therefore, a procedure is needed for providing the states  $y$  obtained during a forward integration phase of (9) to CVODES during the backward integration phase of (20) or (23). The approach adopted in CVODES, based on *check-pointing* is described below.

**Check-pointing scheme.** During the backward integration, the evaluation of the right hand side of the adjoint system requires, at the current time, the states  $y$  which were computed in the forward integration phase. Since CVODES implements variable-stepsizes integration formulas, it is unlikely that the states will be available at the desired time and therefore some form of interpolation is needed. The CVODES implementation being also variable-order, it is possible that during the forward integration phase the order may be reduced as low as 1st order, which means that there may be points in time where only  $y$  and  $\dot{y}$  are available. Therefore, CVODES employs a cubic Hermite interpolation algorithm. However, especially for large-scale problems and long integration intervals, the number and size of the vectors  $y$  and  $\dot{y}$  that would need to be stored make this approach computationally intractable.

CVODES settles for a compromise *storage space - execution time* by implementing a so-called *check-pointing scheme*. At the cost of one additional forward integration, this approach offers the best possible estimate of memory requirements for adjoint sensitivity analysis. To begin with, based on the problem size  $N$  and the available memory, the user decides on the number  $N_d$  of data pairs  $y-\dot{y}$  that can be kept in memory for the purpose of interpolation. Then, during the first forward integration stage, every  $N_d$  integration steps a check point is formed by saving enough information (either in memory or on disk if needed) to allow for a hot restart, that is a restart which will exactly reproduce the forward integration. In order to avoid storing Jacobina-related data at each check point, a reevaluation of the iteration matrix is forced before each check point. At the end of this stage, we are left with  $N_c$  check points, including one at  $t_0$ . During the backward integration stage, the adjoint variables are integrated from  $t_1$  to  $t_0$  going from one check point to the previous one. The backward integration from check point  $i + 1$  to check point  $i$  is preceded by a forward integration from  $i$  to  $i + 1$  during which  $N_d$  data pairs  $y-\dot{y}$  are generated and stored in memory for interpolation.

This approach transfers the uncertainty in the number of integration steps in the forward integration phase to uncertainty in the final number of check points. However,  $N_c$  is much smaller than the number of steps taken during the forward integration and there is no major penalty for writing and then reading check point data to/from a temporary file.

Finally, we note that the adjoint sensitivity module in CVODES provides the infrastructure to integrate backwards in time any ODE terminal value problem dependent on the solution of the IVP (9), including adjoint systems (20) or (23), as well as any other quadrature ODEs that may be needed in evaluating the integrals in (21) or (22). In particular, for ODE systems arising from semi-discretization of time-dependent PDEs, this feature allows for integration of both the discretized adjoint PDE system and the adjoint of the discretized PDE.

## 2.4 BDF Stability Limit Detection

CVODES includes an algorithm, STALD (STability Limit Detection), which provides protection against potentially unstable behavior of the BDF multistep integration methods in certain situations, as described below.

When the BDF option is selected, CVODES uses Backward Differentiation Formula methods of orders 1 to 5. At order 1 or 2, the BDF method is A-stable, meaning that for any complex constant  $\lambda$  in the open left half-plane, the method is unconditionally stable (for any step size) for the standard scalar model problem  $dy/dt = \lambda y$ . For an ODE system, this means that, roughly speaking, as long as all modes in the system are stable, the method is also stable for any choice of step size, at least in the sense of a local linear stability analysis.

At orders 3 to 5, the BDF methods are not A-stable, although they are *stiffly stable*. In each case, in order for the method to be stable at step size  $h$  on the scalar model problem, the product  $h\lambda$  must lie in a *region of absolute stability*. That region excludes a portion of the left half-plane that is concentrated near the imaginary axis. The size of that region of instability grows as the order increases from 3 to 5. What this means is that, when running BDF at any of these orders, if an eigenvalue  $\lambda$  of the system lies close enough to the imaginary axis, the step sizes  $h$  for which the method is stable are limited (at least according to the linear stability theory) to a set that prevents  $h\lambda$  from leaving the stability region. The meaning of *close enough* depends on the order. At order 3, the unstable region is much narrower than at order 5, so the potential for unstable behavior grows with order.

System eigenvalues that are likely to run into this instability are ones that correspond to weakly damped oscillations. A pure undamped oscillation corresponds to an eigenvalue on the imaginary axis. Problems with modes of that kind call for different considerations, since the oscillation generally must be followed by the solver, and this requires step sizes ( $h \sim 1/\nu$ , where  $\nu$  is the frequency) that are stable for BDF anyway. But for a weakly damped oscillatory mode, the oscillation in the solution is eventually damped to the noise level, and at that time it is important that the solver not be restricted to step sizes on the order of  $1/\nu$ . It is in this situation that the new option may be of great value.

In terms of partial differential equations, the typical problems for which the stability limit detection option is appropriate are semi-discrete ODE systems (i.e. discretized in space) from PDEs with advection and diffusion, but with advection dominating over diffusion. Diffusion alone produces pure decay modes, while advection tends to produce undamped oscillatory modes. A mix of the two with advection dominant will have weakly damped oscillatory modes.

The STALD algorithm attempts to detect, in a direct manner, the presence of a stability region boundary that is limiting the step sizes in the presence of a weakly damped oscillation [10]. The algorithm supplements (but differs greatly from) the existing algorithms in CVODES for choosing step size and order based on estimated local truncation errors. The STALD algorithm works directly

with history data that is readily available in CVODES. If it concludes that the step size is in fact stability-limited, it dictates a reduction in the method order, regardless of the outcome of the error-based algorithm. The STALD algorithm has been tested in combination with the VODE solver on linear advection-dominated advection-diffusion problems [11], where it works well. The implementation in CVODES has been successfully tested on linear and nonlinear advection-diffusion problems, among others.

This stability limit detection option adds some overhead computational cost to the CVODES solution. (In timing tests, these overhead costs have ranged from 2% to 7% of the total, depending on the size and complexity of the problem, with lower relative costs for larger problems.) Therefore, it should be activated only when there is reasonable expectation of modes in the user's system for which it is appropriate. In particular, if a CVODES solution with this option turned off appears to take an inordinately large number of steps at orders 3-5 for no apparent reason in terms of the solution time scale, then there is a good chance that step sizes are being limited by stability, and that turning on the option will improve the efficiency of the solution.

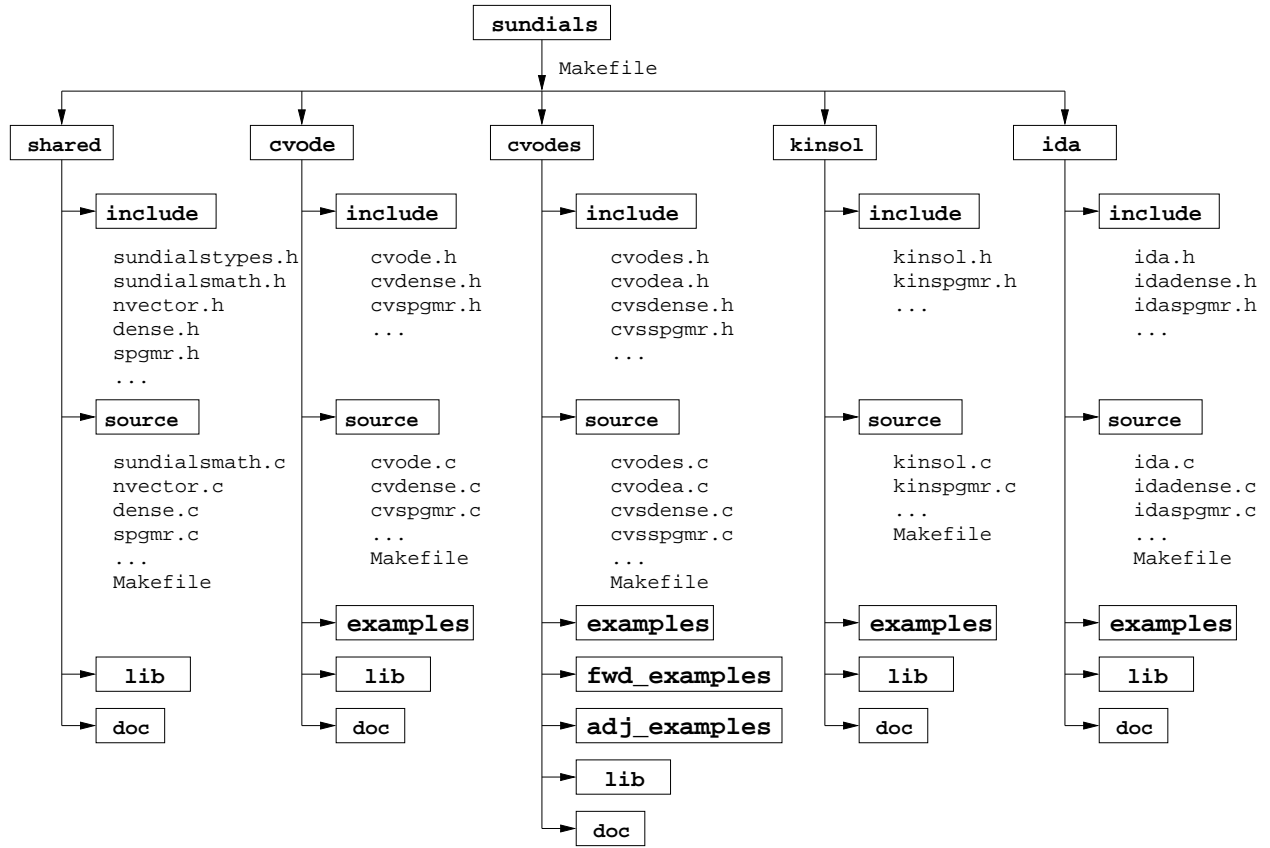


Figure 1: Organization of the SUNDIALS suite

## 3 Code Organization

### 3.1 SUNDIALS Organization

The family of solvers referred to as SUNDIALS consists of solvers CVODE (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, variants of these which also do sensitivity analysis calculations are available or in development. CVODES, an extension of CVODE that provides both forward and adjoint sensitivity capabilities is available, while IDAS and KINSOLS are currently in development.

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 1). The following is a list of the solver packages presently available:

- CVODE, a solver for stiff and nonstiff ODE systems  $dy/dt = f(t, y)$ ;
- CVODES, a solver for stiff and nonstiff ODE systems  $dy/dt = f(t, y)$  with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems  $F(u) = 0$ ;
- IDA, a solver for differential-algebraic systems  $F(t, y, y') = 0$ .

### 3.2 CVODES Organization

The CVODES package is written in the ANSI C language. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the CVODES package is shown in Figure 2. The basic elements of the structure are a module for the basic integration algorithm (including forward sensitivity analysis), a module for adjoint sensitivity analysis, and a set of modules for the solution of linear systems that arise in the case of a stiff system.

The central integration module, implemented in the files `cvodes.h` and `cvodes.c`, deals with the evaluation of integration coefficients, the functional or Newton iteration process, estimation of local error, selection of stepsize and order, and interpolation to user output points, among other issues. Although this module contains logic for the basic Newton iteration algorithm, it has no knowledge of the method being used to solve the linear systems that arise. For any given user problem, one of the linear system modules is specified, and is then invoked as needed during the integration.

In addition, if forward sensitivity analysis is turned on, the main module will integrate the forward sensitivity equations, simultaneously with the original IVP. The sensitivities variables may or may not be included in the local error control mechanism of the main integrator. CVODES provides three different strategies of dealing with the correction stage for the sensitivity variables, **SIMULTANEOUS**, **STAGGERED**, and **STAGGERED1** (see §2.2 and §5.2.1). The CVODES package includes an algorithm for the approximation of the sensitivity equations right hand sides by difference quotients, but the user has the option of supplying these right hand sides directly.

The adjoint sensitivity module provides the infrastructure needed for the integration backwards in time of any system of ODEs which depends on the solution of the original IVP, in particular the adjoint system and any quadratures required in evaluating the gradient of the objective functional. This module deals with the set-up of the check points, interpolation of the forward solution during the backward integration, and backward integration of the adjoint equations.

At present, the package includes the following four CVODES linear system modules:

- CVDENSE: LU factorization and backsolving with dense matrices;
- CVBAND: LU factorization and backsolving with banded matrices;
- CVDIAG: an internally generated diagonal approximation to the Jacobian;
- CVSPGMR: scaled preconditioned GMRES method.

This set of linear solver modules is intended to be expanded in the future as new algorithms are developed.

In the case of the direct CVDENSE and CVBAND methods, the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. In the case of the iterative CVSPGMR method, the package includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector of appropriate length. Again, the user has the option of providing a routine for this operation. In the case of CVSPGMR, the preconditioning must be supplied by the user, in two phases: setup (preprocessing of Jacobian data) and solve. While there is no default choice of preconditioner analogous to the difference quotient approximation

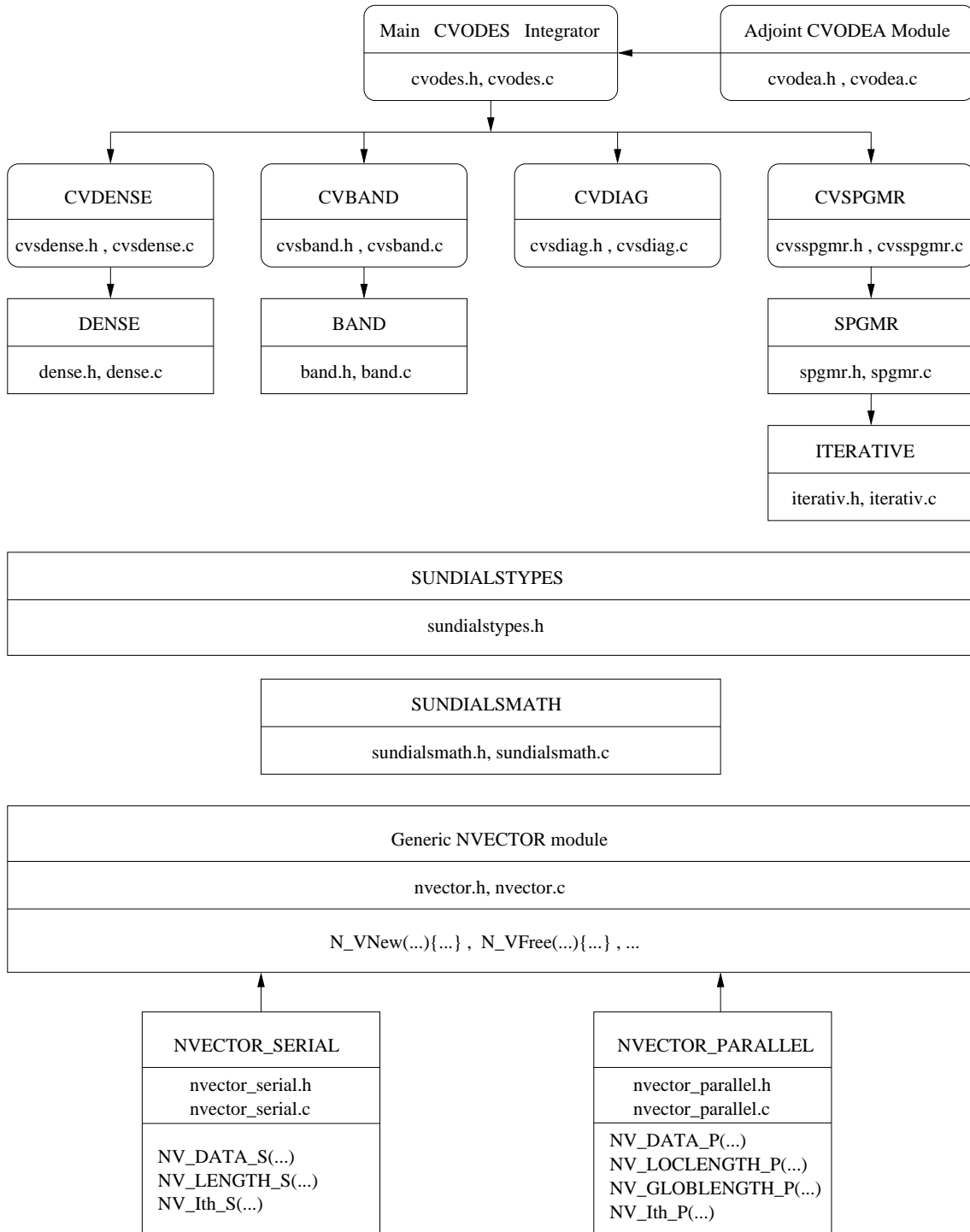


Figure 2: Overall structure diagram of the CVODES package. Modules specific to CVODES are distinguished by rounded boxes, while generic solver and auxiliary modules are in unrounded boxes.



in the direct case, the references [2]-[3], together with the example and demonstration programs included with CVODES, offer considerable assistance in building preconditioners.

Each CVODES linear solver module consists of five routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, (4) solution of the system in the context of forward sensitivity analysis, and (5) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration, as required to achieve convergence. The call list within the central CVODES module to each of the five associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. Each of the modules CVDENSE, CVBAND, and CVSPGMR is a set of interface routines built on top of a generic solver module, named DENSE, BAND, and SPGMR, respectively. The interfaces deal with the use of these methods in the CVODES context, whereas the generic solver is independent of the context. While the generic solvers here were generated with SUNDIALS in mind, our intention is that they be usable in other applications as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the CVODES package elsewhere.

CVODES also provides two preconditioner modules. The first one, CVBANDPRE, is intended to be used with NVECTOR\_SERIAL and provides a banded difference quotient Jacobian based preconditioner and solver routines for use with CVSPGMR. The second preconditioner module, CVBBDPRE, works in conjunction with NVECTOR\_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix.

All state information used by CVODES to solve a given problem is saved in a structure, and a pointer to that structure is returned to the user. There is no global data in the CVODES package, and so in this respect it is reentrant. State information specific to the linear solver is saved in separate structure, a pointer to which resides in the CVODES memory structure. The reentrancy of CVODES was motivated by the anticipated multicomputer extension, but is also essential in a uniprocessor setting where two or more problems are solved by intermixed calls to the package from one user program.

Table 1 below is a complete list of files in the CVODES package and the routines in each file. Header and source files are described as a single unit.

Table 1: List of files in the CVODES package

File(s)	Description and Contents
cvodes.h, .c	Central CVODES integrator module CVodeMalloc, CVReInit, CVode, CVodeFree, CVodeDky, CVodeSensMalloc, CVSensReInit, CVodeSensExtract, CVodeSensDkyAll, CVodeSensDky, CVodeMemExtract, CVSensRhsDQ, CVSensRhs1DQ
cvodea.h, .c	Adjoint sensitivity CVODES module CVadjMalloc, CVodeF, CVDenseB, CVBandB, CVBandPreAllocB, CVSpgrmrB CVodeMallocB, CVodeB, CVadjFree, CVadjGetY, CVadjCheckPointsList
cvsdense.h, .c	CVODES dense linear solver CVDENSE CVDense, CVReInitDense, CVDenseDQJac
cvsband.h, .c	CVODES band linear solver CVBAND CVBand, CVReInitBand, CVBandDQJac
cvdiag.h, .c	CVODES diagonal linear solver CVDIAG CVDiag
cvsspgmr.h, .c	CVODES GMRES linear solver CVSPGMR CVSpgrmr, CVReInitSpgrmr, CVSpgrmrDQJtimes
cvsbandpre.h, .c	Band preconditioner module CVBANDPRE CVBandPreAlloc, CVReInitBandPre, CVBandPreFree CVBandPrecond, CVBandPSolve
cvsbbdpre.h, .c	Band-block-diagonal preconditioner module CVBBDPRE CVBBDAAlloc, CVReInitBBD, CVBBDFree CVBBDPrecon, CVBBDPSo1

## 4 Using CVODES for IVP Solution

This section is concerned with the use of CVODES for the integration of IVPs. The following subsections treat the header files, the layout of the user's main program, description of the CVODES user-callable routines, and user-supplied functions or routines. The listings of the sample programs in §7 may also be helpful. Those codes are intended to serve as templates and are included in the CVODES package.

The user should be aware that not all linear solver modules are compatible with all NVECTOR implementations. For example, NVECTOR\_PARALLEL is not compatible with the direct dense or direct band linear solvers since these linear solver modules need to form the system Jacobian. The following CVODES modules can only be used with NVECTOR\_SERIAL: CVDENSE, CVBAND, and CVBANDPRE. The preconditioner module CVBBDPRE can only be used with NVECTOR\_PARALLEL.

### 4.1 Header Files

The calling program must include several header files so that various macros and data types can be used. The header files that are always required are:

- `sundialtypes.h`, which defines the types `realtype`, `integertype`, `booleantype` and constants `FALSE` and `TRUE`;
- `cvodes.h`, the header file for CVODES, which defines the several types and various constants, and includes function prototypes.

The calling program must also include an NVECTOR implementation header file (see §11 for details). For the two NVECTOR implementations that are included in the CVODES package, the corresponding header files are:

- `nvector_serial.h`, which defines the serial implementation NVECTOR\_SERIAL;
- `nvector_parallel.h`, which defines the parallel MPI implementation, NVECTOR\_PARALLEL.

Note that both these files include in turn the header file `nvector.h` which defines the abstract `N_Vector` and `M_Env` types.

Finally, if the user chooses Newton iteration for the solution of the nonlinear systems, then a linear solver module header file will be required. The header files corresponding to the various linear solver options in CVODES are:

- `cvdense.h`, which is used with the dense direct linear solver in the context of CVODES. This in turn includes a header file (`dense.h`) which defines the `DenseMat` type and corresponding accessor macros;
- `cvband.h`, which is used with the band direct linear solver in the context of CVODES. This in turn includes a header file (`band.h`) which defines the `BandMat` type and corresponding accessor macros;
- `cvdiag.h`, which is used with a diagonal linear solver in the context of CVODES;

- `cvsspgmr.h`, which is used with the Krylov solver SPGMR in the context of CVODES. This in turn includes a header file (`iterativ.h`) which enumerates the kind of preconditioning and the choices for the Gram-Schmidt process.

Other headers may be needed, according as to the choice of preconditioner, etc. In one of the examples to follow, preconditioning is done with a block-diagonal matrix. For this, the header `smalldense.h` is included.

## 4.2 A Skeleton of the User's Main Program

A high-level view of the combined user program and CVODES package is shown in Figure 3. The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP. Most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with CVODES: steps marked with [P] correspond to NVECTOR\_PARALLEL, while steps marked with [S] correspond to NVECTOR\_SERIAL.

1. [P] `MPI_Init(&argc, &argv)`; to initialize MPI if used by the user's program, aside from the internal use in NVECTOR\_PARALLEL. Here `argc` and `argv` are the command line argument counter and array received by `main`.
2. Set the problem dimensions:
  - [S] Set `N`, the problem size  $N$ .
  - [P] Set `Nlocal`, the local vector length (the sub-vector length for this processor); `N`, the global vector length (the problem size  $N$ , and the sum of all the values of `Nlocal`); and the active set of processors.
3. Initialize the machine environment variable:
  - [S] `machEnv = M_EnvInit_Serial(N)`;
  - [P] `machEnv = M_EnvInit_Parallel(comm, Nlocal, N, &argc, &argv)`; Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active processors is to be used, `comm` must be set by suitable MPI calls. Otherwise, to specify that all processors are to be used, `comm` must be `MPI_COMM_WORLD`.
4. Set the vector `y0` of initial values. Use macros defined by a particular NVECTOR implementation:
  - [S] `NV_MAKE_S(y0, ydata, machEnv)`;
  - [P] `NV_MAKE_P(y0, ydata, machEnv)`;

if an existing real array `ydata` contains the initial values of  $y$ . Otherwise, make the call `y0 = NV_New(N, machEnv)`; and load initial values into the real array defined by:

  - [S] `NV_DATA_S(y0)`
  - [P] `NV_DATA_P(y0)`

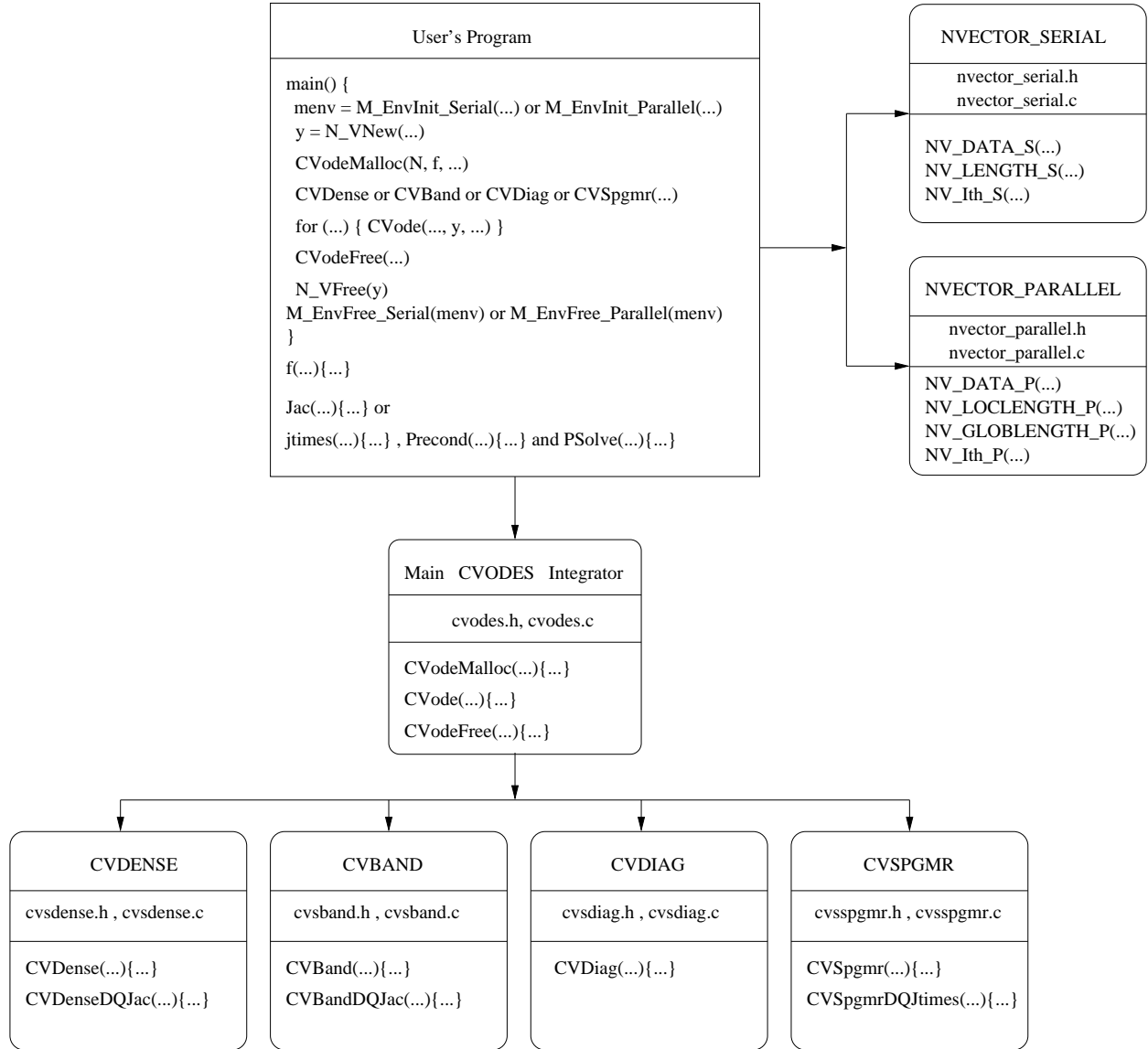


Figure 3: Diagram of the user program and CVODES package for integration of IVP

5. Call `cvode_mem = CVodeMalloc(...)`; to provide problem specifications, allocate internal memory for CVODES, provide solution method options and tolerances, and initialize CVODES. `CVodeMalloc` returns a pointer to the CVODES memory structure (for details see §4.3.1).
6. If Newton iteration is chosen, initialize the linear solver module with one of the following calls (for details see §4.3.2):
  - `[S] ier = CVDense(...)`;
  - `[S] ier = CVBand(...)`;
  - `ier = CVDiag(...)`;
  - `ier = CVSpgmr(...)`;
7. For each point at which output is desired, call `ier = CVode(cvode_mem, tout, y, &t, itask)`; Set `itask` to `NORMAL` to have the integrator overshoot `tout` and interpolate, or `ONE_STEP` to take a single step and return. The vector `y` (which can be the same as the vector `y0` above) will contain  $y(t)$ .
8. Upon completion of the integration, deallocate memory for the vector `y` by either calling a macro defined by the `NVECTOR` implementation:
  - `[S] NV_DISPOSE_S(y)`;
  - `[P] NV_DISPOSE_P(y)`;

if `y` was created from `ydata`, or by making the call `N_VFree(y)`; if `y` was created by a call to `N_VNew`.
9. `CVodeFree(cvode_mem)`; to free the memory allocated for CVODES.
10. Free the machine environment variable:
  - `[S] M_EnvFree_Serial(machEnv)`;
  - `[P] M_EnvFree_Parallel(machEnv)`;

## 4.3 User-Callable Routines for IVP Solution

### 4.3.1 CVODES Initialization Routine

The form of the call to `CVodeMalloc` (step 5) is

```
cvode_mem = CVodeMalloc(N, f, t0, y0, lmm, iter, itol, &rtol,
                        atol, f_data, errfp, optIn, iopt, ropt, machEnv);
```

where `N` is the number of ODEs in the system, `f` is the C function to compute  $f$  in the ODE, `t0` is the initial value of  $t$  and `y0` is the initial value of  $y$ . `f` has the form `f(N, t, y, ydot, f_data)` (for full details see §4.4). The flag `lmm` is used to select the linear multistep method and may be one of two possible values: `ADAMS` or `BDF`. The type of iteration is selected by replacing `iter` with either `NEWTON` or `FUNCTIONAL`. The typical choices for `(lmm, iter)` are `(ADAMS, FUNCTIONAL)` for nonstiff problems and `(BDF, NEWTON)` for stiff problems. The next three parameters are used

to set the error control. The flag `itol` is replaced by either `SS` or `SV`, where `SS` indicates scalar relative error tolerance and scalar absolute error tolerance, while `SV` indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the ODE. The arguments `&rtol` and `atol` are pointers to the user's error tolerances, and `f_data` is a pointer to user-defined space passed directly to the user's `f` function. The file pointer `errfp` points to the file where error messages from CVODES are to be written (NULL for `stdout`). The final argument, `machEnv`, is a pointer to machine environment-specific information.

Provision is made for certain optional inputs and optional outputs. Optional inputs communicated in the `CVodeMalloc` call are placed in the arrays `iopt` and `ropt`. These include the maximum order, the tentative initial stepsize, and the maximum stepsize. Each CVODES linear solver may or may not have optional inputs, which are passed through the associated initialization call list. Of the existing four linear solvers, only CVSPGMR has optional inputs. In any case, there is a default available for every optional input. Optional outputs from the central CVODES module are also communicated through the `iopt` and `ropt` arrays which are passed to `CVodeMalloc`. They include step and function evaluation counts, current stepsize and order, and workspace lengths. Optional outputs specific to each linear solver are loaded into `iopt` and `ropt`, following those from the central integrator module. For full details on the optional inputs and outputs, see §4.3.4.

If `optIn` is `FALSE`, then CVODES assumes that the user is not providing any optional input, while if it is `TRUE` then all optional inputs are examined in `iopt` and `ropt`.

If there was a failure, the return value of `CVodeMalloc` is NULL and an error message is printed.

### 4.3.2 Linear Solver Specification Routines

As previously explained, Newton iteration requires the solution of linear systems of the form (5). There are four CVODES linear solvers currently available for this task: `CVDENSE`, `CVBAND`, `CVDIAG`, and `CVSPGMR`. The first three are direct solvers and derive their name from the type of approximation used for the Jacobian  $J = \partial f / \partial y$ . `CVDENSE`, `CVBAND`, and `CVDIAG` work with dense, banded, and diagonal approximations to  $J$ , respectively. The fourth CVODES linear solver, `CVSPGMR`, is an iterative solver. The SPGMR in the name indicates that it uses a scaled preconditioned GMRES method.

To specify a CVODES linear solver, after the call to `CVodeMalloc` but before any calls to `CVode`, the user's program must call one of the functions `CVDense`, `CVBand`, `CVDiag`, `CVSpgmr`, as documented below. The first argument passed to these functions is the CVODES memory pointer returned by `CVodeMalloc`. A call to one of these functions links the main CVODES integrator to a linear solver and allows the user to specify parameters which are specific to a particular solver, such as the bandwidths in the `CVBAND` case.

The use of each of the linear solvers involves certain constants (such as locations of optional outputs in `iopt`), and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

In each case except the diagonal approximation case `CVDIAG`, the linear solver module used by CVODES is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted `DENSE`, `BAND`, and `SPGMR`, are described separately in §13.

- *Dense linear solver specification*

In using the CVDENSE solver with CVODES, the calling program must include the corresponding header file, with the line

```
#include "cvsdense.h"
```

After the call to `CVodeMalloc`, the user must call the routine `CVDense` to select the CVDENSE solver. The call to this routine has the following form:

```
ier = CVDense(cvode_mem, djac, jac_data);
```

Note that the CVDENSE linear solver may not be compatible with a particular implementation of the NVECTOR module. Of the two NVECTOR modules provided by SUNDIALS, only NVECTOR\_SERIAL is compatible, while NVECTOR\_PARALLEL is not.

The CVDENSE solver needs a routine to compute a dense approximation to the Jacobian matrix  $J(t, y)$ . This routine must be of type `CVDenseJacFn`, and is communicated through the `CVDense` formal parameter `djac` (see §4.4 for specification details). The user can supply his/her own dense Jacobian routine, or use the difference quotient routine `CVDenseDQJac` that comes with the CVDENSE solver. To use `CVDenseDQJac`, the user must pass NULL for the `djac` parameter.

The `CVDense` formal parameter `jac_data` is a pointer that accommodates a user-defined data structure. The CVDENSE solver passes the pointer it receives in the `CVDense` call to its dense Jacobian function (the `djac` parameter). This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian routine, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter is passed to `CVodeMalloc`.

The return value `ier` of `CVDense` is

- SUCCESS if the CVDENSE initialization was successful;
- LMEM\_FAIL if `cvode_mem` was NULL, if the NVECTOR module is incompatible with CVDENSE, or if there was a memory allocation failure.

The CVDENSE module provides three optional outputs. One is the number of calls made to the Jacobian routine. It is placed in `iopt[DENSE_NJE]`, where `iopt` is the array supplied by the user in the `CVodeMalloc` call. The other two are the sizes of the real and integer workspaces used by CVDENSE, stored in `iopt[DENSE_LRW]` and `iopt[DENSE_LIW]`, respectively. In terms of the problem size  $N$ , the actual sizes of these workspaces are  $2N^2$  realtype words and  $N$  integertype words.

- *Banded linear solver specification*

In using the CVBAND solver with CVODES, the calling program must include the corresponding header file, with the line

```
#include "cvsband.h"
```

After the call to `CVodeMalloc`, the user must call the routine `CVBand` to select the CVBAND solver. The call to this routine has the following form:



```
ier = CVBand(cvode_mem, mupper, mlower, bjac, jac_data);
```

The upper and lower half-bandwidths of problem Jacobian (or of the approximation of it to be used in CVODES) are specified in this call through the `mupper` and `mlower` parameters.

Note that the CVBAND linear solver may not be compatible with a particular implementation of the NVECTOR module. Of the two NVECTOR modules provided by SUNDIALS, only NVECTOR\_SERIAL is compatible, while NVECTOR\_PARALLEL is not.

The CVBAND solver requires a routine to compute a banded approximation to the Jacobian matrix  $J(t, y)$ . This routine must be of type `CVBandJacFn`, and is communicated through the `CVBand` formal parameter `bjac` (see §4.4 for specification details). The user can supply his/her own banded Jacobian approximation routine, or use the difference quotient routine `CVBandDQJac` that comes with the CVBAND solver. To use the `CVBandDQJac`, the user must pass `NULL` for `bjac`.

As in the CVDENSE case, the `CVBand` formal parameter `jac_data` is a pointer to a user-defined data structure, which the CVBAND solver passes to the Jacobian function `bjac`. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian routine, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter is passed to `CVodeMalloc`.

The return value `ier` of `CVBand` is

- `SUCCESS` if the CVBAND initialization was successful;
- `LMEM_FAIL`: if `cvode_mem` was `NULL`, if the NVECTOR module is incompatible with CVBAND, or if there was a memory allocation failure;
- `LIN_ILL_INPUT` if there was an illegal input.

The CVBAND module provides three optional outputs. One is the number of calls made to the Jacobian routine. It is placed in `iopt[BAND_NJE]`, where `iopt` is the array supplied by the user in the `CVodeMalloc` call. The other two are the sizes of the real and integer workspaces used by CVBAND, stored in `iopt[BAND_LRW]` and `iopt[BAND_LIW]`, respectively. In terms of the problem size  $N$ , the actual sizes of these workspaces are (roughly)  $N * (2 \text{ mupper} + 3 \text{ mlower} + 2)$  realtype words and  $N$  integertype words.

- *Diagonal linear solver specification*

In using the CVDIAG solver with CVODES, the calling program must include the corresponding header file, with the line

```
#include "cvdiag.h"
```

After the call to `CVodeMalloc`, the user must call the routine `CVDiag` to select the CVDIAG solver. The call to this routine has the following form:

```
ier = CVDiag(cvode_mem);
```

The CVDIAG solver is the simplest of all the current CVODES linear solvers. The `CVDiag` routine receives only the CVODES memory pointer returned by `CVodeMalloc`. The CVDIAG solver uses an approximate diagonal Jacobian formed by way of a difference quotient. The user does *not* have the option to supply a routine to compute an approximate diagonal Jacobian.

The return value `ier` of `CVDiag` is

- `SUCCESS` if the CVDIAG initialization was successful;
- `LMEM_FAIL` if `cvode_mem` was `NULL`, or if there was a memory allocation failure.

The CVDIAG module provides two optional outputs. These are the sizes of the real and integer workspaces used by CVDIAG, stored in `iopt[DIAG_LRW]` and `iopt[DIAG_LIW]`, respectively. In terms of the problem size  $N$ , the actual sizes of these workspaces are  $3N$  realtype words and no integertype words. The number of approximate diagonal Jacobians formed is equal to `iopt[NSETUPS]`.

- *SPGMR linear solver specification*

The CVSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear system (5).

With this SPGMR method, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. For a given preconditioner matrix, the merits of left vs. right preconditioning are unclear in general, and the user should experiment with both choices. Performance will differ because the inverse of the left preconditioner is included in the linear system residual whose norm is being tested in the SPGMR algorithm. As a rule, however, if the preconditioner is the product of two matrices, we recommend that preconditioning be done either on the left only or the right only, rather than using one factor on each side.

In using the CVSPGMR solver with CVODES, the calling program must include two associated header files, with the lines

```
#include "iterativ.h"
#include "cvspgmr.h"
```

After the call to `CVodeMalloc`, the user must call the routine `CVSpgrmr` to select the CVDIAG solver. This routine has the following form:

```
ier = CVSpgrmr(cvode_mem, pretype, gstype, maxl, delt, Precond,
               PSolve, P_data, jtimes, jac_data);
```

The call to `CVSpgrmr` is used to communicate the type of preconditioning (`pretype`), the user's preconditioner setup routine (`precond`), the preconditioner solve routine (`psolve`), and the type of Gram-Schmidt procedure (`gstype`). The `pretype` parameter can be `NONE`, `LEFT`, `RIGHT`, or `BOTH`. (These constants are defined in `iterativ.h`.) If no preconditioning is desired (pass `NONE` for `pretype`), then both `precond` and `psolve` are ignored. Otherwise, a preconditioner solve function `psolve` is required. Regardless of the type of preconditioning, a preconditioner setup function `precond` is *not* required. The `gstype` parameter can be `MODIFIED_GS` or `CLASSICAL_GS` (these constants are also defined in `iterativ.h`) according

to whether the user wants the CVSPGMR solver to use modified or classical Gram-Schmidt orthogonalization.

The call to `CVSpgrmr` is also used to communicate two optional inputs to the CVSPGMR solver. One is `maxl`, the maximum dimension of the Krylov subspace to be used. The other is `delt`, a factor by which the GMRES convergence test constant is reduced from the Newton iteration test constant. Both of these inputs have defaults, which can be invoked by setting the actual parameter to zero in the call. The actual default values are 5 for the maximum Krylov dimension, and .05 for the test constant factor.

The routine `CVSpgrmr` takes in a parameter `P_data`, a pointer to a user-defined data structure, which the CVSPGMR solver passes to the preconditioner setup and solve functions `precond` and `psolve`. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner routines without using global data in the program. The pointer `P_data` may be identical to `f_data`, if the latter is passed to `CVodeMalloc`.

If any type of preconditioning is to be done within the SPGMR method, then the user must supply a preconditioner solve routine `psolve` (see §4.4). The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve routine is done in the optional user-supplied routine `precond` (see §4.4).

The CVSPGMR solver requires a routine to compute an approximation to the product between the Jacobian matrix  $J(t, y)$  and a vector  $v$ . This routine must be of type `CVSpgrmrJtimesFn`, and is communicated through the `CVSpgrmr` formal parameter `jtimes` (see §4.4 for specification details). The user can supply his/her own Jacobian times vector approximation routine, or use the difference quotient routine `CVSpgrmrDQJtimes` that comes with the CVSPGMR solver. To use the `CVSpgrmrDQJtimes`, the user must pass `NULL` for `jtimes`.

As in the `CVDENSE` and `CVBAND` cases, the `CVSpgrmr` formal parameter `jac_data` is a pointer to a user-defined data structure, which the CVSPGMR solver passes to the Jacobian times vector function `jtimes`. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian times vector routine, without using global data in the program. The pointer `jac_data` may be identical to `f_data`, if the latter is passed to `CVodeMalloc`.

The return value `ier` of `CVSpgrmr` is

- `SUCCESS` if the CVSPGMR initialization was successful;
- `LMEM_FAIL` if `cnode_mem` was `NULL`, or if there was a memory allocation failure;
- `LIN_ILL_INPUT` if there was an illegal input.

The CVSPGMR solver provides six optional outputs. The total number of calls to `precond` is given in `iopt[SPGMR_NPE]`, and the number of calls to `psolve` is in `iopt[SPGMR_NPS]`. The number of linear iterations is in `iopt[SPGMR_NLI]`, and the number of linear convergence failures is in `iopt[SPGMR_NCFL]`. The sizes of the real and integer workspaces used by CVSPGMR are stored in `iopt[SPGMR_LRW]` and `iopt[SPGMR_LIW]`, respectively. In terms of the problem size  $N$  and the maximum Krylov dimension  $\ell_{max}$ , the actual sizes of these workspaces are  $N * (\ell_{max} + 5) + \ell_{max} * (\ell_{max} + 4) + 1$  realtype words and no integertype words.

For users interested in the generic SPGMR solver used by CVSPGMR, a note of caution is in order: the routines in SPGMR have arguments `l_max`, `delta`, `psolve`, `P_data`, which are *not* the same as the CVSpgr arguments `maxl`, `delt`, `psolve`, `P_data`, although the names are the same or very similar. The arguments `pretype` and `gstype` are identical in meaning in both contexts. For more on the generic SPGMR solver, see §13.3.

### 4.3.3 CVODES Solver Routine

The call to the `CVode` function itself has the form

```
ier = CVode(cvode_mem, tout, y, &t, itask);
```

In addition to the CVODES memory pointer `cvode_mem`, it specifies only two inputs: (1) a flag `itask` showing whether the integration is to be done in the “normal mode” or in the “one-step mode” and (2) a value, `tout`, of the independent variable  $t$  at which a computed solution is desired. In the normal mode, the integration proceeds in steps (with stepsizes determined internally) up to and past `tout`, and `CVode` interpolates  $y$  at  $t = \text{tout}$ . In the one-step mode, `CVode` takes only one step in the desired direction and returns to the calling program. In the one-step mode, `tout` is required on the first call only, to get the direction and rough scale of the independent variable. On return, `CVode` returns a vector  $y$  and a corresponding independent variable value  $t = *t$ , such that  $y$  is the computed value of  $y(t)$ . In the normal mode, with no failures,  $*t$  will be equal to `tout`.

Note that the vector  $y$  can be the same as the  $y_0$  vector of initial conditions that was passed to `CVodeMalloc`.

The return value `ier` for `CVode` will be one of the following:

- `SUCCESS=0`: `CVode` succeeded;
- `TSTOP_RETURN=1`: `CVode` succeeded by reaching the stopping point specified through the optional inputs `iopt[ISTOP]` and `ropt[TSTOP]` (see §4.3.4);
- `CVODE_NO_MEM`: The `cvode_mem` argument was `NULL`;
- `ILL_INPUT`: One of the inputs to `CVode` is illegal. This includes the situation when a component of the error weight vectors becomes negative during internal time-stepping. The `ILL_INPUT` flag will also be returned if the linear solver routine initialization (called by the user after calling `CVodeMalloc`) failed to set one of the linear solver-related fields in `cvode_mem` or if the linear solver’s initialization routine failed. In any case, the user should see the printed error message for more details;
- `TOO_MUCH_WORK`: The solver took `mxstep` internal steps but could not reach `tout`. The default value for `mxstep` is `MXSTEP_DEFAULT = 500`;
- `TOO_MUCH_ACC`: The solver could not satisfy the accuracy demanded by the user for some internal step;
- `ERR_FAILURE`: Error test failures occurred too many times (`MXNEF = 7`) during one internal time step or occurred with  $|h| = h_{min}$ ;
- `CONV_FAILURE`: Convergence test failures occurred too many times (`MXNCF = 10`) during one internal time step or occurred with  $|h| = h_{min}$ ;

- **SETUP\_FAILURE**: The linear solver's setup routine failed in an unrecoverable manner;
- **SOLVE\_FAILURE**: The linear solver's solve routine failed in an unrecoverable manner.

All failure return values are negative and therefore a test `ier < 0` will trap all **CVode** failures.

#### 4.3.4 Optional Input/Output

In order to change some of the **CVODES** constants (such as the maximum method order) or if additional diagnostic output values are desired, the user should declare two arrays for optional input and output, an `iopt` array for optional integer input and output and an `ropt` array for optional real input and output. The size of both these arrays should be `OPT_SIZE`. So the user's declarations should look like:

```
long int iopt[OPT_SIZE];
realtype ropt[OPT_SIZE];
```

Tables 2 and 3 contain detailed descriptions of the optional integer and real input-output arrays, respectively. Only locations corresponding to the main **CVODES** solver are given in these tables. Locations beyond `CVODE_IOPT_SIZE` and `CVODE_ROPT_SIZE` in `iopt` and `ropt`, respectively, are used by the linear solvers and are described in §4.3.2.

Default values of the optional inputs are obtained by setting the corresponding entry to 0. If **FALSE** is passed for `optIn` in the call to **CVodeMalloc**, no optional input is examined. Note also that when computing forward sensitivities, **CVODES** loads some additional optional output entries in `iopt`. These are described in §5.2.3.

#### 4.3.5 Interpolated Output Routines

An optionally callable function **CVodeDky** is available to obtain additional output values. This function must be called after a successful return from **CVode** and provides interpolated values of  $y$  or its derivatives, up to the current order of the integration method, interpolated to any value of  $t$  in the last internal step taken by **CVODES**.

The call to the **CVodeDky** function has the form

```
ier = CVodeDky(cvode_mem, t, k, dky);
```

and computes the  $k$ -th derivative of the  $y$  function at time  $t$ , i.e.  $d^{(k)}y/dt^{(k)}(t)$ , where  $t_n - h_u \leq t \leq t_n$ ,  $t_n$  denotes the current internal time reached, and  $h_u$  is the last internal step size successfully used by the solver. The user may request  $k = 0, 1, \dots, q_u$ , where  $q_u$  is the current order. The derivative vector is returned in `dky`. This vector must be allocated by the caller. The first argument `cvode_mem` is the pointer to the **CVODES** memory returned by **CVodeMalloc**.

Note that it is only legal to call the function **CVodeDky** after a successful return from **CVode**. The return value `ier` for **CVodeDky** is

- **OKAY** if **CVodeDky** succeeded;
- **BAD\_K** if  $k$  is not in the range  $0, 1, \dots, q_u$ ;
- **BAD\_T** if  $t$  is not in the interval  $[t_n - h_u, t_n]$ ;

Table 2: Description of the optional integer input-output array `iopt`

Index	I/O	Default value	Description
MAXORD	I	12 (ADAMS) 5 (BDF)	Maximum <code>lmm</code> order to be used by the solver.
MXSTEP	I	500	Maximum number of internal steps to be taken by the solver in its attempt to reach <code>tout</code> .
MXHNIL	I	10	Maximum number of warning messages issued by the solver that $t + h == t$ on the next internal step. A value of -1 means no such messages are issued.
SLDET	I	0	Flag to turn on/off stability limit detection (1 = on, 0 = off). When BDF is used and order is 3 or greater, <code>CVslDET</code> is called to detect stability limit. If limit is detected, the order is reduced.
ISTOP	I	0	Flag to turn on/off testing for <code>tstop</code> (1=on, 0=off). When on, <code>CVODES</code> uses <code>ropt[TSTOP]</code> as the value <code>tstop</code> of the independent variable past which the solution is not to proceed.
NST	O		Cumulative number of internal steps taken by the solver (total so far).
NFE	O		Number of calls to the user's <code>f</code> function.
NSETUPS	O		Number of calls made to the linear solver's setup routine.
NNI	O		Number of nonlinear (FUNCTIONAL or NEWTON) iterations performed.
NCFN	O		Number of nonlinear convergence failures that have occurred.
NETF	O		Number of local error test failures that have occurred.
QU	O		Order used during the last internal step.
QCUR	O		Order to be used on the next internal step.
LENRW	O		Size of required <code>CVODES</code> internal real work space, in realtype words.
LENIW	O		Size of required <code>CVODES</code> internal integer work space, in integertype words.
NOR	O		Number of order reductions due to stability limit detection.

Table 3: Description of the optional real input-output array `ropt`

Index	I/O	Default value	Description
H0	I	computed	Initial step size.
HMAX	I	$\infty$	Maximum absolute value of step size allowed. Note: If <code>optIn=TRUE</code> , the value of <code>ropt[HMAX]</code> is examined on every call to <code>CVode</code> , and so can be changed between calls.
HMIN	I	0.0	Minimum absolute value of step size allowed.
TSTOP	I	–	The independent variable value past which the solution is not to proceed. Testing for this condition must be turned on through <code>iopt[ISTOP]</code> .
H0U	O		Actual initial step size used.
HU	O		Step size for the last internal step.
HCUR	O		Step size to be attempted on the next internal step.
TCUR	O		Current internal time reached by the solver.
TOLSF	O		A suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

- BAD\_DKY if the `dky` argument was NULL;
- DKY\_NO\_MEM if the `cvode_mem` argument was NULL.

#### 4.3.6 CVODES Reinitialization Routine

The function `CVReInit` reinitializes the main CVODES solver for the solution of a problem, where a prior call to `CVodeMalloc` has been made with the same problem size `N`. `CVReInit` performs the same input checking and initializations that `CVodeMalloc` does (except for `N`), but does no memory allocation, assuming that the existing internal memory is sufficient for the new problem.

The use of `CVReInit` requires that the maximum method order, `maxord`, is no larger for the new problem than for the problem specified in the last call to `CVodeMalloc`. This condition is automatically fulfilled if the multistep method parameter `lmm` is unchanged (or changed from `ADAMS` to `BDF`) and the default value for `maxord` is specified.

If `iter = NEWTON`, then following the call to `CVReInit`, a call to the linear solver specification routine is necessary if a different linear solver is chosen, but may not be otherwise. If the same linear solver is chosen, and there are no changes in the input parameters to the specification routine, then no call to that routine is needed. If there are changes in parameters, but they do not increase the linear solver memory size, then a call to the corresponding `CVReInit<lin sol>` routine must be made to communicate the new parameters (see §4.3.7); in that case the linear solver memory is reused. If the parameter changes do increase the linear solver memory size, then the main linear solver specification routine must be called (§4.3.2).

The call to the `CVReInit` function has the form

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

Its first argument, `cvode_mem` is the pointer to the CVODES memory returned by `CVodeMalloc`. All the remaining arguments to `CVReInit` have names and meanings identical to those of `CVodeMalloc`. Note that the problem size `N` is not passed as an argument to `CVReInit`, as that is assumed to be unchanged since the `CVodeMalloc` call.

The return value `ier` of `CVReInit` is equal to:

- `SUCCESS=0` if there were no errors;
- `CVREI_NO_MEM` if `cvode_mem` was `NULL`;
- `CVREI_ILL_INPUT` if an input argument was illegal (including an attempt to increase `maxord`).

In case of an error return, an error message is also printed.

Finally, note that the reported workspace sizes `iopt[LENRW]` and `iopt[LENIW]` are left unchanged from the values computed by `CVodeMalloc`, and so may be larger than would be computed for the new problem.

#### 4.3.7 Linear Solver Reinitialization Routines

Linear solver reinitialization routines reset the link between the main CVODES integrator and the linear solver module. Such a routine must be called after a call to `CVReInit` to solve another problem of the same size if there is a change in some of the linear solver parameters (such as the Jacobian data approximation routine or the user-defined data structure). Reinitialization routines exist for all but the `CVDIAG` linear solver.

- *Dense linear solver reinitialization*

A call to the `CVReInitDense` function resets the link between the main CVODES integrator and the `CVDENSE` linear solver. After solving one problem using `CVDENSE`, call `CVReInit` and then `CVReInitDense` to solve another problem of the same size, if there is a change in the `CVDense` parameters `djac` or `jac_data`. If there is no change in parameters, it is not necessary to call either `CVReInitDense` or `CVDense` for the new problem.

The call to the `CVDENSE` reinitialization routine has the following form:

```
ier = CVReInitDense(cvode_mem, djac, jac_data);
```

All arguments to `CVReInitDense` have the same names and meanings as those of `CVDense`. The `cvode_mem` argument must be identical to its value in the previous `CVDense` call.

The return values of `CVReInitDense` are:

- `SUCCESS` if successful;
- `LMEM_FAIL` if the `cvode_mem` argument is `NULL`.



Note that `CVReInitDense` performs the same tests for a compatible `NVECTOR` module as `CVDense`.

- *Banded linear solver reinitialization*

A call to the `CVReInitBand` function resets the link between the main `CVODES` integrator and the `CVBAND` linear solver. After solving one problem using `CVBAND`, call `CVReInit` and then `CVReInitBand` to solve another problem of the same size, if there is a change in the `CVBand` parameters `bjac` or `jac_data`, but no change in `mupper` or `mlower`. If there is a change in `mupper` or `mlower`, then `CVBand` must be called again, and the linear solver memory will be reallocated. If there is no change in parameters, it is not necessary to call either `CVReInitBand` or `CVBand` for the new problem.

The call to the `CVBAND` reinitialization routine has the following form:

```
ier = CVReInitBand(cvode_mem, mupper, mlower, bjac, jac_data);
```

All arguments to `CVReInitBand` have the same names and meanings as those of `CVBand`. The `cvode_mem` argument must be identical to its value in the previous `CVBand` call.

The return values of `CVReInitBand` are:

- `SUCCESS` if successful;
- `LMEM_FAIL` if the `cvode_mem` argument is `NULL`;
- `LIN_ILL_INPUT` if there was an illegal input.

Note that `CVReInitBand` performs the same tests for a compatible `NVECTOR` module as `CVBand`.

- *SPGMR linear solver reinitialization*

A call to the `CVReInitSpgmr` function resets the link between the main `CVODES` integrator and the `CVSPGMR` linear solver. After solving one problem using `CVSPGMR`, call `CVReInit` and then `CVReInitSpgmr` to solve another problem of the same size, if there is a change in the `CVSpgmr` parameters `pretype`, `gstype`, `delt`, `precond`, `psolve`, `P_data`, `jtimes`, or `jac_data`, but not in `maxl`. If there is a change in `maxl`, then `CVSpgmr` must be called again, and the linear solver memory will be reallocated. If there is no change in parameters, it is not necessary to call either `CVReInitSpgmr` or `CVSpgmr` for the new problem.

The call to the `CVSPGMR` reinitialization routine has the following form:

```
ier = CVReInitSpgmr(cvode_mem, pretype, gstype, maxl, delt, Precond,  
                    PSolve, P_data, jtimes, jac_data);
```

All arguments to `CVReInitSpgmr` have the same names and meanings as those of `CVSpgmr`. The `cvode_mem` argument must be identical to its value in the previous `CVSpgmr` call.

The return values of `CVReInitSpgmr` are:

- `SUCCESS` if successful;
- `LMEM_FAIL` if the `cvode_mem` argument is `NULL`;
- `LIN_ILL_INPUT` if there was an illegal input.

## 4.4 User-Supplied Routines for IVP Solution

The user-supplied routines consist of one function defining the ODE, (optionally) a function that provides Jacobian related information for the linear solver (if Newton iteration is chosen), and (optionally) one or two functions that define the preconditioner for use in the SPGMR algorithm.

- *ODE right hand side*

The user must provide a function of type `RhsFn` defined by

```
typedef void (*RhsFn)(integertype N, realtype t, N_Vector y,
                     N_Vector ydot, void *f_data);
```

to compute the right hand side of the ODE system.

This function takes as input the problem size `N`, the independent variable value `t`, and the dependent variable vector `y`. It must store the result of  $f(t, y)$  in the vector `ydot`. The `y` and `ydot` arguments are of type `N_Vector`. Allocation of memory for `ydot` is handled within `CVODES`. The `f_data` parameter is the same as the `f_data` parameter passed by the user to the `CVodeMalloc` routine. This user-supplied pointer is passed to the user's `f` function every time it is called. A `RhsFn` function type does not have a return value.

- *Jacobian information (direct method with dense Jacobian)*

If the direct linear solver with dense treatment of the Jacobian is used (i.e. `CVDense` is called in step 6 of §4.2), the user may provide a function of type `CVDenseJacFn` defined by

```
typedef void (*CVDenseJacFn)(integertype N, DenseMat J, RhsFn f,
                             void *f_data, realtype t, N_Vector y,
                             N_Vector fy, N_Vector ewt, realtype h,
                             realtype uround, void *jac_data,
                             long int *nfePtr, N_Vector vtemp1,
                             N_Vector vtemp2, N_Vector vtemp3);
```

to compute the dense Jacobian  $J = \partial f / \partial y$  (or an approximation to it).

A user-supplied dense Jacobian routine must load the `N` by `N` dense matrix `J` with an approximation to the Jacobian matrix  $J$  at the point  $(t, y)$ . Only nonzero elements need to be loaded into `J` because `J` is set to the zero matrix before the call to the Jacobian routine. The type of `J` is `DenseMat`. The accessor macros `DENSE_ELEM` and `DENSE_COL` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `DenseMat` type. `DENSE_ELEM(A, i, j)` references the  $(i, j)$ th element of the dense matrix `A` ( $i, j = 0..N-1$ ). This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices  $m$  and  $n$  running from 1 to  $N$ , the Jacobian element  $J_{m,n}$  can be loaded with the statement `DENSE_ELEM(A, m-1, n-1) = Jm,n`. Alternatively, `DENSE_COL(A, j)` returns a pointer to the storage for the  $j$ th column of `A`, and the elements of the  $j$ th column are then accessed via ordinary array indexing. Thus  $J_{m,n}$  can be loaded with the statements `col_n = DENSE_COL(J, n-1); col_n[m-1] = Jm,n`. For large problems, it is more efficient to use `DENSE_COL` than to use `DENSE_ELEM`. Note that both of

these macros number rows and columns starting from 0, not 1. The `DenseMat` type and the accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in §13.1.

Typically, a user-supplied Jacobian function `djac` would be expected to access the arguments `N`, `t`, `y`, `J`, `f_data`, and `jac_data`, at most. The remaining arguments would not typically be accessed, but appear in the call list because they are needed by the function `CVDenseDQJac` that computes a difference quotient approximation to  $J$ , when the user specifies that option.

- *Jacobian information (direct method with banded Jacobian)*

If the direct linear solver with banded treatment of the Jacobian is used (i.e. `CVBand` is called in step 6 of §4.2), the user may provide a function of type `CVBandJacFn` defined by

```
typedef void (*CVBandJacFn)(integertype N, integertype mupper,
                             integertype mlower, BandMat J, RhsFn f,
                             void *f_data, realtype t, N_Vector y,
                             N_Vector fy, N_Vector ewt, realtype h,
                             realtype uround, void *jac_data,
                             long int *nfePtr, N_Vector vtemp1,
                             N_Vector vtemp2, N_Vector vtemp3);
```

to generate the banded Jacobian  $J = \partial f / \partial y$  (or a banded approximation to it).

A user-supplied band Jacobian routine must load the band matrix `J` of type `BandMat` with the elements of the Jacobian  $J(t, y)$  at the point  $(t, y)$ . Only nonzero elements need to be loaded into `J` because `J` is preset to zero before the call to the Jacobian routine. The accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the `BandMat` type. `BAND_ELEM(A, i, j)` references the  $(i, j)$ th element of the band matrix `A`. This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices  $m$  and  $n$  running from 1 to  $N$  with  $(m, n)$  within the band defined by `mupper` and `mlower`, the Jacobian element  $J_{m,n}$  can be loaded with the statement `BAND_ELEM(A, m-1, n-1) = J_{m,n}`. The elements within the band are those with  $-mupper \leq m-n \leq mlower$ . Alternatively, `BAND_COL(A, j)` returns a pointer to the diagonal element of the  $j$ th column of `A`, and if we assign this address to `realtype *col_j`, then the  $i$ th element of the  $j$ th column is given by `BAND_COL_ELEM(col_j, i, j)`. Thus for  $(m, n)$  within the band,  $J_{m,n}$  can be loaded by setting `col_n = BAND_COL(J, n-1); BAND_COL_ELEM(col_n, m-1, n-1) = J_{m,n}`. The elements of the  $j$ th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `BandMat`. The array `col_n` can be indexed from  $-mupper$  to `mlower`. For large problems, it is more efficient to use the combination of `BAND_COL` and `BAND_COL_ELEM` than to use the `BAND_ELEM`. As in the dense case, these macros all number rows and columns starting from 0, not 1. The `BandMat` type and the accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` are documented in §13.2.

Typically, a user-supplied Jacobian function `bjac` would be expected to access the arguments `N`, `mupper`, `mlower`, `t`, `y`, `J`, `f_data`, and `jac_data`, at most. The remaining arguments would

not typically be accessed, but appear in the call list because they are needed by the function `CVBAndDQJac` that computes a difference quotient approximation to  $J$ , when the user specifies that option.

- *Jacobian information (SPGMR case)*

If an iterative SPGMR linear solver is selected (`CVSpgmr` is called in step 6 of §4.2) the user may provide a function of type `CVSpgmrJtimesFn` in the form

```
typedef int (*CVSpgmrJtimesFn)(integertype N, N_Vector v, N_Vector Jv,
                                RhsFn f, void *f_data, realtype t,
                                N_Vector y, N_Vector fy,
                                realtype vnrm, N_Vector ewt, realtype h,
                                realtype ound, void *jac_data,
                                long int *nfePtr, N_Vector work);
```

to compute the product  $Jv = (\partial f / \partial y)v$  (or an approximation to it).

A user-supplied Jacobian-times-vector routine must load the vector `Jv` with the result of the product between the Jacobian  $J(t, y)$  at the point  $(t, y)$  and the vector `v` of dimension `N`.

Typically, a user-supplied Jacobian-times-vector function `jtimes` would be expected to access the arguments `N`, `v`, `t`, `y`, `Jv`, `f_data`, and `jac_data`, at most. The remaining arguments would not typically be accessed, but appear in the call list because they are needed by the function `CVSpgmrDQJtimes` that computes a difference quotient approximation to  $Jv$ , when the user specifies that option.

The value to be returned by the Jacobian times vector routine should be 0 if successful. Any other return value will result in an unrecoverable error of the SPGMR generic solver, in which case the integration is halted.

- *Preconditioning (linear system solution)*

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

```
typedef int (*CVSpgmrPSolveFn)(integertype N, realtype t, N_Vector y,
                                N_Vector fy, N_Vector vtemp, realtype gamma,
                                N_Vector ewt, realtype delta, long int *nfePtr,
                                N_Vector r, int lr, void *P_data, N_Vector z);
```

Its parameters are as follows:

- `N` is the length of all vector arguments;
- `t` is the current value of the independent variable;
- `y` is the current value of the dependent variable vector;
- `fy` is the vector  $f(t, y)$ ;

- `vtemp` is a pointer to memory allocated for a vector of length `N` which can be used for work space;
- `gamma` is the scalar appearing in the Newton matrix;
- `ewt` is the error weight vector (input). See `delta` below;
- `delta` is an input tolerance to be use if an iterative method is employed in the solution. In that case, the residual vector  $Res = r - Pz$  of the system should be made less than `delta` in weighted  $l_2$  norm, i.e.,  $\sqrt{\sum_i (Res_i \cdot ewt_i)^2} < delta$ ;
- `nfePtr` is a pointer to the memory location containing the CVODES problem data `nfe` = number of calls to `f`. The preconditioner solve routine should update this counter by adding on the number of `f` calls made in order to carry out the solution, if any. For example, if the routine calls `f` a total of `W` times, then the update is `*nfePtr += W`;
- `r` is the right-hand side vector of the linear system;
- `lr` is an input flag indicating whether the preconditioner solve routine is to use the left preconditioner (`lr=1`) or the right preconditioner (`lr=2`);
- `P_data` is a pointer to user data - the same as the `P_data` parameter passed to `CVSpgmr`;
- `z` is the output vector computed.

The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).

- *Preconditioning (Jacobian data)*

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then this needs to be done in a user-supplied C function of type `CVSpgmrPrecondFn` as defined by

```
typedef int (*CVSpgmrPrecondFn)(integertype N, realtype t, N_Vector y,
                                N_Vector fy, booleantype jok,
                                booleantype *jcurPtr, realtype gamma,
                                N_Vector ewt, realtype h, realtype uround,
                                long int *nfePtr, void *P_data,
                                N_Vector vtemp1, N_Vector vtemp2,
                                N_Vector vtemp3);
```

The operations performed by such a routine might include forming a crude approximate Jacobian, and performing an LU factorization on the resulting approximation to  $M = I - \gamma J$ .

This routine is not called in advance of every call to the preconditioner solve routine, but rather is called only as often as needed to achieve convergence in the Newton iteration.

The `jok` argument provides for the re-use of Jacobian data in the preconditioner solve routine. When `jok == FALSE`, Jacobian data should be computed from scratch, but when `jok == TRUE`, Jacobian data saved earlier can be retrieved and used to form the preconditioner matrices (with the current  $\gamma = \text{gamma}$ ). Each call to the preconditioner setup function is

preceded by a call to the **RhsFn** user routine with the same **(t,y)** arguments. Thus the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the ODE right hand side.

The error weight vector **ewt**, step size **h**, and unit roundoff **uround** are provided for possible use in approximating Jacobian data, e.g. by difference quotients.

The arguments of a **CVSpgmrPrecondFn** are as follows:

- **N** is the length of all vector arguments;
- **t** is the current value of the independent variable;
- **y** is the current value of the dependent variable vector, namely the predicted value of  $y(t)$ ;
- **fy** is the vector  $f(t, y)$ ;
- **jok** is an input flag indicating whether Jacobian-related data needs to be recomputed. **jok == FALSE** means that Jacobian-related data must be recomputed from scratch. **jok == TRUE** means that Jacobian data, if saved from the previous Precond call, can be reused (with the current value of **gamma**). A call with **jok == TRUE** can only occur after a call with **jok == FALSE**;
- **jcurPtr** is a pointer to an output integer flag which is to be set to **TRUE** if Jacobian data was recomputed or to **FALSE** if Jacobian data was not recomputed, but saved data was reused;
- **gamma** is the scalar appearing in the Newton matrix;
- **ewt** is the error weight vector;
- **h** is a tentative step size in **t**;
- **uround** is the machine unit roundoff;
- **nfePtr** is a pointer to the memory location containing the CVODES problem data **nfe** = number of calls to **f**. The preconditioner solve routine should update this counter by adding on the number of **f** calls made in order to carry out the solution, if any. For example, if the routine calls **f** a total of **W** times, then the update is **\*nfePtr += W**;;
- **P\_data** is a pointer to user data, the same as the **P\_data** parameter passed to **CVSpgmr**;
- **vtemp1**, **vtemp2**, and **vtemp3** are pointers to memory allocated for vectors of length **N** which can be used by **CVSpgmrPrecondFn** as temporary storage or work space.

The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).

## 4.5 CVODES Preconditioner Modules

### 4.5.1 A Serial Banded Preconditioner Module

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, CVODES provides a banded preconditioner in the module **CVBANDPRE**.

This preconditioner provides a band matrix preconditioner based on difference quotients of the ODE right-hand side function  $\mathbf{f}$ . It generates a band matrix of bandwidth  $m_l + m_u + 1$ , where the number of super-diagonals ( $m_u$ , the upper bandwidth) and sub-diagonals ( $m_l$ , the lower bandwidth) are specified by the user and uses this to form a preconditioner for use with the Krylov linear solver in CVSPGMR. Although this matrix is intended to approximate the Jacobian  $\partial \mathbf{f} / \partial \mathbf{y}$ , it may be a very crude approximation. The true Jacobian need not be banded, or its true bandwidth may be larger than  $m_l + m_u + 1$ , as long as the banded approximation generated here is sufficiently accurate to speed convergence as a preconditioner.

In order to use the CVBANDPRE module, the user needs not define any additional routines. The following is a summary of the usage of this module and describes the sequence of calls in the user main program.

- `#include "cvsbandpre.h"` for needed function prototypes and for type `CVBandPreData`;
- `#include "nvector_serial.h"` for the serial NVECTOR module;
- `M_Env machEnv;`
- `CVBandPreData bp_data;`
- `machEnv = M_EnvInit_Serial(N);` to initialize the serial machine environment;
- `cnode_mem = CVodeMalloc(N, f, ...);`
- `bp_data = CVBandPreAlloc(N, f, f_data, mu, ml, cnode_mem);`  
where the upper and lower half-bandwidths are `mu` and `ml`, respectively; `f_data` is a pointer to private data; and `cnode_malloc` is the pointer to CVODES memory returned by `CVodeMalloc`;
- `ier = CVSpmgr(cnode_mem, pretype, gstype, maxl, delt, CVBandPrecond, CVBandPSolve, bp_data, jtimes, jac_data);`  
with the pointers `cnode_mem` and `bp_data` returned by the two previous calls, the six SPGMR parameters (`pretype`, `gstype`, `maxl`, `delt`, `jtimes`, `jac_data`) and the names of the preconditioner routines (`CVBandPrecon`, `CVBandPSol`) supplied with the CVBANDPRE module;
- `ier = CVode(cnode_mem, tout, y, &t, itask);` to carry out the integration;
- `CVBandPreFree(bp_data);` to free the CVBANDPRE memory block;
- `CVodeFree(cnode_mem);` to free the CVODES memory block;
- `M_EnvFree_Serial(machEnv);` to free the machine environment memory block.

Note that the `CVBandPrecond` and `CVBandPSolve` functions are never called by the user explicitly; they are simply passed to the `CVSpmgr` function.

#### 4.5.2 A Parallel Band-Block-Diagonal Preconditioner Module

A principal reason for using a parallel ODE solver such as CVODES lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (5) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [12] and is included in a software module within the CVODES package. This module works with the parallel vector module NVECTOR\_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called CVBBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being subdivided into  $M$  non-overlapping subdomains. Each of these subdomains is then assigned to one of the  $M$  processors to be used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate right-hand side function. This requires the definition of a new function  $g(t, y)$  which approximates the function  $f(t, y)$  in the definition of the ODE system (1). However, the user may set  $g = f$ . Corresponding to the domain decomposition, there is a decomposition of the solution vector  $y$  into  $M$  disjoint blocks  $y_m$ , and a decomposition of  $g$  into blocks  $g_m$ . The block  $g_m$  depends on  $y_m$  and also on components of blocks  $y_{m'}$  associated with neighboring subdomains (so-called ghost-cell data). Let  $\bar{y}_m$  denote  $y_m$  augmented with those other components on which  $g_m$  depends. Then we have

$$g(t, y) = [g_1(t, \bar{y}_1), g_2(t, \bar{y}_2), \dots, g_M(t, \bar{y}_M)]^T \quad (24)$$

and each of the blocks  $g_m(t, \bar{y}_m)$  is uncoupled from the others.

The preconditioner associated with this decomposition has the form

$$P = \text{diag}[P_1, P_2, \dots, P_M] \quad (25)$$

where

$$P_m \approx I - \gamma J_m \quad (26)$$

and  $J_m$  is a difference quotient approximation to  $\partial g_m / \partial y_m$ . This matrix is taken to be banded, with upper and lower half-bandwidths `mudq` and `mldq` defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using `mudq` + `mldq` + 2 evaluations of  $g_m$ , but only a matrix of bandwidth `mu` + `ml` + 1 is retained. Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of  $g$ , if smaller values provide a more efficient preconditioner. The solution of the complete linear system

$$Px = b \quad (27)$$



reduces to solving each of the equations

$$P_m x_m = b_m \quad (28)$$

and this is done by banded LU factorization of  $P_m$  followed by a banded backsolve.

To use this CVBBDPRE module, the user must supply two functions which the module calls to construct  $P$ . These are in addition to the user-supplied right-hand side function  $f$ .

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

Both functions take as input the same pointer `f_data` as that passed by the user to `CVodeMalloc` and passed to the user's function `f`, and neither function has a return value. The user is responsible for providing space (presumably within `f_data`) for components of `y` that are communicated by `cfn` from the other processors, and that are then used by `gloc`, which is not expected to do any communication.

The user's calling program should include the following elements:

- `#include "cvsbbdpre.h"` for needed function prototypes and for type `CVBBDData`;
- `#include "nvector_parallel.h"` for the parallel `NVECTOR` module;
- `CVBBDData p_data;`
- `machEnv = M_EnvInitParallel(comm, Nlocal, N, argc, argv);`
- `y = N_VNew(N, machEnv);`
- `cvode_mem = CVodeMalloc(N, f, ...);`
- `p_data = CVBBDAlloc(Nlocal, mudq, mldq, mukeep, mlkeep, dqrely, gloc, cfn, f_data, cvode_mem);`

where `gloc` and `cfn` are names of user-supplied functions; `f_data` is a pointer to private data; and `cvode_malloc` is the pointer to CVODES memory returned by `CVodeMalloc`. The `CVBBDAlloc` call includes half-bandwidths `mudq` and `mldq` to be used in the difference-quotient calculation of the approximate Jacobian. They need not be the true half-bandwidths of the Jacobian of the local block of  $g$ , when smaller values may provide a greater efficiency. Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further. For all four half-bandwidths, the values need not be the same on every processor.

- `ier = CVSpgmr(cvode_mem, pretype, gstype, maxl, delt, CVBBDPrecon, CVBBDPSol, p_data);`

with the pointers `cvode_mem` and `p_data` returned by the two previous calls, the four SPGMR parameters (`pretype`, `gstype`, `maxl`, `delt`) and the names of the preconditioner routines (`CVBBDPrecon`, `CVBBDPSol`) supplied with the CVBBDPRE module;

- `ier = CNode(cnode_mem, tout, y, &t, itask);` to carry out the integration;
- `CVBBDFree(p_data);` to free the CVBBDPRE memory block;
- `CNodeFree(cnode_mem);` to free the CNODES memory block;
- `M_EnvFree_Parallel(machEnv);` to free the machine environment memory block.

Three optional outputs associated with this module are available by way of macros. These are:

- `CVBBD_RPWSIZE(p_data)` the size of the real workspace (local to the current processor) used by CVBBDPRE.
- `CVBBD_IPWSIZE(p_data)` the size of the integer workspace (local to the current processor) used by CVBBDPRE.
- `CVBBD_NGE(p_data)` the cumulative number of  $g$  evaluations (calls to `gloc`) so far.

The costs associated with CVBBDPRE also include `nsetups` LU factorizations, `nsetups` calls to `cfn`, and `nps` banded backsolve calls, where `nsetups` and `nps` are optional CNODES outputs.

Similar block-diagonal preconditioners could be considered with different treatment of the blocks  $P_m$ . For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

## 5 Using CVODES for Forward Sensitivity Analysis

This section describes the use of CVODES to compute solution sensitivities using forward sensitivity analysis. One of our main guiding principles was to design the CVODES user interface for forward sensitivity analysis as an extension of that for IVP integration. Assuming a user main program and user-defined support routines for IVP integration have already been defined, in order to perform forward sensitivity analysis the user only has to insert a few more calls into the main program and (optionally) define an additional routine which computes the right hand side of the sensitivity systems (11). The only departure from this philosophy is due to the `RhsFn` type definition (§4.4). Without changing the definition of this type, the only way to pass values of the problem parameters to the ODE right hand side function is to require the user data structure `f_data` to contain a pointer to the array of real parameters  $p$ .

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in §4.

### 5.1 A Skeleton of the User's Main Program

The following is a skeleton of the user's main program (or calling program) as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.2, most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with CVODES: steps marked with **[P]** correspond to NVECTOR\_PARALLEL, while steps marked with **[S]** correspond to NVECTOR\_SERIAL. Differences between the user main program in §4.2 and the one below start only at step (8).

1. Include relevant header files. No additional header files need be included for forward sensitivity analysis beyond those for IVP solution (§4.2);
2. **[P]** If MPI is needed by the user code, call `MPI_Init(&argc, &argv);`
3. Set problem dimensions (excluding sensitivity equations):  
**[S]** set `N`; **[P]** set `N` and `Nlocal`;
4. Initialize the machine environment block by calling the appropriate NVECTOR routine:  
**[S]** `M_EnvInit_Serial`; **[P]** `M_EnvInit_Parallel`;
5. Set the vector `y0` of initial values;
6. Call `cvode_mem = CVodeMalloc()` to allocate internal memory for CVODES not related to forward sensitivity computations and initializes CVODES;
7. If Newton iteration is chosen, initialize the linear solver module by calling the appropriate initialization routine;
8. Define the sensitivity problem (see §5.2.1 for more details)

- Set **p**, an array of **Np** real parameters upon which the IVP depends (both through its right-hand side and initial conditions). Only parameters with respect to which sensitivities are (potentially) desired need to be included. Also set **pbar**, an array of **Np** scaling factors.
- Attach **p** to the user data structure **f\_data**. For example, **f\_data->p = p;**
- Set **Ns**, the number of parameters with respect to which sensitivities are to be computed. **Ns** must not be larger than **Np**;
- Set **plist**, an array of **Ns** integer flags to specify the parameters **p** with respect to which solution sensitivities are to be computed.

Note that the names for **p**, **pbar**, **plist**, as well as the field *p* of **f\_data** are arbitrary, but they must agree with the arguments to **CCodeSensMalloc** below;

9. Set the **Ns** vectors **yS0[i]** of **N** initial values for sensitivities (for  $i = 0, \dots, Ns - 1$ ). If an existing data array **ySdata** (of type **realtype\*\*** and pointing to **Ns** vectors of length **N** each) contains the initial values **yS0**, then use a macro of type **NVS\_MAKE** defined by the current **NVECTOR** implementation:

- **[S]** **NVS\_MAKE\_S(Ns, yS0, ySdata, machEnv);**
- **[P]** **NVS\_MAKE\_P(Ns, yS0, ySdata, machEnv);**

Otherwise, make the call **yS0 = N\_VNew\_S(Ns,N,machEnv);** to create an array of **N\_Vector**'s and load initial values for sensitivities **yS0[i]** into the array given by:

- **[S]** **NV\_DATA\_S(yS0[i])**
- **[P]** **NV\_DATA\_P(yS0[i])**

10. Call **ier = CCodeSensMalloc(...);** to activate forward sensitivity computations and allocate internal memory for **CVODES** related to sensitivity calculations (see §5.2.1);
11. Call **CCode** for each point at which output is desired. The forward sensitivity equations will be integrated together with the original IVP;
12. After each successful return from **CCode**, the solution of the original IVP is available in the **y** argument of **CCode**, while the sensitivity solution can be extracted into **yS** (which can be the same as **yS0**) by calling the routine **ier = CCodeSensExtract(cvode\_mem, t, yS);** (see §5.2.2);
13. Upon completion of the integration, deallocate memory for the vector **y** and the vectors **yS**. If **yS** was created from **ysdata**, then use the implementation-dependent **NVECTOR** deallocation macro:

- **[S]** **NVS\_DISPOSE\_S(yS, Ns);**
- **[P]** **NVS\_DISPOSE\_P(yS, Ns);**

If **yS** was allocated through a call to **N\_VNew\_S** then deallocate it by calling **N\_VFree\_S(Ns, yS);**

14. Before freeing the pointer to the user-defined data block `f_data`, release the array containing the real parameters `p`: `free(f_data->p); free(f_data);`
15. Free the memory allocated for CVODES by calling `CVodeFree(cvode_mem);`
16. Free the machine environment block by calling the appropriate NVECTOR implementation-dependent routine;
17. [P] If MPI was initialized by the user main program, call `MPI_Finalize();`.

## 5.2 User-Callable Routines for Forward Sensitivity Analysis

### 5.2.1 Forward Sensitivity Initialization Routine

The routine `CVodeSensMalloc` activates forward sensitivity computations and allocates internal memory related to sensitivity calculations. The form of the call to this routine is

```
ier = CVodeSensMalloc(cvode_mem, Ns, ism, p, pbar, plist,
                    ifS, fS, errcon, rhomax, yS0, rtolS, atolS);
```

- `cvode_mem` is the pointer to the CVODES memory returned by `CVodeMalloc`;
- `Ns` is the number of sensitivities to be computed;
- `ism` is a flag used to select the sensitivity solution method and can be `SIMULTANEOUS`, `STAGGERED`, or `STAGGERED1`:
  - In the `SIMULTANEOUS` approach, the state and sensitivity variables are corrected at the same time. If `NEWTON` was selected as the nonlinear system solution method, this amounts to performing a modified Newton iteration on the combined nonlinear system;
  - In the `STAGGERED` approach, the correction step for the sensitivity variables takes place at the same time for all sensitivity equations, but only after the correction of the state variables has converged and the state variables have passed the local error test;
  - In the `STAGGERED1` approach, all corrections are done sequentially, first for the state variables and then for the sensitivity variables, one parameter at a time. If the sensitivity variables are not included in the error control, this approach is equivalent to `STAGGERED`. Note that the `STAGGERED1` approach can be used only if `ifS = ONESENS`.
- `p` is a pointer to the array of real problem parameters used to evaluate  $f(t, y, p)$ . The user data block `f_data` must include a realtype pointer (e.g. `p`) that points to `p`. For example, if the pointer to the data block has the form `typedef struct{..., realtype *p;}*f_data;` then `f_data->p = p;` must point to the array in which `p[i-1] = pi`, for  $i = 1, \dots, N_p$ ;
- `pbar` is an array of real values that are used to scale the sensitivity absolute error tolerance vectors. Each `pbar[i]` must be set to a nonzero constant that is dimensionally consistent with `p[i]`. Typically, `pbar[i]=p[i]` whenever `p[i]` is nonzero;

- **plist** is an array of **Ns** nonzero integer flags that serves two functions: it specifies parameter indices in  $\{1, \dots, N_p\}$  with respect to which sensitivities are to be computed and indicates whether a given parameter affects the right hand side of the original IVP or only its initial conditions. More specifically, a positive  $j = \text{plist}[i]$  indicates that the sensitivity of the solution with respect to the  $j$ -th parameter  $p[j-1]$  is to be computed. A negative  $j = \text{plist}[i]$  indicates that the sensitivity of the solution with respect to the  $(-j)$ -th parameter  $p[-j-1]$  is to be computed and indicates that  $p[-j-1]$  does not enter  $f(t, y, p)$  thus increasing the efficiency of the difference quotient approximation routine for sensitivity right hand side evaluation;
- **ifS** is the type of sensitivity right hand side. The legal values are **ALLSENS** or **ONESENS**. The **ALLSENS** type means that right hand sides for all sensitivities are provided simultaneously. In this case **fS** (if provided by the user) must be of type **SensRhsFn**. The **ONESENS** type means that **fS** (which, if provided by the user, must be of type **SensRhs1Fn**) computes one sensitivity right hand side at a time. Note that **ism = STAGGERED1** requires **ifS = ONESENS**. Either value for **ifS** is valid for **ism = SIMULTANEOUS** or **ism = STAGGERED**;
- **fS**, if not **NULL**, is a user-provided C function to evaluate the right-hand sides of the sensitivity equations (11). If a **NULL** pointer is passed then **CVODES** uses one the default difference quotient routines (**CVSensRhsDQ** or **CVSensRhs1DQ**, depending on the value **ifS**) to evaluate these quantities. For more details, see §5.3;
- **errcon** is a flag used to specify whether partial or full error control is to be used. If **errcon = FULL** then both state variables and sensitivity variables are included in the error tests. If **errcon = PARTIAL** then the sensitivity variables are excluded from the error tests. Note that, in any event, all variables are considered in the convergence tests;
- **rhomax** is a real scalar used to decide the finite differencing strategy in the case in which residuals of sensitivity equations are to be computed by **CVODES** (**fS = NULL**) using the internal difference quotient routines (see §2.2 for details);
- **yS0** is a pointer to an array of **Ns** vectors of length **N** containing the initial values of the sensitivities;
- **rtolS** is a pointer to the user's relative error tolerance for sensitivity variables. If a **NULL** pointer is passed for **rtolS** then **CVODES** uses the same relative tolerance for sensitivity variables as for the state variables;
- **atolS** must point to a vector of **Ns** absolute tolerance values for sensitivity variables (if **itol = SS**) or to an array of **Ns** vectors of sensitivity absolute tolerances (if **itol = SV**). If a **NULL** pointer is passed for **atolS**, **CVODES** defaults to using as absolute error tolerance for sensitivity **yS[i]** a multiple of the absolute error tolerance for the state variables, with the scale factor being  $1/\text{pbar}[j-1]$ ,  $j = |\text{plist}[i]|$ .

Note that, unlike the argument **y0** of **CVodeMalloc** - which could be deallocated by the user before the call to **CVode**, the space **yS0** is used during the integration of the sensitivity equations and should therefore not be deallocated until after the last call to **CVode**.

The return value **ier** of **CVodeSensMalloc** is equal to:

- SUCCESS=0 if there were no errors;
- SCVM\_NO\_MEM if `cvmem` was NULL;
- SCVM\_ILL\_INPUT if an input argument was illegal;
- SCVM\_MEM\_FAIL if a memory request failed.

### 5.2.2 Forward Sensitivity Extraction Routine

If forward sensitivity computations have been initialized by a call to `CVodeSensMalloc`, or reinitialized by a call to `CVSensReInit`, then CVODES computes both solution and sensitivities at time `t`. However, `CVode` will still return only the solution  $y$  in `y`. Solution sensitivities can be obtained through the routine `CVodeSensExtract`:

```
ier = CVodeSensExtract(cvmem, t, yS);
```

Its arguments are as follows:

- `cvmem` is the pointer to the memory previously allocated by `CVodeMalloc`.
- `t` specifies the time at which sensitivity information is requested. The time `t` must fall within the interval defined by the last successful step taken by CVODES.
- `yS` must be declared of type `N_Vector_S` (i.e. a pointer to `N_Vector`). The user can use `yS = yS0`. If successful, `CVodeSensExtract` will load `yS` with the values of the solution sensitivities at time `t`. Sensitivity with respect to the  $i$ -th sensitivity parameter (i.e. parameter `p[plist[i]-1]`) can be accessed in the `N_Vector` `yS[i]`.

The return value `ier` of `CVodeSensExtract` is equal to:

- SUCCESS=0 if there were no errors;
- DKY\_NO\_MEM if `cvmem` was NULL;
- DKY\_NO\_SENSI if sensitivity computation was not turned on by a call to `CVodeSensMalloc`;
- BAD\_T if the time `t` is not in the allowed range.

In case of an error return, an error message is also printed.

### 5.2.3 Additional Optional Input/Output

As mentioned in §4.3.4, during forward sensitivity analysis, CVODES stores some additional integer output values in `iopt`. These values are described in Table 4.

Table 4: Additional optional integer output entries in the array `iopt` from forward sensitivity analysis

Index	Description
NFSE	Number of calls made to the sensitivity right hand side evaluation routine.
NNIS	Number of Newton iterations performed during sensitivity corrections (sum over all sensitivities in the <b>STAGGERED1</b> case).
NCFNS	Number of nonlinear convergence failures during the sensitivity corrections (sum over all sensitivities in the <b>STAGGERED1</b> case).
NETFS	Number of error test failures for sensitivity variables.

#### 5.2.4 Additional Diagnostics Extraction Routine

In the **STAGGERED1** approach, the correction and error test phases are done sequentially for each sensitivity parameter. In this case, `CVODES` collects additional information regarding the performance of the nonlinear solvers for the each individual sensitivity system, in particular the number of nonlinear iterations and nonlinear solver convergence failures for each sensitivity system. Upon return from `CVode`, the user can obtain this information with a call to the function `CVodeMemExtract`.

The call to `CVodeMemExtract` has the following form:

```
ier = CVodememExtract(cvode_mem, n_niS1, n_cfnS1);
```

The vectors `n_niS1` and `n_cfnS1`, each of length `Ns` and type `long int *` must be allocated by the user if the corresponding information is desired. Upon a successful return, the return value is `OKAY`, and `n_niS1`, if non-NULL, contains the nonlinear iteration counts, while `n_cfnS1`, if non-NULL, contains the nonlinear solver convergence failure counts. If `cvode_mem` is NULL, `CVodememExtract` returns `MEXT_NO_MEM`.

#### 5.2.5 Interpolated Sensitivity Output Routines

The two routines described here are available to obtain additional output values for the sensitivity variables.

The routine `CVodeSensDkyAll` computes the  $k$ -th derivatives of the interpolating polynomials for each sensitivity variable at time `t`. This function is called by `CVodeSensExtract` with `k = 0`, but may also be called directly by the user.

```
ier = CVodeSensDkyAll(cvode_mem, t, k, dkyA)
```

The arguments `cvode_mem` and `t` are as above. The argument `k` specifies the derivative order and must be  $0 \leq k \leq q$ , where  $q$  is the order of the LMM used on the last step. If its value is illegal, the return value is `BAD_K`. The argument `dkyA` must be declared as a pointer to `N_Vector` (i.e. of type `N_Vector_S`) and the user must allocate space for it. If `dkyA` or any of its component vectors is NULL, the return value is `BAD_DKY`.



The routine `CVodeSensDky` computes the  $k$ -th derivatives of the interpolating polynomial for the  $is$ -th sensitivity variable at time  $t$ . This function is called by `CVodeSensDkyAll` for all sensitivities, but may also be called directly by the user.

```
ier = CVodeSensDky(cvode_mem, t, k, is, dky)
```

The arguments `cvode_mem`,  $t$ , and  $k$  are as above. The argument `is` specifies the sensitivity for which information is requested and must be  $0 \leq is < Ns$ . If its value is illegal, the return value is `BAD_IS`. `dkyA` must be declared as an `N_Vector` and the user must allocate space for it. If `dkyA` is `NULL`, the return value is `BAD_DKY`.

### 5.2.6 Forward Sensitivity Reinitialization Routine

The routine `CVSensReInit`, useful during the solution of a sequence of problems of same size, reinitializes the sensitivity related internal memory and must follow a call to `CVodeSensMalloc` (and maybe a call to `CVReInit`. The number  $Ns$  of sensitivities is assumed to be unchanged since the call to `CVodeSensMalloc`.

The call to the `CVSensReInit` function has the form

```
ier = CVSensReInit(cvode_mem, ism, p, pbar, plist,
                  ifS, fS, errcon, rhomax, yS0, rtolS, atolS);
```

The arguments have names and meanings identical to those of `CVodeSensMalloc`. Note that the number of sensitivities  $Ns$  is not passed as an argument to `CVSensReInit`, as that is assumed to be unchanged since the `CVodeSensMalloc` call.

The return value `ier` of `CVSensReInit` is equal to:

- `SUCCESS=0` if there were no errors;
- `SCVREI_NO_MEM` if `cvode_mem` was `NULL`;
- `SCVREI_NO_SEN` if sensitivity computation was not turned on by a call to `CVodeSensMalloc`;
- `SCVREI_ILL_INPUT` if an input argument was illegal;
- `SCVREI_MEM_FAIL` if a memory request failed.

In case of an error return, an error message is also printed.

Note that `CVSensReInit` is not a completely genuine reinitialization routine as it may perform some memory allocation. This can happen if `atolS = NULL` is passed to `CVSensReInit`, in which case `CVODES` must allocate and set its own internal absolute tolerances for the sensitivity variables, or if `ism = STAGGERED1`, in which case some internal counter arrays of length  $Ns$  are allocated by `CVSensReInit`.

## 5.3 User-Supplied Routines for Forward Sensitivity Analysis

In addition to the required and optional user-supplied routines described in §4.4, when using `CVODES` for forward sensitivity analysis, the user has the option of providing a routine that calculates the right hand side of the sensitivity equations (11).

CVODES provides difference quotient approximation routines for the right hand sides of the sensitivity equations, `CVSensRhsDQ` if `ifS = ALLSENS` and `CVSensRhs1DQ` if `ifS = ONESENS`. To use these routines, the user must pass `fS = NULL` in the call to `CVodeSensMalloc` (see §5.2.1). However, CVODES allows the option for user-defined sensitivity right hand side routines (which also provides a mechanism for interfacing CVODES to routines generated by automatic differentiation).

Recall that, if sensitivity analysis is to be performed, the user-supplied data structure `f_data` contains a pointer (e.g., `p`) that points to the array of real parameters upon which the original IVP depends.

- *Sensitivity equations right hand side (the case ALLSENS)*

If the `SIMULTANEOUS` or `STAGGERED` approach was selected in the call to `CVodeSensMalloc`, the user may provide the right hand sides of the sensitivity equations (11), for all sensitivity parameters at once, through a function of type `SensRhsFn` defined by

```
typedef void (*SensRhsFn) (RhsFn f, integertype Ns, integertype N,
                           realtype t, N_Vector y, N_Vector ydot,
                           N_Vector *yS, N_Vector *ySdot,
                           realtype *p, realtype *pbar, integertype *plist,
                           void *f_data, N_Vector ewt, N_Vector *ewtS,
                           realtype *reltol, realtype *reltolS,
                           realtype uround, realtype rhomax, long int *nfePtr,
                           N_Vector ytemp, N_Vector ftemp);
```

In this case, the argument `ifS` of `CVodeSensMalloc` must be set to `ALLSENS`. Note that a sensitivity right hand side function of type `SensRhsFn` is not compatible with the `STAGGERED1` approach.

A function of type `SensRhsFn` receives as input the value of the independent variable `t`, the ODE solution vector `y` and its derivative `ydot`, and sensitivity vectors `yS`. It must compute the vectors  $(\partial f / \partial y) s_i(t) + (\partial f / \partial p_i)$  and store them in `ySdot[i]`. There is no return value for a `SensRhsFn`.

The complete list of arguments is given below. Typically, a user-supplied sensitivity right hand side function would be expected to access the arguments `Ns`, `N`, `t`, `y`, `yS`, `p`, `plist`, and `f_data`. The remaining arguments would not typically be accessed, but appear in the call list because they are needed by the difference quotient function `CVSensRhsDQ`.

- `f` is a pointer to the user-supplied routine of type `RhsFn` passed to `CVodeMalloc`;
- `Ns` is the number of sensitivity parameters;
- `N` is the problem dimension;
- `t` is the value of the independent variable;
- `y` and `ydot` are the ODE solution and its derivative;
- `yS` contains the `Ns` sensitivity vectors;
- `ySdot` is the output of `SensRhsFn`. On exit it must contain the sensitivity right hand side vectors;

- `p` is the vector of problem parameters;
  - `pbar` is the vector containing scaling factors for `p`;
  - `plist` is a vector of flags specifying the sensitivity parameters among the problem parameters;
  - `f_data` is a pointer to the user-data space passed to `CVodeMalloc`;
  - `ewt` contains error weights for the state variables `y`;
  - `ewtS` is an array of vectors `ewtS[i]` containing error weights for sensitivity variables `yS[i]`;
  - `reltol` and `reltolS` are the relative error tolerances for state and sensitivity variables, respectively;
  - `uround` is the machine unit roundoff;
  - `rhomax` is the finite difference threshold parameter used by `CVSensRhsDQ`;
  - `nfePtr` is a pointer to the number of function evaluation calls counter. If any calls to `f` are made, the user has the option of accounting for them in the final CVODES statistics;
  - `ytemp` and `ftemp` are pointers to memory allocated for vectors of length `N` which can be used by a `SensRhsFn` function as temporary storage or work space.
- Sensitivity equations right hand side (the case `ONESENS`)

Alternatively, the user may provide the sensitivity right hand sides, one sensitivity parameter at a time through a function of type `SensRhs1Fn`. In this case, the argument `ifS` of `CVodeSensMalloc` must be set to `ONESENS`. Note that a sensitivity right hand side function of type `SensRhs1Fn` is compatible with any legal value of the `CVodeSensMalloc` argument `ism`, and is required if `ism = STAGGERED1`.

The type `SensRhs1Fn` is defined by

```
typedef void (*SensRhs1Fn) (RhsFn f, integertype Ns, integertype N,
                           realtype t, N_Vector y, N_Vector ydot,
                           integertype iS, N_Vector yS, N_Vector ySdot,
                           realtype *p, realtype *pbar, integertype *plist,
                           void *f_data, N_Vector ewt, N_Vector *ewtS,
                           realtype *reltol, realtype *reltolS,
                           realtype uround, realtype rhomax, long int *nfePtr,
                           N_Vector ytemp, N_Vector ftemp);
```

Except for `iS`, `yS`, and `ySdot`, the arguments are identical to those of `SensRhsFn` functions. A function of type `SensRhs1Fn` receives as an argument `iS` specifying the sensitivity parameter for which the sensitivity right hand side vector must be evaluated ( $0 \leq iS < Ns$ ). The argument `yS` contains the `iS`-th sensitivity vector and, on return, `ySdot` must contain the right hand side of the `iS`-th sensitivity system.

Typically, a user-supplied sensitivity right hand side function would be expected to access the arguments `iS`, `Ns`, `N`, `t`, `y`, `yS`, `p`, `plist`, and `f_data`. The remaining arguments would not typically be accessed, but appear in the call list because they are needed by the difference quotient function `CVSensRhs1DQ`.

## 6 Using CVODES for Adjoint Sensitivity Analysis

This section describes the use of CVODES to compute sensitivities of derived functions using adjoint sensitivity analysis. As mentioned before, the adjoint sensitivity module of CVODES provides the infrastructure for integrating backwards in time any system of ODEs that depends on the solution of the original IVP, by providing various interfaces to the main CVODES integrator, as well as several supporting user-callable routines. For this reason, in the following sections we refer to the *backward problem* and not to the *adjoint problem* when discussing details relevant to the ODEs that are integrated backwards in time. The backward problem can be the adjoint problem (20) or (23), maybe augmented with some quadrature differential equations.

We begin with a brief overview, in the form of a skeleton user program. Following that are detailed descriptions of the interface to the various user-callable routines and of the user-supplied routines that were not already described in §4.

### 6.1 A Skeleton of the User's Main Program

The following is a skeleton of the user's main program as an application of CVODES. The user program is to have these steps in the order indicated, unless otherwise noted. For the sake of brevity, we defer many of the details to the later sections. As in §4.2, most steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the two implementations provided with CVODES: steps marked with **[P]** correspond to NVECTOR\_PARALLEL, while steps marked with **[S]** correspond to NVECTOR\_SERIAL.

1. Include necessary header files. The header file that is always required is `cvodea.h` which defines additional types and constants, and includes function prototypes for the adjoint sensitivity module user-callable routines. The header file `cvodes.h` need not be included by the user, as it is included by `cvodea.h`. In addition, the main program should include an NVECTOR implementation header file (`nvector_serial.h` or `nvector_parallel.h` for the two implementations provided with CVODES) and, if Newton iteration was selected, a header file from the desired linear solver module.
2. **[P]** If MPI is needed by the user code, call `MPI_Init(&argc, &argv);`
3. Set problem dimensions for the forward problem:  
**[S]** set `N`; **[P]** set `N` and `Nlocal`;
4. Initialize the machine environment block for the forward problem by calling the appropriate implementation-dependent NVECTOR routine:  
**[S]** `M_EnvInit_Serial`; **[P]** `M_EnvInit_Parallel`;
5. Set the vector `y0` of initial values for the forward problem;
6. Call `cvode_mem = CVodeMalloc()` to allocate internal memory for CVODES and initializes CVODES for the forward problem;
7. If Newton iteration is chosen, initialize the linear solver module for the forward problem, by calling the appropriate initialization routine;

8. Call `cvadj_mem = CVadjMalloc()` to allocate memory for the combined forward-backward problem (see §6.2.1 for more details). This call requires `Nd`, the number of steps between two consecutive check points;
9. Call `CVodeF`, a wrapper around the CVODES main integration routine `CVode`, either in `NORMAL` mode to the time `tout` or in `ONE_STEP` mode inside a loop (if intermediate solutions of the forward problem are desired (see §6.2.2). The final value of `tout` (or of `t`), denoted `tlast`, is then the endpoint  $t_1$ ;
10. Set problem dimensions for the backward problem:  
`[S]` set `NB`, the number of variables in the backward problem; `[P]` set `NB` and `NBlocal`;
11. Initialize the machine environment block for the backward problem by calling the appropriate implementation-dependent `NVECTOR` routine (typically the same as for the forward problem):  
`[S]` `M_EnvInit_Serial`; `[P]` `M_EnvInit_Parallel`;
12. Set the vector `yB0` of final values for the backward problem;
13. Call `CVodeMallocB`, a wrapper around the `CVodeMalloc`, to allocate internal memory and initialize CVODES for the backward problem (see §6.2.3);
14. If Newton iteration is chosen, initialize the linear solver module for the backward problem, by calling the appropriate wrapper around the initialization routines `CVDenseB`, `CVBandB`, `CVDiagB`, or `CVSpqrB` (see §6.2.4). Note that it is not required to use the same linear solver module for both the forward and the backward problems; for example, the forward problem could be solved with the `CVDENSE` linear solver and the backward problem with `CVSPGMR`;
15. Call `CVodeB`, a second wrapper around the CVODES main integration routine `CVode`, to integrate the backward problem from `tlast` to `t0`, returning the solution of the backward problem at `t0` into `yB` (see §6.2.5);
16. Upon completion of the backward integration, call all necessary deallocation routines. These include calls to `NV_Free` for the vectors `y` and `yB`, calls to the appropriate implementation-dependent `NVECTOR` free routine for the forward and backward machine environment blocks, and calls to `CVodeFree` to free the CVODES memory block for the forward problem and to `CVadjFree` (see §6.2.6) to free the memory allocated for the combined problem. Note that `CVadjFree` also deallocates the CVODES memory for the backward problem;
17. `[P]` If MPI was initialized by the user main program, call `MPI_Finalize()`;

The above user interface to the adjoint sensitivity module in CVODES was motivated by the desire to keep it as close as possible in look and feel to the one for ODE IVP integration. Note that if steps (10)-(15) are not present, a program with the above structure will have the same functionality as one described in §4.2 for integration of ODEs, albeit with some overhead due to the check pointing scheme.

## 6.2 User-Callable Routines for Adjoint Sensitivity Analysis

### 6.2.1 Adjoint Sensitivity Allocation Routine

The routine `CVadjMalloc` allocates internal memory for the combined forward-backward integration, other than the `CVODES` memory block. Space is allocated for the  $N_d$  interpolation data points and a linked list of check points is initialized. The form of the call to this routine is

```
cadj_mem = CVadjMalloc(cvode_mem, Nd);
```

where `cvode_mem` is the `CVODES` memory block for the forward problem returned by a previous call to `CVodeMalloc` and `Nd` is the number of integration steps between two consecutive check points. If successful, `CVadjMalloc` returns a pointer of type `void *`. The user doesn't need to access this memory block but must pass it to other adjoint module user-callable routines. In case of failure (`cvode_mem` is `NULL`, `Nd` is non-positive, or a memory request fails), `CVadjMalloc` prints an error message to the standard output stream `stdout` and returns `NULL`.

The user must set `Nd` so that all data needed for interpolation of the forward problem solution between two check points fits in memory. `CVadjMalloc` attempts to allocate space for  $(2Nd+3)$  vectors of length  $N$ , the dimension of the forward problem.

### 6.2.2 Forward Integration Routine

The routine `CVodeF` is very similar to the `CVODES` routine `CVode` (see §4.3.3) in that it integrates the solution of the forward problem and returns the solution in `y`. At the same time, however, `CVodeF` stores check point data every `Nd` integration steps. `CVodeF` can be called repeatedly by the user. The last value of `tout` (or of `t`), `tlast`, will be used as the starting time for the backward integration.

The call to this routine has the form

```
ier = CVodeF(cadj_mem, tout, y, &t, itask, &ncheckPtr);
```

Most of its arguments have names and meanings identical to those of `CVode`. In addition, `CVodeF` takes as first argument the memory pointer `cvad_mem` returned by `CVadjMalloc` and returns in its last argument the final number of check points. If an error occurred during the memory allocation of a new check point, `CVodeF` returns `CVODEF_MEM_FAIL` and prints an error message to `stdout`. Otherwise, since `CVodeF` wraps around `CVode`, its return value `ier` is the return value of `CVode` (see §4.3.3): `SUCCESS=0` or `TSTOP_RETURN=1` for a successful return, or a negative value in case of failure.

Note that, at this time, `CVodeF` stores check point information into memory only. Future versions will provide for a safe-guard option of dumping check point data to a temporary file as needed. The data stored at each check point is basically a snapshot of the `CVODES` internal memory block and contains enough information to restart the integration from that time and proceed with the same stepsize and method order sequence as during the forward integration.

### 6.2.3 Backward Problem Initialization Routine

The routine `CVodeMallocB` initializes and allocates memory for the backward problem. It has the following form

```
ier = CVMallocB(cvadj_mem, NB, fB, yB0, lmmB, iterB, itolB, &rtolB,
               atolB, f_dataB, errfpB, optInB, ioptB, roptB, machEnvB);
```

and is essentially a call to `CVMalloc` with some particularization for backward integration. First, `CVMallocB` takes as an argument `cvadj_mem`, the memory pointer returned by `CVadjMalloc`. Secondly, no integration starting time is required, as the backward integration starts with the `tlast` value from the last `CVodeF` call. Finally, the argument `fB`, the C function to compute the right hand side of the backward problem, must be of type `RhsFnB` and has the form `fB(NB, t, y, yB, yBdot, f_dataB)` (see §6.3 for details). All other arguments are equivalent to those of `CVMalloc`. The relative and absolute tolerances `rtolB` and `atolB` for `yB` need to be set appropriately; this may require some experimentation. Data that could help in the selection of appropriate tolerances includes the time scale of the forward problem (including the actual initial stepsize used, available in the optional I/O real array at `ropt[HOU]`), the `yB` values at  $t_0$  after the backward integration phase, and (if an adjoint problem is integrated backwards in time) the value of the gradient  $g_y$  at time  $t_1$ .

The return value of `CVMallocB` is `SUCCESS` if there were no errors, `CVBM_NO_MEM` if `cvadj_mem` was `NULL`, or `CVBM_MEM_FAIL` if a memory request failed.

#### 6.2.4 Linear Solver Initialization Routines for Backward Problem

The adjoint sensitivity module in `CVODES` provides interfaces to the initialization routines of all supported linear solver modules for the case in which Newton iteration is selected for the solution of the backward problem. The initialization routines described in §4.3.2 cannot be directly used since the optional user-defined Jacobian-related routines have different prototypes for the backward problem than for the forward problem. The initialization routine `CVDiag` for the diagonal linear solver can be used directly, since `CVDIAG` does not provide for a user-defined diagonal Jacobian approximation routine.

- *Dense linear solver initialization*

In order to use the `CVDENSE` solver for the backward problem, after the call to `CVMallocB`, the calling program must make the call

```
ier = CVDenseB(cvadj_mem, djacB, jac_dataB);
```

The argument `cvadj_mem` is a pointer to the memory block returned by `CVadjMalloc`. The `CVDENSE` solver routine for computing a dense approximation to the Jacobian matrix of the backward problem must be of type `CVDenseJacFnB`, and is communicated through the argument `djacB`. The user can supply his/her own dense Jacobian routine (see §6.3), or use the difference quotient routine `CVDenseDQJac` that comes with the `CVDENSE` solver. To use `CVDenseDQJac`, the user must pass `NULL` for the `djacB` parameter. As with `CVDense`, `jac_dataB` is a pointer to a user-defined data structure that the `CVDENSE` solver passes to its Jacobian function and which can be used to store data relevant to the Jacobian computation.

The possible return values `ier` are identical to those of `CVDense` (see §4.3.2).

- *Banded linear solver initialization*

In order to use the `CVBAND` solver for the backward problem, after the call to `CVMallocB`, the calling program must make the call

```
ier = CVBandB(cvadj_mem, mupperB, mlowerB, bjacB, jac_dataB);
```

The CVBandB argument `cvadj_mem` is a pointer to the memory block returned by `CVadjMalloc`. The upper and lower bandwidths of the Jacobian of the backward problem (or its approximation) are specified in this call through the `mupperB` and `mlowerB` parameters. The CVBAND solver routine for computing a banded approximation to the Jacobian matrix of the backward problem must be of type `CVBandJacFnB`, and is communicated through the argument `bjacB`. The user can supply his/her own banded Jacobian routine (see §6.3), or use the difference quotient routine `CVBandDQJac` that comes with the CVBAND solver. To use `CVBandDQJac`, the user must pass NULL for the `bjacB` parameter. As before, `jac_dataB` is a pointer to a user-defined data structure that the CVBAND solver passes to its Jacobian function and which can be used to store data relevant to the Jacobian computation.

The possible return values `ier` are identical to those of `CVBand` (see §4.3.2).

- *SPGMR linear solver initialization*

The CVSPGMR solver can be linked to CVODES for use during the backward problem solution by calling the routine `CVSpgrmrB`. The call to this routine has the following form:

```
ier = CVSpgrmrB(cvadj_mem, pretypeB, gstypeB, maxlB, deltb, PrecondB,
                PSolveB, P_dataB, jtimesB, jac_dataB);
```

The first argument, `cvadj_mem`, is a pointer to the memory block returned by `CVadjMalloc`. The rest of the arguments are equivalent to those of `CVSpgrmr` (see §4.3.2) with the following exceptions:

- `PrecondB`, the optional user-defined preconditioner setup routine, must now have function type `CVSpgrmrPrecondFnB`;
- `PSolveB`, the user-defined preconditioner solve routine, must now have function type `CVSpgrmrPSolveFnB`;
- `jtimesB`, the optional user-defined Jacobian times vector routine, must now be of type `CVSpgrmrJtimesFnB`. The user can supply his/her own Jacobian times vector approximation routine, or use the difference quotient routine `CVSpgrmrDQJtimes`. To use the `CVSpgrmrDQJtimes`, the user must pass NULL for `jtimesB`.

For details on the prototypes of these user-defined functions, see §6.3. The possible return values `ier` are identical to those of `CVSpgrmr` (see §4.3.2).

## 6.2.5 Backward Integration Routine

The routine `CVodeB` performs the integration of the backward problem from the final to the initial time. It is essentially a wrapper around the CVODES main integration routine `CVode` and, in the case in which check points were needed, it evolves the solution of the backward problem through a sequence of forward-backward integrations between consecutive check points. The first run integrates the original IVP forward in time and stores interpolation data; the second run integrates the backward problem backward in time and performs Hermite interpolation to provide the solution of the IVP to the backward problem.

The call to this routine has the form



```
ier = CVodeB(cvadj_mem, yB);
```

and loads the solution of the backward problem at the initial time into `yB`.

If `CVodeB` successfully integrates the backward problem, the return value is `TSTOP_RETURN=1`, as the `TSTOP` option is automatically turned on in this phase. If an error occurred, `CVodeB` prints an error message indicating whether the error occurred during the forward or the backward integration phase and returns `ier` from `CVode` (see §4.3.3).

### 6.2.6 Adjoint Sensitivity Deallocation Routine

To free the `cvadj_mem` memory block allocated by `CVadjMalloc`, the user must call `CVadjFree`. The call to this routine has the form

```
CVadjFree(cvadj_mem);
```

and it frees check-point-related memory, as well as the `CVODES` memory block allocated for the integration of the backward problem. There is no return value for `CVadjFree`.

### 6.2.7 Check Point Listing Routine

For debugging purposes, `CVODES` provides a routine `CVadjCheckPointsList` which displays partial information from the linked list of check points generated by `CVodeF`. The call to this routine has the form:

```
CVadjCheckPointsList(cvadj_mem);
```

For a typical output of `CVadjCheckPointsList`, see the example section §9.1.

## 6.3 User-Supplied Routines for Adjoint Sensitivity Analysis

In addition to the required ODE right hand side routine and any optional routines for the forward problem, when using the adjoint sensitivity module in `CVODES`, the user must supply one function defining the backward problem ODE and, optionally, routines to supply Jacobian related information (if Newton iteration is chosen) and one or two functions that define the preconditioner (if the `CVSPGMR` solver is selected) for the backward problem.

Type definitions for all these user-supplied routines are given below.

- *ODE right hand side for the backward problem*

The user must provide a function of type `RhsFnB` defined by

```
typedef void (*RhsFnB)(integertype NB, realtype t, N_Vector y,
                      N_Vector yB, N_Vector yBdot, void *f_dataB);
```

to compute the right hand side of the backward problem ODE system. This could be (20) or (23), possibly with one or more quadrature ODEs appended.

This function takes as input the size `NB` of the backward problem, the independent variable value `t`, and the dependent variable vector `yB`, as well as the solution of the original IVP `y` at

time `t`. It must store the backward problem right hand side in the vector `yBdot`. Allocation of memory for `yBdot` is handled within `CVODES`.

The `y`, `yB`, and `yBdot` arguments are all of type `N_Vector`, but `yB` and `yBdot` typically have different internal representations from `y`. It is the user's responsibility to access the vector data consistently (including the use of the correct accessor macros from each `NVECTOR` implementation). For the sake of computational efficiency, the vector kernels in the two `NVECTOR` implementations provided with `CVODES` do not perform any consistency checking for their `N_Vector` arguments (see §11.1 and §11.2).

The `f_dataB` parameter is the same as the `f_dataB` parameter passed by the user to the `CVodeMallocB` routine. This user-supplied pointer is passed to the user's `fB` function every time it is called and can be the same as the `f_data` pointer used for the forward problem. A `RhsFnB` function type does not have a return value.

- *Jacobian information for the backward problem (direct method with dense Jacobian)*

If the direct linear solver with dense treatment of the Jacobian is selected for the backward problem (i.e. `CVDenseB` is called in step 14 of §6.1), the user may provide a function of type `CVDenseJacFnB` defined by

```
typedef void (*CVDenseJacFnB)(integertype NB, DenseMat JB, RhsFnB fB,
                             void *f_dataB, realtype t, N_Vector y,
                             N_Vector yB, N_Vector fyB, N_Vector ewtB,
                             realtype hB, realtype uaroundB, void *jac_dataB,
                             long int *nfePtrB, N_Vector vtemp1B,
                             N_Vector vtemp2B, N_Vector vtemp3B);
```

to compute the dense Jacobian of the backward problem (or an approximation to it). If the backward problem is the adjoint of the original IVP then this Jacobian is nothing else than the transpose of  $J = \partial f / \partial y$  with a change in sign.

A user-supplied dense Jacobian routine must load the `NB` by `NB` dense matrix `JB` with an approximation to the Jacobian matrix at the point  $(t, y, yB)$ , where `y` is the solution of the original IVP at time `t` and `yB` is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into `JB` as this matrix is set to zero before the call to the Jacobian routine. The type of `JB` is `DenseMat`. The user is referred to §4.4 (page 34) for details regarding accessing a `DenseMat` object as well as details on the rest of the arguments of a function of type `CVDenseJacFnB`.

Typically, a user-supplied Jacobian function `djacB` would be expected to access the arguments `NB`, `t`, `y`, `yB`, `JB`, `f_dataB`, and `jac_dataB`, at most. The remaining arguments would not typically be accessed, but appear in the call list because they are needed by the function `CVDenseDQJac` that computes a difference quotient dense Jacobian approximation, when the user specifies that option.

- *Jacobian information for the backward problem (direct method with banded Jacobian)*

If the direct linear solver with banded treatment of the Jacobian is selected for the backward problem (i.e. `CVBandB` is called in step 14 of §6.1), the user may provide a function of type `CVBandJacFnB` defined by

```
typedef void (*CVBandJacFnB)(integertype NB, integertype mupperB,
                             integertype mlowerB, BandMat JB, RhsFnB fB,
                             void *f_dataB, realtype t, N_Vector y,
                             N_Vector yB, N_Vector fyB, N_Vector ewtB,
                             realtype hB, realtype uaroundB, void *jac_dataB,
                             long int *nfePtrB, N_Vector vtemp1B,
                             N_Vector vtemp2B, N_Vector vtemp3B);
```

to compute the banded Jacobian of the backward problem (or an approximation to it).

A user-supplied band Jacobian routine must load the band matrix JB of type BandMat with the elements of the Jacobian at the point  $(t, y, yB)$ , where  $y$  is the solution of the original IVP at time  $t$  and  $yB$  is the solution of the backward problem at the same time. Only nonzero elements need to be loaded into JB because JB is preset to zero before the call to the Jacobian routine. More details on the accessor macros provided for a BandMat object and on the rest of the arguments of a function of type CVBandJacFnB are given in §4.4 on page 35.

Typically, a user-supplied Jacobian function `bjacB` would be expected to access the arguments NB, `mupperB`, `mlowerB`,  $t$ ,  $y$ ,  $yB$ , JB, `f_dataB`, and `jac_dataB`, at most. The remaining arguments would not typically be accessed, but appear in the call list because they are needed by the function `CVBandDQJac` that computes a difference quotient banded Jacobian approximation, when the user specifies that option.

- *Jacobian information for the backward problem (SPGMR case)*

If an iterative SPGMR linear solver is selected (`CVSpgmrB` is called in step 14 of §6.1) the user may provide a function of type `CVSpgmrJtimesFnB` in the form

```
typedef int (*CVSpgmrJtimesFnB)(integertype NB, N_Vector vB, N_Vector JvB,
                                RhsFnB fB, void *f_dataB, realtype t,
                                N_Vector y, N_Vector yB, N_Vector fyB,
                                realtype vnrmB, N_Vector ewtB, realtype hB,
                                realtype uaroundB, void *jac_dataB,
                                long int *nfePtrB, N_Vector workB);
```

to compute the action of the Jacobian on a given vector  $vB$  for the backward problem (or an approximation to it). A user-supplied Jacobian times vector routine must load the vector  $JvB$  with the result of the product between the Jacobian of the backward problem at the point  $(t, y, yB)$  and the vector  $vB$  of dimension NB. Here,  $y$  is the solution of the original IVP at time  $t$  and  $yB$  is the solution of the backward problem at the same time. The rest of the arguments are equivalent to those of a function of type `CVSpgmrJtimesFn` (see §4.4 on page 36). If the backward problem is the adjoint of  $\dot{y} = f(t, y)$ , then this routine is to compute  $-(\partial f / \partial y)^T v_B$ .

The return value of a function of type `CVSpgmrJtimesFnB` should be 0 if successful or non-zero if an error was encountered, in which case the integration is halted.

- *Preconditioning for the backward problem (linear system solution)*

If preconditioning is used during integration of the backward problem, then the user must provide a C function to solve the linear system  $Pz = r$  where  $P$  may be either a left or a right preconditioner matrix. This function must be of type `CVSpgmrPSolveFnB` defined by

```
typedef int (*CVSpgmrPSolveFnB)(integertype NB, realtype t, N_Vector y,
                                N_Vector yB, N_Vector fyB,
                                N_Vector vtempB, realtype gammaB,
                                N_Vector ewtB, realtype deltaB,
                                long int *nfePtrB, N_Vector rB,
                                int lrB, void *P_dataB, N_Vector zB);
```

The only difference between this function type and `CVSpgmrPSolveFn` (defined in §4.4 on page 36) is that a function of type `CVSpgmrPSolveFnB` also receives as an argument  $y$ , the solution of the forward problem at time  $t$ .

The return value of a preconditioner solve routine for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).

- *Preconditioning for the backward problem (Jacobian data)*

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then this needs to be done in a user-supplied C function of type `CVSpgmrPrecondFnB` as defined by

```
typedef int (*CVSpgmrPrecondFnB)(integertype NB, realtype t, N_Vector y,
                                N_Vector yB, N_Vector fyB, booleantype jokB,
                                booleantype *jcurPtrB, realtype gammaB,
                                N_Vector ewtB, realtype hB, realtype oundB,
                                long int *nfePtrB, void *P_dataB,
                                N_Vector vtemp1B, N_Vector vtemp2B,
                                N_Vector vtemp3B);
```

This function type is identical to the type `CVSpgmrPrecondFn` (§4.4 on page 37) with the exception of the additional argument  $y$  which contains the solution of the forward problem at time  $t$ .

The return value of a preconditioner setup routine for the backward problem should be 0 if successful, positive for a recoverable error (in which case the step will be retried), negative for an unrecoverable error (in which case the integration is halted).

## 6.4 Using the Banded Preconditioner Module for Adjoint Sensitivity Analysis

The adjoint sensitivity module in CVODES offers an interface to the banded preconditioner module `CVBANDPRE` described in section §4.5.1. This preconditioner provides a band matrix preconditioner based on difference quotients of the backward problem right hand side function `fB`. It generates a banded approximation to the Jacobian with  $m_{lB}$  sub-diagonals and  $m_{uB}$  super-diagonals to be used with the Krylov linear solver in `CVSPGMR`.

In order to use the CVBANDPRE module in the solution of the backward problem, the user need not define any additional routines. Before the call to CVSpgrmrB (step 14 in §6.1), the user must initialize the CVBANDPRE module by calling

```
bp_dataB = CVBandPreAllocB(cvadj_mem, NB, muB, mlB);
```

where `cvadj_mem` is the pointer to the memory block allocated by `CVadjMalloc`, `NB` is the size of the backward problem, and `muB` and `mlB` are the upper and lower half-bandwidths, respectively. `CVBandPreAlloc` returns an object of type `CVBandPreData` which must then be passed to `CVSpgrmrB` as the `P_dataB` argument:

```
ier = CVSpgrmrB(cvadj_mem, pretypeB, gstypeB, maxlB, deltb,  
                CVBandPrecondB, CVBandPSolveB, bp_dataB,  
                jtimesB, jac_dataB);
```

Here the preconditioner setup and solve routines `CVBandPSolveB` and `CVBandPrecondB` are provided in the interface to the CVBANDPRE module. They are never called by the user explicitly but are simply passed as arguments to `CVSpgrmrB`. Although these functions simply call `CVBandPSolve` and `CVBandPrecond`, without any other processing, they need to be defined because they must be of the correct function types expected by `CVSpgrmrB`. For example, `CVBandPSolveB` must be of type `CVSpgrmrPSolveFnB` and thus receives `y`, the solution of the original IVP, as an argument (which is consequently ignored as it is not needed by CVBANDPRE).

## 7 Example Problems for IVP Solution

The CVODES distribution contains, in the `sundials/cvodes/examples` directory, the following nine examples for ODE IVP solution:

- **cvdx** solves a chemical kinetics problem consisting of three rate equations.  
This program solves the problem with the BDF method, Newton iteration with the CVDENSE linear solver, and a user-supplied Jacobian routine;
- **cvbx** solves the semi-discrete form of an advection-diffusion equation in 2-D.  
This program solves the problem with the BDF method, 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 space.  
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 left and right preconditioning;
- **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) band, user-supplied, (2) band, 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 defined implicitly, 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;

- **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.

The first six are serial examples that use the `NVECTOR_SERIAL` module and the last three are parallel examples using the `NVECTOR_PARALLEL` module.

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

## 7.1 A Serial Sample Problem

As an initial illustration of the use of the CVODES package for the integration of IVP ODEs, the following is a sample program provided as part of the package. It uses the CVODES dense linear solver module `CVDENSE` and the `NVECTOR_SERIAL` module (which provides a serial implementation of `NVECTOR`) in the solution of a small chemical kinetics problem. For the source listed in App. A.1, we give a rather detailed explanation of the parts of the program and their interaction with CVODES.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in CVODES header files. The `sundialtypes.h` file provides the definitions of the types `realtype` and `integertype` (see §10 for details). For now, it suffices to read `realtype` as `double` and `integertype` as `int`. The `cvodes.h` file provides prototypes for the three CVODES functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in dimensioning, setting input arguments, testing the return value of `CVode`, and accessing the integer optional outputs. The `cvdense.h` file provides the prototype for the `CVDense` function, and a constant `DENSE_NJE` for accessing optional output specific to `CVDENSE`. The `nvector_serial.h` file is the header file for the serial implementation of the `NVECTOR` module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. Finally, the `dense.h` file provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements. We have explicitly included `dense.h`, but this is not necessary because it is included by `cvdense.h`.

This program includes two user-defined accessor macros, `Ith` and `IJth` that are useful in writing the problem functions in a form closely matching the mathematical description of the ODE system, i.e. with components numbered from 1 instead of from 0. The `Ith` macro is used to access components of a vector of type `N_Vector` with a serial implementation. It is defined using the `NVECTOR_SERIAL` accessor macro `NV_Ith_S` which numbers components starting with 0. The `IJth`

macro is used to access elements of a dense matrix of type `DenseMat`. It is defined using the `DENSE` accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The macros `NV_Ith_S` and `DENSE_ELEM` are fully described in §11.1 and §13.1, respectively.

Next, the program includes some problem-specific constants, which are isolated to this early location to make it easy to change them as needed. The program prologue ends with the prototype of a private helper function and the two user-supplied functions that are called by `CVODES`.

The `main` function begins with some dimensions and type declarations. These make use of the constant `OPT_SIZE` and the type `N_Vector`. The first line initializes the serial machine environment by calling the `M_EnvInit_Serial` routine implemented by `NVECTOR_SERIAL` (see §11.1). The next two lines allocate memory for the `y` and `abstol` vectors using `N_VNew` with a length argument of `NEQ` ( $= 3$ ). The next several lines load the initial values of the dependent variable vector into `y` and set the absolute tolerance vector `abstol` using the `Ith` macro.

The call to `CVodeMalloc` specifies the BDF integration method with `NEWTON` iteration. The `SV` argument specifies a vector of absolute tolerances, and this is followed by the address of the relative tolerance `reltol` and the absolute tolerance vector `abstol`. The `FALSE` argument indicates that no optional inputs are present in `iopt` or `ropt`. The two `NULL` actual parameters in the `CVodeMalloc` call are for features that are not used in this example. The first one is passed for the `CVodeMalloc` formal parameter `f_data`. This pointer is passed to `f` every time `f` is called, and is intended to point to user problem data that might be needed in `f`. The second `NULL` forces `CVODES` error messages to be sent to standard output; a file pointer (of type `FILE*`) may be given in this position otherwise. The return value of `CVodeMalloc` is a pointer to a `CVODES` solver memory block for the problem and solver options specified by the inputs. In the case of failure, the return value is `NULL`. This pointer must be passed in the remaining calls to `CVODES` functions. See §4.3.1 for full details of the call to `CVodeMalloc`.

The call to `CVDense` with a non-`NULL` Jacobian function `Jac` specifies the `CVDENSE` linear solver with an analytic Jacobian supplied by the user-supplied function `Jac`. The `NULL` argument is passed for the `CVDense` formal parameter `jac_data`. In a role similar to `f_data`, this pointer is passed to `Jac` every time `Jac` is called, and is intended to point to user problem data that might be needed in `Jac`. See §4.3.2 for full details of the call to `CVDense`.

The actual solution of the ODE initial value problem is accomplished in the loop over values of `tout`. For each value, the program calls `CVode` in the `NORMAL` mode, meaning that the integrator takes steps until it overshoots `tout` and then interpolates to  $t = \text{tout}$ , putting the computed value of  $y(\text{tout})$  into `y`. The program prints `t` and `y`, and tests for a return value other than `SUCCESS` by `CVode`. See §4.3.3 for full details of the call to `CVode`.

Finally, the main program calls `NV_Free` to free the vectors `y` and `abstol`, calls `CVodeFree` to free the `CVODES` memory block, calls `M_EnvFree_Serial` to free the serial machine environment memory block, and prints all of the statistical quantities in the private helper function `PrintFinalStats`. See §11.1 for details on `M_EnvFree_Serial`.

The function `PrintFinalStats` used here is actually suitable for general use in connection with the use of `CVODES` for any problem with a dense Jacobian. It prints the cumulative number of steps (`nst`), the number of `f` evaluations (`nfe`), the number of matrix factorizations (`nsetups`), the number of `Jac` evaluations (`nje`), the number of nonlinear iterations (`nni`), the number of nonlinear convergence failures (`ncnf`), and the number of local error test failures (`netf`). These optional outputs are described in §4.3.4, except for `nje = iopt[DENSE_NJE]`, which is described in §4.3.2.



The function `f` is a straightforward expression of the ODEs. It uses the user-defined macro `Ith` to extract the components of `y` and load the components of `ydot`. See §4.4 for a detailed specification of `f`.

The function `Jac` sets the nonzero elements of the Jacobian as a dense matrix. (Zero elements need not be set because `J` is preset to zero.) It uses the user-defined macro `IJth` to reference the elements of a dense matrix of type `DenseMat`. Here the problem size is small, so we need not worry about the inefficiency of using `NV_Ith_S` and `DENSE_ELEM` to do `N_Vector` and `DenseMat` element accesses. Note that in this example `Jac` only accesses the `y` and `J` arguments. See §4.4 for a detailed description of the dense `Jac` function.

The output generated by `cvdx` is shown below.

3-species kinetics problem

At t = 4.0000e-01	y = 9.851641e-01	3.386242e-05	1.480205e-02
At t = 4.0000e+00	y = 9.055097e-01	2.240338e-05	9.446793e-02
At t = 4.0000e+01	y = 7.158014e-01	9.185060e-06	2.841895e-01
At t = 4.0000e+02	y = 4.505419e-01	3.223113e-06	5.494549e-01
At t = 4.0000e+03	y = 1.832057e-01	8.942854e-07	8.167934e-01
At t = 4.0000e+04	y = 3.898153e-02	1.621745e-07	9.610183e-01
At t = 4.0000e+05	y = 4.936102e-03	1.984115e-08	9.950639e-01
At t = 4.0000e+06	y = 5.165912e-04	2.067419e-09	9.994834e-01
At t = 4.0000e+07	y = 5.202658e-05	2.081170e-10	9.999480e-01
At t = 4.0000e+08	y = 5.202811e-06	2.081135e-11	9.999948e-01
At t = 4.0000e+09	y = 5.206213e-07	2.082486e-12	9.999995e-01
At t = 4.0000e+10	y = 5.099726e-08	2.039890e-13	9.999999e-01

Final Statistics..

nst = 536	nfe = 789	nsetups = 116	nje = 12
nmi = 786	ncfn = 0	netf = 32	

## 7.2 A Parallel Sample Program

As an example of using CVODES with the Krylov linear solver CVSPGMR and the parallel MPI NVECTOR\_PARALLEL module, we describe a test problem based on a two-dimensional system of two PDEs involving diurnal kinetics, advection, and diffusion. These equations represent a simplified model for the transport, production, and loss of the oxygen singlet and ozone in the upper atmosphere. The PDEs can be written as

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y} K_v(y) \frac{\partial c^i}{\partial y} + R^i(c^1, c^2, t) \quad (i = 1, 2), \quad (29)$$

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

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2 \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 - q_4(t) c^2 \end{aligned} \quad (30)$$

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

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

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

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

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

We discretize the PDE system with central differencing, to obtain an ODE system  $\dot{u} = f(t, u)$  representing (29). For this example, we may think of the processors as being laid out in a rectangle, and each processor being assigned a subgrid of size  $\text{MXSUB} \times \text{MYSUB}$  of the  $x - y$  grid. If there are  $\text{NPEX}$  processors in the  $x$  direction and  $\text{NPEY}$  processors in the  $y$  direction then the overall grid size is  $\text{MX} \times \text{MY}$  with  $\text{MX} = \text{NPEX} \times \text{MXSUB}$  and  $\text{MY} = \text{NPEY} \times \text{MYSUB}$ . There are  $2 \times \text{MX} \times \text{MY}$  equations in this system of ODEs. To compute  $f$  in this setting, the processors pass and receive information as follows. The solution components for the bottom row of grid points in the current processor are passed to the processor below it and the solution for the top row of grid points is received from the processor below the current processor. The solution for the top row of grid points for the current processor is sent to the processor above the current processor, while the solution for the bottom row of grid points is received from that processor by the current processor. Similarly the solution for the first column of grid points is sent from the current processor to the processor to its left and the last column of grid points is received from that processor by the current processor. The communication for the solution at the right edge of the processor is similar. If this is the last processor in a particular direction, then message passing and receiving are bypassed for that direction.

The code listing for this example is given in App. A.2. The purpose of this code is to provide a more complicated example than Example 1, and to provide a template for a stiff ODE system arising from a PDE system. The solution method is BDF with Newton iteration and SPGMR. The left preconditioner is the block-diagonal part of the Newton matrix, with  $2 \times 2$  blocks, and the corresponding diagonal blocks of the Jacobian are saved each time the preconditioner is generated, for re-use later under certain conditions.

The organization of the `pvkx` program deserves some comments. The right-hand side routine `f` calls two other routines: `ucomm`, which carries out inter-processor communication; and `fcalc` which operates on local data only and contains the actual calculation of  $f(t, u)$ . The `ucomm` function in turn calls three routines which do, respectively, non-blocking receive operations, blocking send operations, and receive-waiting. All three use MPI, and transmit data from the local `u` vector into a local working array `uext`, an extended copy of `u`. The `fcalc` function copies `u` into `uext`, so that the calculation of  $f(t, u)$  can be done conveniently by operations on `uext` only.

Sample output from `pvkx` follows. The output will vary if the number of processors is changed. The output is for four processors (in a  $2 \times 2$  array) with a  $5 \times 5$  subgrid on each processor.

## 2-species diurnal advection-diffusion problem

t = 7.20e+03 no. steps = 219 order = 5 stepsize = 1.59e+02  
At bottom left: c1, c2 = 1.047e+04 2.527e+11  
At top right: c1, c2 = 1.119e+04 2.700e+11

t = 1.44e+04 no. steps = 251 order = 5 stepsize = 3.77e+02  
At bottom left: c1, c2 = 6.659e+06 2.582e+11  
At top right: c1, c2 = 7.301e+06 2.833e+11

t = 2.16e+04 no. steps = 277 order = 5 stepsize = 2.75e+02  
At bottom left: c1, c2 = 2.665e+07 2.993e+11  
At top right: c1, c2 = 2.931e+07 3.313e+11

t = 2.88e+04 no. steps = 317 order = 4 stepsize = 3.89e+02  
At bottom left: c1, c2 = 8.702e+06 3.380e+11  
At top right: c1, c2 = 9.650e+06 3.751e+11

t = 3.60e+04 no. steps = 354 order = 4 stepsize = 6.82e+01  
At bottom left: c1, c2 = 1.404e+04 3.387e+11  
At top right: c1, c2 = 1.561e+04 3.765e+11

t = 4.32e+04 no. steps = 418 order = 3 stepsize = 3.80e+02  
At bottom left: c1, c2 = -8.138e-07 3.382e+11  
At top right: c1, c2 = -1.218e-06 3.804e+11

t = 5.04e+04 no. steps = 439 order = 4 stepsize = 3.81e+02  
At bottom left: c1, c2 = 4.720e-15 3.358e+11  
At top right: c1, c2 = 6.925e-15 3.864e+11

t = 5.76e+04 no. steps = 453 order = 5 stepsize = 3.47e+02  
At bottom left: c1, c2 = -2.056e-15 3.320e+11  
At top right: c1, c2 = -3.729e-15 3.909e+11

t = 6.48e+04 no. steps = 465 order = 5 stepsize = 6.29e+02  
At bottom left: c1, c2 = 5.079e-13 3.313e+11  
At top right: c1, c2 = 9.699e-13 3.963e+11

t = 7.20e+04 no. steps = 476 order = 5 stepsize = 6.29e+02  
At bottom left: c1, c2 = 1.955e-14 3.330e+11  
At top right: c1, c2 = 3.117e-14 4.039e+11

t = 7.92e+04 no. steps = 488 order = 5 stepsize = 6.29e+02  
At bottom left: c1, c2 = -1.854e-16 3.334e+11  
At top right: c1, c2 = -3.388e-16 4.120e+11

t = 8.64e+04 no. steps = 499 order = 5 stepsize = 6.29e+02  
At bottom left: c1, c2 = -1.189e-18 3.352e+11  
At top right: c1, c2 = -2.480e-18 4.163e+11

Final Statistics..

lenrw	=	2000	leniw	=	0
llrw	=	2046	lliw	=	0
nst	=	499	nfe	=	1293
nni	=	649	nli	=	641
nsetups	=	87	netf	=	33
npe	=	9	nps	=	1226
ncfn	=	0	ncfl	=	0

## 8 Example Problems for Forward Sensitivity Analysis

The CVODES distribution contains, in the `sundials/cvodes/fwd_examples` directory, the following five examples for forward sensitivity analysis:

- **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;
- **cvfdx** solves a chemical kinetics problem consisting of three rate equations.  
CVODES computes both its solution and solution sensitivities with respect to the three reaction rate constants appearing in the model. This program solves the problem with the BDF method, Newton iteration with the CVDENSE linear solver, and a user-supplied Jacobian routine;
- **cvfkx** solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D space.  
CVODES computes both its solution and solution sensitivities with respect to two parameters affecting the kinetic rate terms. The problem is solved with the BDF/GMRES method (i.e. using the CVSPGMR linear solver) and the block-diagonal part of the Newton matrix as a left preconditioner;
- **pvfnx** is the parallel version of **cvfnx**;
- **pvfkx** is the parallel version of **cvfkx**.

The first three are serial examples that use the `NVECTOR_SERIAL` module and the last two are parallel examples using the `NVECTOR_PARALLEL` module. For all the above examples, any of three sensitivity methods (`SIMULTANEOUS`, `STAGGERED`, and `STAGGERED1`) can be used and sensitivities may be included in the error test or not (error control set on `FULL` or `PARTIAL`, respectively).

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

### 8.1 A Serial Sample Problem

As a first example of using CVODES for forward sensitivity analysis, we provide a modification of the chemical kinetics problem described in §7.1 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:

$$\begin{aligned}\dot{y}_1 &= -p_1 y_1 + p_2 y_2 y_3 \\ \dot{y}_2 &= p_1 y_1 - p_2 y_2 y_3 - p_3 y_2^2 \\ \dot{y}_3 &= p_3 y_2^2,\end{aligned}\tag{31}$$

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 (31) are

$$\begin{aligned} \dot{s}_i &= \begin{bmatrix} -p_1 & p_2 y_3 & p_2 y_2 \\ p_1 & -p_2 y_3 - 2p_3 y_2 & -p_2 y_2 \\ 0 & 2p_3 y_2 & 0 \end{bmatrix} s_i + \frac{\partial f}{\partial p_i}, \quad s_i(t_0) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad i = 1, 2, 3 \\ \frac{\partial f}{\partial p_1} &= \begin{bmatrix} -y_1 \\ y_1 \\ 0 \end{bmatrix}, \quad \frac{\partial f}{\partial p_2} = \begin{bmatrix} y_2 y_3 \\ -y_2 y_3 \\ 0 \end{bmatrix}, \quad \frac{\partial f}{\partial p_3} = \begin{bmatrix} 0 \\ -y_2^2 \\ y_2^2 \end{bmatrix}. \end{aligned} \quad (32)$$

The source code for this example is listed in App. B.1. The main program is described below with emphasis on the sensitivity related components. These explanations, together with those given for the code `cvdx` in §7.1, 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 in 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 §5.1 and §5.2.1, 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.

The program prologue ends by defining three additional private helper functions. The first one, `WrongArgs` (which would not be present in a typical user code) prints a usage message and stops execution if the command line arguments to `cvfdx` are wrong. After each successful return from the main CVODES integrator, the functions `PrintOutput` and `PrintOutputS` print the state and sensitivity variables, respectively. The program does not define any additional user-supplied functions since this example uses the CVODES internal difference quotient routines to compute the sensitivity equations right hand sides (see below).

The `main` function begins with definitions and type declarations. In addition to those in `cvdx.s`, it defines the vector `pbar` of `NP` scaling factors for the model parameters `p`, the array `plist` of `integertype` flags specifying the sensitivity parameters among the model parameters, and the array `yS` of `N_Vector`'s 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-data structure to be passed to CVODES and used in the evaluation of the ODEs 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.

Next, the serial machine environment variable `machEnv` is initialized by calling the `NVECTOR_SERIAL` function `M_EnvInit_Serial` and the user-data structure is allocated and initialized with the model parameter values.

The next block of code is identical to that in `cvdx.c` and involves allocation and initialization of the state variables and integration tolerances for the state variables, initialization of `cvode_mem`

and of the CVODES solver memory. It also attaches CVDENSE, with a non-NULL Jacobian function, as the linear solver to be used in the Newton nonlinear solver.

If sensitivity analysis is enabled (through the command line arguments), the main program will then set the scaling parameters `pbar` ( $pbar_i = p_i$ , which can typically be used for non-zero model parameters) and the array of flags `plist`. The choice  $plist_i = i + 1$  indicates that sensitivities with respect to all model parameters will be computed. Next, the program allocates memory for `yS`, by calling the NVECTOR function `N_VNew_S`, and initializes all sensitivity variables to 0.0.

The call to `CVodeSensMalloc` specifies the sensitivity solution method through `sensimeth` (read from the command line arguments) as `SIMULTANEOUS`, `STAGGERED`, or `STAGGERED1` and the error control strategy through `err_con` (also read from the command line arguments) as either `FULL` or `PARTIAL`. The `ifS` parameter indicates the type of sensitivity right hand side function (`ALLSENS`, or `ONESENS` if the `STAGGERED1` approach was selected), while the first NULL parameter indicates that the CVODES internal difference quotient routines (`CVSensRhsDQ` or `CVSensRhsDQ1`, depending on the value of `ifS`) should be used. The last two NULL parameters in the call to `CVodeSensMalloc` force CVODES to set the relative and absolute tolerances `rtolS` and `atolS` for sensitivity variables based on the tolerances for state variables and the scaling parameters `pbar` (see §2.2 for details).

The return value of `CVodeSensMalloc` is an `int` flag. In case of failure (`flag != SUCCESS`), the main program prints an error message and stops execution.

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 `SUCCESS` and prints the state variables. Sensitivity variables at `tout` are loaded into `yS` by calling `CVodeSensExtract`. The program tests the return from `CVodeSensExtract` for a value other than `SUCCESS` and then prints the sensitivity variables.

Finally, the program prints some final statistics and deallocates memory through calls to `N_VFree`, `N_VFree_S`, `CVodeFree`, and `M_EnvFree_Serial`.

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-data structure `f_data`, which must be first cast to the `UserData` type.

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 `SIMULTANEOUS`, `STAGGERED`, or `STAGGERED1` method, respectively and `err_con` must be one of `full` or `partial` to select the `FULL` or `PARTIAL` error control strategy, respectively.

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

```
3-species chemical kinetics problem
```

```
=====
```

T	Q	H	NST		y1	y2	y3
4.000e-01	3	3.507e-02	106				
				Solution	9.8517e-01	3.3864e-05	1.4794e-02
				Sensitivity 1	-3.5595e-01	3.9026e-04	3.5556e-01
				Sensitivity 2	9.5428e-08	-2.1310e-10	-9.5215e-08
				Sensitivity 3	-1.5832e-11	-5.2900e-13	1.6361e-11
4.000e+00	4	1.705e-01	145				
				Solution	9.0552e-01	2.2405e-05	9.4458e-02
				Sensitivity 1	-1.8761e+00	1.7922e-04	1.8759e+00
				Sensitivity 2	2.9615e-06	-5.8304e-10	-2.9609e-06
				Sensitivity 3	-4.9336e-10	-2.7627e-13	4.9363e-10
4.000e+01	3	1.872e+00	226				
				Solution	7.1582e-01	9.1854e-06	2.8417e-01
				Sensitivity 1	-4.2476e+00	4.5906e-05	4.2476e+00
				Sensitivity 2	1.3731e-05	-2.3572e-10	-1.3731e-05
				Sensitivity 3	-2.2884e-09	-1.1381e-13	2.2885e-09
4.000e+02	3	1.006e+01	317				
				Solution	4.5051e-01	3.2228e-06	5.4949e-01
				Sensitivity 1	-5.9584e+00	3.5418e-06	5.9584e+00
				Sensitivity 2	2.2738e-05	-2.2595e-11	-2.2738e-05
				Sensitivity 3	-3.7897e-09	-4.9947e-14	3.7898e-09
4.000e+03	2	1.486e+02	487				
				Solution	1.8319e-01	8.9415e-07	8.1681e-01
				Sensitivity 1	-4.7500e+00	-5.9942e-06	4.7500e+00
				Sensitivity 2	1.8809e-05	2.3128e-11	-1.8809e-05
				Sensitivity 3	-3.1348e-09	-1.8758e-14	3.1348e-09
4.000e+04	3	2.000e+03	567				
				Solution	3.8978e-02	1.6215e-07	9.6102e-01
				Sensitivity 1	-1.5748e+00	-2.7622e-06	1.5749e+00
				Sensitivity 2	6.2870e-06	1.1003e-11	-6.2870e-06
				Sensitivity 3	-1.0478e-09	-4.5363e-15	1.0478e-09
4.000e+05	3	1.765e+04	625				
				Solution	4.9391e-03	1.9853e-08	9.9506e-01
				Sensitivity 1	-2.3638e-01	-4.5854e-07	2.3639e-01
				Sensitivity 2	9.4523e-07	1.8330e-12	-9.4523e-07
				Sensitivity 3	-1.5752e-10	-6.3633e-16	1.5752e-10
4.000e+06	4	2.252e+05	678				
				Solution	5.1687e-04	2.0685e-09	9.9948e-01
				Sensitivity 1	-2.5669e-02	-5.1067e-08	2.5669e-02
				Sensitivity 2	1.0267e-07	2.0425e-13	-1.0267e-07
				Sensitivity 3	-1.7111e-11	-6.8516e-17	1.7111e-11
4.000e+07	3	1.457e+06	740				



				Solution	5.2046e-05	2.0820e-10	9.9995e-01
				Sensitivity 1	-2.6001e-03	-5.1968e-09	2.6001e-03
				Sensitivity 2	1.0400e-08	2.0787e-14	-1.0400e-08
				Sensitivity 3	-1.7333e-12	-6.9338e-18	1.7333e-12
-----							
4.000e+08	3	2.570e+07	796				
				Solution	5.2108e-06	2.0843e-11	9.9999e-01
				Sensitivity 1	-2.6043e-04	-5.2069e-10	2.6043e-04
				Sensitivity 2	1.0417e-09	2.0828e-15	-1.0417e-09
				Sensitivity 3	-1.7367e-13	-6.9470e-19	1.7367e-13
-----							
4.000e+09	3	3.805e+08	828				
				Solution	5.1965e-07	2.0786e-12	1.0000e+00
				Sensitivity 1	-2.6131e-05	-5.2722e-11	2.6131e-05
				Sensitivity 2	1.0452e-10	2.1089e-16	-1.0452e-10
				Sensitivity 3	-1.7322e-14	-6.9288e-20	1.7322e-14
-----							
4.000e+10	3	7.725e+09	851				
				Solution	5.8386e-08	2.3355e-13	1.0000e+00
				Sensitivity 1	-2.7256e-06	-5.0248e-12	2.7257e-06
				Sensitivity 2	1.0903e-11	2.0099e-17	-1.0903e-11
				Sensitivity 3	-1.9441e-15	-7.7764e-21	1.9441e-15
-----							

```

=====
Final Statistics
Sensitivity: YES ( SIMULTANEOUS + FULL ERROR CONTROL )

nst      =    851

nfe      =   7987      nfSe =   1141
nni      =   1138      nniS =     0
ncfn     =     4       ncfnS =     0
netf     =    17       netfS =     0

nsetups  =    136
nje      =     24
=====

```

The output generated by cvfdx when computing sensitivities with the STAGGERED1 method and PARTIAL error control (cvfdx -sensi stg1 partial) is:

3-species chemical kinetics problem

T	Q	H	NST		y1	y2	y3
4.000e-01	3	1.153e-01	60				
				Solution	9.8517e-01	3.3863e-05	1.4797e-02
				Sensitivity 1	-3.5609e-01	3.9023e-04	3.5570e-01
				Sensitivity 2	9.4898e-08	-2.1323e-10	-9.4685e-08

				Sensitivity 3	-1.5744e-11	-5.2897e-13	1.6273e-11
<hr/>							
4.000e+00	4	4.988e-01	74	Solution	9.0552e-01	2.2404e-05	9.4458e-02
				Sensitivity 1	-1.8761e+00	1.7920e-04	1.8759e+00
				Sensitivity 2	2.9614e-06	-5.8312e-10	-2.9608e-06
				Sensitivity 3	-4.9343e-10	-2.7624e-13	4.9371e-10
<hr/>							
4.000e+01	4	3.084e+00	149	Solution	7.1583e-01	9.1856e-06	2.8416e-01
				Sensitivity 1	-4.2476e+00	4.5911e-05	4.2475e+00
				Sensitivity 2	1.3731e-05	-2.3572e-10	-1.3730e-05
				Sensitivity 3	-2.2885e-09	-1.1381e-13	2.2887e-09
<hr/>							
4.000e+02	3	8.617e+00	253	Solution	4.5055e-01	3.2232e-06	5.4945e-01
				Sensitivity 1	-5.9583e+00	3.5447e-06	5.9583e+00
				Sensitivity 2	2.2738e-05	-2.2614e-11	-2.2738e-05
				Sensitivity 3	-3.7895e-09	-4.9951e-14	3.7896e-09
<hr/>							
4.000e+03	3	7.015e+01	309	Solution	1.8323e-01	8.9446e-07	8.1677e-01
				Sensitivity 1	-4.7502e+00	-5.9923e-06	4.7502e+00
				Sensitivity 2	1.8809e-05	2.3118e-11	-1.8809e-05
				Sensitivity 3	-3.1345e-09	-1.8756e-14	3.1345e-09
<hr/>							
4.000e+04	4	2.367e+03	376	Solution	3.8981e-02	1.6216e-07	9.6102e-01
				Sensitivity 1	-1.5748e+00	-2.7616e-06	1.5748e+00
				Sensitivity 2	6.2869e-06	1.1001e-11	-6.2869e-06
				Sensitivity 3	-1.0480e-09	-4.5369e-15	1.0480e-09
<hr/>							
4.000e+05	4	1.343e+04	418	Solution	4.9411e-03	1.9861e-08	9.9506e-01
				Sensitivity 1	-2.3635e-01	-4.5819e-07	2.3635e-01
				Sensitivity 2	9.4504e-07	1.8315e-12	-9.4504e-07
				Sensitivity 3	-1.5763e-10	-6.3679e-16	1.5763e-10
<hr/>							
4.000e+06	4	2.009e+05	462	Solution	5.1705e-04	2.0693e-09	9.9948e-01
				Sensitivity 1	-2.5660e-02	-5.1012e-08	2.5660e-02
				Sensitivity 2	1.0265e-07	2.0411e-13	-1.0265e-07
				Sensitivity 3	-1.7124e-11	-6.8567e-17	1.7124e-11
<hr/>							
4.000e+07	3	2.705e+06	519	Solution	5.2016e-05	2.0808e-10	9.9995e-01
				Sensitivity 1	-2.5994e-03	-5.1969e-09	2.5994e-03
				Sensitivity 2	1.0398e-08	2.0789e-14	-1.0398e-08
				Sensitivity 3	-1.7327e-12	-6.9316e-18	1.7327e-12
<hr/>							
4.000e+08	3	2.965e+07	555				

				Solution	5.2035e-06	2.0814e-11	9.9999e-01
				Sensitivity 1	-2.6029e-04	-5.2083e-10	2.6029e-04
				Sensitivity 2	1.0413e-09	2.0839e-15	-1.0413e-09
				Sensitivity 3	-1.7343e-13	-6.9374e-19	1.7343e-13
-----							
4.000e+09	3	5.173e+08	582				
				Solution	5.2150e-07	2.0860e-12	1.0000e+00
				Sensitivity 1	-2.5711e-05	-5.0779e-11	2.5711e-05
				Sensitivity 2	1.0343e-10	2.0546e-16	-1.0343e-10
				Sensitivity 3	-1.7220e-14	-6.8881e-20	1.7220e-14
-----							
4.000e+10	3	4.476e+09	599				
				Solution	4.4137e-08	1.7655e-13	1.0000e+00
				Sensitivity 1	-2.3026e-06	-4.7966e-12	2.3026e-06
				Sensitivity 2	1.1004e-11	2.6360e-17	-1.1004e-11
				Sensitivity 3	-1.8767e-15	-7.5069e-21	1.8767e-15
-----							

=====

Final Statistics

Sensitivity: YES ( STAGGERED1 + PARTIAL ERROR CONTROL )

nst = 599

nfe	=	6413	nfSe	=	2510
nni	=	791	nniS	=	2507
ncfn	=	0	ncfnS	=	0
netf	=	24	netfS	=	0

nsetups = 112

nje = 13

=====

## 8.2 A Parallel Sample Program

The second example program for forward sensitivity analysis, `pvfkk` is the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D space (29), described in §7.2, for which we compute solution sensitivities with respect to the parameters  $q_1$  and  $q_2$  that appear in the kinetic rate terms (30).

The source code for this example is listed in App. B.2. The overall structure of the `main` function is very similar to that of the code `cvfdx` (§8.1) with differences arising from the use of the parallel vector module. On the other hand, the user-supplied routines in `pvfkk`, `f` for the right-hand side of the original system, `Precond` for the preconditioner set-up, and `PSolve` for the preconditioner solve, are identical to those defined for the sample program `pvkx` (see §7.2). 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 `pvfkk` defines no additional user-supplied routines, as it uses the CVODES internal

difference quotient routines to compute the sensitivity equation right hand sides (as indicated by passing NULL for `fS` in the call to `CVodeSensMalloc`).

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

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

if no sensitivity calculations are desired, or

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

where `nproc` is the number of processes, `sensi_meth` must be one of `sim`, `stg`, or `stg1` to indicate the SIMULTANEOUS, STAGGERED, or STAGGERED1 method, respectively and `err_con` must be one of `full` or `partial` to select the FULL or PARTIAL error control strategy, respectively.

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

2-species diurnal advection-diffusion problem

```
=====
      T      Q      H      NST
=====
7.200e+03  3  3.757e+01  613
      Solution      1.0468e+04  1.1185e+04
                      2.5267e+11  2.6998e+11
      -----
      Sensitivity 1  -6.4201e+19 -6.8598e+19
                      7.1178e+19  7.6557e+19
      -----
      Sensitivity 2  -4.3853e+14 -5.0065e+14
                      -2.4408e+18 -2.7843e+18
      -----
1.440e+04  3  4.063e+01  806
      Solution      6.6590e+06  7.3008e+06
                      2.5819e+11  2.8329e+11
      -----
      Sensitivity 1  -4.0848e+22 -4.4785e+22
                      5.9549e+22  6.7173e+22
      -----
      Sensitivity 2  -4.5235e+17 -5.4318e+17
                      -6.5418e+21 -7.8315e+21
      -----
2.160e+04  3  4.223e+01  1325
      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 -9.4501e+22
      -----
```

2.880e+04	2	3.242e+01	1593			
				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	4.744e+01	1708			
				Solution	1.4040e+04	1.5609e+04
					3.3868e+11	3.7652e+11
				Sensitivity 1	-8.6141e+19	-9.5761e+19
					5.2718e+23	6.6030e+23
				Sensitivity 2	-8.4328e+15	-1.0549e+16
					-1.8439e+23	-2.3096e+23
4.320e+04	4	1.375e+02	1884			
				Solution	1.1785e-06	1.2879e-06
					3.3823e+11	3.8035e+11
				Sensitivity 1	2.0518e+08	2.2861e+08
					5.2753e+23	6.7448e+23
				Sensitivity 2	4.1216e+05	6.7832e+05
					-1.8454e+23	-2.3595e+23
5.040e+04	4	2.551e+02	1919			
				Solution	3.1876e-06	3.5142e-06
					3.3582e+11	3.8645e+11
				Sensitivity 1	4.2351e+11	4.6705e+11
					5.2067e+23	6.9664e+23
				Sensitivity 2	1.5899e+08	1.9244e+08
					-1.8214e+23	-2.4370e+23
5.760e+04	4	2.269e+02	1944			
				Solution	1.9220e-09	2.0977e-09
					3.3203e+11	3.9090e+11
				Sensitivity 1	3.1095e+08	3.4117e+08
					5.0825e+23	7.1205e+23
				Sensitivity 2	-2.7959e+03	-3.3004e+03
					-1.7780e+23	-2.4910e+23

6.480e+04 4 2.472e+02 2011

Solution	-1.3922e-07	-1.4787e-07
	3.3130e+11	3.9634e+11
-----		
Sensitivity 1	-3.9244e+10	-4.1562e+10
	5.0442e+23	7.3274e+23
-----		
Sensitivity 2	2.5992e+08	3.3160e+08
	-1.7646e+23	-2.5633e+23

7.200e+04 4 3.633e+02 2031

Solution	6.3840e-08	6.7901e-08
	3.3297e+11	4.0389e+11
-----		
Sensitivity 1	-7.6522e+10	-8.1142e+10
	5.0783e+23	7.6382e+23
-----		
Sensitivity 2	-7.0536e+05	-9.0599e+05
	-1.7765e+23	-2.6721e+23

7.920e+04 5 5.981e+02 2047

Solution	-1.9542e-11	-2.0990e-11
	3.3344e+11	4.1203e+11
-----		
Sensitivity 1	3.6012e+08	3.8075e+08
	5.0730e+23	7.9960e+23
-----		
Sensitivity 2	3.1602e+03	4.0628e+03
	-1.7747e+23	-2.7972e+23

8.640e+04 5 5.981e+02 2059

Solution	1.9297e-13	2.0929e-13
	3.3518e+11	4.1625e+11
-----		
Sensitivity 1	1.0203e+07	1.0864e+07
	5.1171e+23	8.2142e+23
-----		
Sensitivity 2	9.2985e+01	1.2081e+02
	-1.7901e+23	-2.8736e+23

=====

Final Statistics

Sensitivity: YES ( SIMULTANEOUS + FULL ERROR CONTROL )

nst = 2059

nfe = 20935      nfSe = 2909

nni = 2907      nniS = 0

ncfn = 5      ncfnS = 0

```
netf      = 125      netfS = 0
```

```
nsetups = 358
```

```
nli      = 6390      ncfl = 0
```

```
npe      = 45      nps  = 13756
```

```
=====
```

The output generated by pvfkx when computing sensitivities with the STAGGERED1 method and PARTIAL error control (mpirun -np 4 pvfkx -sensi stg1 partial) is:

2-species diurnal advection-diffusion problem

=====						
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	3.892e+02	317			
				Solution	8.7021e+06	9.6500e+06
					3.3804e+11	3.7510e+11
				-----		
				Sensitivity 1	-5.3375e+22	-5.9187e+22
					5.4487e+23	6.7430e+23
				-----		

				Sensitivity 2	-4.8855e+18	-6.1040e+18
					-1.7194e+23	-2.1518e+23
-----						
3.600e+04	4	6.819e+01	354			
				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	-8.4327e+15	-1.0549e+16
					-1.8439e+23	-2.3096e+23
-----						
4.320e+04	3	3.803e+02	418			
				Solution	-8.1385e-07	-1.2180e-06
					3.3823e+11	3.8035e+11
-----						
				Sensitivity 1	1.7207e+10	2.5638e+10
					5.2753e+23	6.7448e+23
-----						
				Sensitivity 2	1.4577e+10	1.1367e+11
					-1.8454e+23	-2.3595e+23
-----						
5.040e+04	4	3.814e+02	439			
				Solution	4.7196e-15	6.9247e-15
					3.3582e+11	3.8644e+11
-----						
				Sensitivity 1	-2.0944e+01	-3.0544e+01
					5.2067e+23	6.9664e+23
-----						
				Sensitivity 2	-1.1189e+00	-8.7265e+00
					-1.8214e+23	-2.4370e+23
-----						
5.760e+04	5	3.468e+02	453			
				Solution	-2.0561e-15	-3.7287e-15
					3.3203e+11	3.9090e+11
-----						
				Sensitivity 1	-2.3030e+02	-4.3335e+02
					5.0825e+23	7.1205e+23
-----						
				Sensitivity 2	-1.3363e-02	-1.0547e-01
					-1.7780e+23	-2.4909e+23
-----						
6.480e+04	5	6.295e+02	465			
				Solution	5.0794e-13	9.6988e-13
					3.3130e+11	3.9634e+11
-----						
				Sensitivity 1	1.6992e+06	3.2384e+06
					5.0442e+23	7.3274e+23
-----						
				Sensitivity 2	2.3651e+03	4.6473e+03



```

-----
-1.7646e+23 -2.5633e+23
-----
7.200e+04  5  6.295e+02  476
Solution      1.9550e-14  3.1172e-14
              3.3297e+11  4.0388e+11
-----
Sensitivity 1  4.3958e+06  8.3486e+06
              5.0783e+23  7.6382e+23
-----
Sensitivity 2 -1.9272e+04 -3.7518e+04
              -1.7765e+23 -2.6721e+23
-----
7.920e+04  5  6.295e+02  488
Solution      -1.8536e-16 -3.3879e-16
              3.3344e+11  4.1203e+11
-----
Sensitivity 1 -4.0622e+05 -7.7061e+05
              5.0730e+23  7.9960e+23
-----
Sensitivity 2 -6.9109e+02 -1.3358e+03
              -1.7747e+23 -2.7972e+23
-----
8.640e+04  5  6.295e+02  499
Solution      -1.1889e-18 -2.4800e-18
              3.3518e+11  4.1625e+11
-----
Sensitivity 1 -7.5772e+03 -1.4351e+04
              5.1171e+23  8.2142e+23
-----
Sensitivity 2 -1.1481e+01 -2.2086e+01
              -1.7901e+23 -2.8736e+23
-----

```

=====

Final Statistics

Sensitivity: YES ( STAGGERED1 + PARTIAL ERROR CONTROL )

nst = 499

nfe = 5214      nfSe = 1112

nni = 612      nniS = 1110

ncfn = 0      ncfnS = 0

netf = 33      netfS = 0

nsetups = 87

nli = 1876      ncfl = 0

npe = 9      nps = 3411

=====

## 9 Example Problems for Adjoint Sensitivity Analysis

The CVODES distribution contains, in the `sundials/cvodes/adj_examples` directory, the following five examples for forward sensitivity analysis:

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

The adjoint capability of CVODES is used to compute gradients of a quantity of the form (17) 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;

- **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;

- **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;

- **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.

The first four are serial examples that use the `NVECTOR_SERIAL` module and the last one is a parallel example using the `NVECTOR_PARALLEL` module.

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.

## 9.1 A Serial Sample Problem

As a first example of using CVODES for adjoint sensitivity analysis we examine the chemical kinetics problem of §7.1 and §8.1:

$$\begin{aligned}\dot{y}_1 &= -p_1 y_1 + p_2 y_2 y_3 \\ \dot{y}_2 &= p_1 y_1 - p_2 y_2 y_3 - p_3 y_2^2 \\ \dot{y}_3 &= p_3 y_2^2 \\ y(t_0) &= y_0,\end{aligned}\tag{33}$$

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

$$G(p) = \int_{t_0}^{t_1} (y_1 + p_2 y_2 y_3) dt,\tag{34}$$

without having to compute the solution sensitivities  $dy/dp$ . Following the derivation of §2.3 and taking into account the fact that the initial values of (33) do not depend on the parameters  $p$ , by (21) this gradient is simply

$$\frac{dG}{dp} = \int_{t_0}^{t_1} (g_p + \lambda^T f_p) dt,\tag{35}$$

where  $g(t, y, p) = y_1 + p_2 y_2 y_3$ ,  $f$  is the vector valued function defining the right hand side of (33), and  $\lambda$  is the solution of the adjoint problem (20),

$$\begin{aligned}\dot{\lambda} &= -(f_y)^T \lambda - (g_y)^T \\ \lambda(t_1) &= 0.\end{aligned}\tag{36}$$

In order to avoid saving intermediate  $\lambda$  values just for the evaluation of the integral in (35), we extend the backward problem with the following  $N_p$  quadrature equations

$$\begin{aligned}\dot{\xi} &= g_p^T + f_p^T \lambda \\ \xi(t_1) &= 0,\end{aligned}\tag{37}$$

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 (34) can be obtained as  $G = -\zeta(t_0)$ , where  $\zeta$  is solution of the following quadrature equation:

$$\begin{aligned}\dot{\zeta} &= g \\ \zeta(t_1) &= 0.\end{aligned}\tag{38}$$

The source code for this example is listed in App. C.1. The main program and the user-defined routines are described below, with emphasis on the aspects particular to adjoint sensitivity calculations. The calling program must include the CVODES header file `cvodea.h` which in turn includes `cvodes.h` and thus provides CVODES function prototypes and constants, including those in the adjoint sensitivity module.

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 check points. 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. Notice that we employ two machine environment variables, `machEnvF` for the forward problem and `machEnvB` for the backward problem. The next code blocks allocate and initialize the user data structure with the values of the parameters  $p$ , initialize `machEnvF` by calling the serial machine environment initialization routine from `NVECTOR_SERIAL`, allocate and initialize `y` with the initial conditions of the forward problem, and finally set the tolerances `rtol` and `atol`.

The call to `CVodeMalloc` sets-up the forward integration and specifies the BDF integration method with `NEWTON` iteration. The linear solver is selected to be `CVDENSE` through the call to its initialization routine `CVDense`, with a non-NULL Jacobian routine `Jac`.

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 check points.

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 check points in the argument `ncheck`. A list of the check points is printed by `CVadjCheckPointsList`.

The next segment of code deals with the setup of the backward problem. First, a serial machine environment variable `machEnvB` is initialized for vectors of length `NEQ + NP + 1` (dimension of  $\lambda$  + dimension of  $\xi$  + one additional quadrature variable to evaluate  $G$ ). Following that, the program allocates space and initializes the variables of the backward problem and the relative and absolute tolerances for the backward integration. `CVODES` memory for the integration of the backward integration is allocated by the call to the interface routine `CVodeMallocB` which specifies the size of the problem, the right hand side user function `fB` and the BDF integration method with `NEWTON` iteration, among other things. The dense linear solver `CVDENSE` is then initialized by calling the `CVDenseB` interface routine with a non-NULL Jacobian routine `JacB`.

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 main program ends by printing the value of  $G$  and its gradient and by freeing previously allocated memory through calls to `CVodeFree` (for the `CVODES` memory for the forward problem), `CVadjFree` (for the memory allocated for the combined problem), `N_VFree` (for the various vectors), and `M_EnvFree_Serial` (for the two machine environment variables `machEnvF` and `machEnvB`).

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 (33), while `fB` and `JacB` are mere translations of the backward problem (36)–(37)–(38).

The output generated by `cvadx` is shown below.

```
Allocate CVODE memory for forward runs
```

```
Allocate global memory
```

Forward integration

List of Check Points (ncheck = 3)

```

Check point 3
  address  50e40
  t0       5210491.598010
  t1       40000000.000000
  next     50c98
Check point 2
  address  50c98
  t0       8078.421607
  t1       5210491.598010
  next     4fee0
Check point 1
  address  4fee0
  t0       66.985880
  t1       8078.421607
  next     4aa58
Check point 0
  address  4aa58
  t0       0.000000
  t1       66.985880
  next     0

```

Allocate CVODE memory for backward run

```

=====
G:          1.8219e+04
Gp:         -7.8383e+05   3.1991e+00  -5.3301e-04
=====
lambda(t0):   3.4249e+04   3.4206e+04   3.4139e+04
=====

```

Free memory

## 9.2 A Parallel Sample Program

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

$$\begin{aligned}
 \frac{\partial u}{\partial t} &= p_1 \frac{\partial^2 u}{\partial x^2} + p_2 \frac{\partial u}{\partial x} \\
 0 &= x_0 \leq x \leq x_1 = 2 \\
 0 &= t_0 \leq t \leq t_1 = 2.5,
 \end{aligned}
 \tag{39}$$

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 §2.3 gives

$$\frac{dg}{dp}(t_1) = \int_{t_0}^{t_1} dt \int_{x_0}^{x_1} dx \mu \cdot \left[ \frac{\partial^2 u}{\partial x^2}; \frac{\partial u}{\partial x} \right] \quad (40)$$

and

$$\delta g|_{t_1} = \int_{x_0}^{x_1} \mu(t_0, x) \delta u_0(x) dx, \quad (41)$$

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

$$\begin{aligned} \frac{\partial \mu}{\partial t} + p_1 \frac{\partial^2 \mu}{\partial x^2} - p_2 \frac{\partial \mu}{\partial x} &= 0 \\ \mu(t_1, x) &= 1 \\ \mu(t, x_0) = \mu(t, x_1) &= 0. \end{aligned} \quad (42)$$

Both the forward problem (39) and the backward problem (42) 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 (40) by introducing the additional equations

$$\begin{aligned} \dot{\xi}_1 &= \int_{x_0}^{x_1} dx \mu \frac{\partial^2 u}{\partial x^2}, \quad \xi_1(t_1) = 0, \\ \dot{\xi}_2 &= \int_{x_0}^{x_1} dx \mu \frac{\partial u}{\partial x}, \quad \xi_2(t_1) = 0, \end{aligned} \quad (43)$$

yielding

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

The space integrals in (41) and (43) 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. C.2. 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` (§9.1) with differences arising from the use of the parallel vector module.

Besides the parallelism implemented by CVODES at the vector kernel 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 (43). 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 (39) and (42) 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 (43), 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.

Sample output generated by `pvanx` is shown below.

(PE# 0)

```
mu(t0)[ 1] = 0.000277648
mu(t0)[ 2] = 0.000562281
mu(t0)[ 3] = 0.000847703
mu(t0)[ 4] = 0.00112702
mu(t0)[ 5] = 0.00139372
```

(PE# 1)

```
mu(t0)[ 6] = 0.00164049
mu(t0)[ 7] = 0.0018611
mu(t0)[ 8] = 0.0020485
mu(t0)[ 9] = 0.00219734
mu(t0)[10] = 0.00230152
```

(PE# 2)

```
mu(t0)[11] = 0.00235718
mu(t0)[12] = 0.00235987
mu(t0)[13] = 0.00230773
mu(t0)[14] = 0.00219852
mu(t0)[15] = 0.00203278
```

(PE# 3)

```
mu(t0)[16] = 0.00181094
mu(t0)[17] = 0.00153609
mu(t0)[18] = 0.00121155
```

```
mu(t0)[19] = 0.000842963  
mu(t0)[20] = 0.000436478
```

```
(PE# 4)  
g(t1) = 0.019903
```

```
(PE# 4)  
dgdp(t1) = [ -1.12076  -1.00896 ]
```



## 10 Types `realtype` and `integertype`

### 10.1 Description

The `sundialstypes.h` file contains the definitions of the types `realtype` and `integertype`. SUNDIALS solvers use the type `realtype` for all floating point data and the type `integertype` for all problem size-related data such as the actual problem size, the bandwidths in the BAND solver, and the integers stored in the length  $N$  pivot arrays in both the DENSE and BAND solvers. These types make it easy to solve problems of virtually any size using single or double precision arithmetic. The type `realtype` can be `double` or `float` and the type `integertype` can be `int` or `long int`. The default settings are `double` and `long int`.

### 10.2 Changing Type `realtype`

The user can change the precision of the SUNDIALS solvers arithmetic from double to single by changing the typedef `typedef double realtype;` to `typedef float realtype;` and by changing the definitions `#define SUNDIALS_FLOAT 0` to `#define SUNDIALS_FLOAT 1` and `#define SUNDIALS_DOUBLE 1` to `#define SUNDIALS_DOUBLE 0` in `sundialstypes.h`. These macro definitions are used to enable `sundialstypes.h` to branch on the setting of `realtype` at compile time.

Changing from double precision to single precision arithmetic also requires minor changes in the implementation file `sundialsmath.c` for the SUNDIALSMATH module which SUNDIALS solvers use. The `RPowerR` and `RSqrt` functions compute a real number raised to a real power and the square root of a number, respectively. The default implementation of these routines calls standard C math library functions which do double precision arithmetic. These implementations should be changed to call single precision routines which are available on the user's machine.

Within SUNDIALS, real constants are set by way of a macro called `RCONST`. It is this macro that needs the ability to branch on the setting for `realtype`. In ANSI C, a floating point constant with no suffix is stored as a `double`. Placing the suffix "F" at the end of a floating point constant makes it a `float`. For example,

```
#define A 1.0
#define B 1.0F
```

defines `A` to be a `double` constant 1.0 and `B` to be a `float` constant 1.0. The macro call `RCONST(1.0)` expands to 1.0 if `realtype` is `double` and it expands to 1.0F if `realtype` is `float`. SUNDIALS uses the `RCONST` macro for all its floating point constants.

A user program which uses the type `realtype` and the `RCONST` macro to handle floating point constants is precision-independent except for any calls to single or double precision standard math library functions. (Our demonstration programs use `realtype` but not `RCONST`.) Users can, however, use the type `double` or `float` in their code (assuming the typedef for `realtype` matches this choice). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the codes to use `realtype`.

### 10.3 Changing Type `integertype`

SUNDIALS uses the type `integertype` for all quantities related to problem size. On some machines the size of an `int` and a `long int` are the same, but this is not always the case. If `int` is sufficiently large on a given machine, and the user wishes to make `int` the `integertype` type, change

the typedef `typedef long int integertype;` to `typedef int integertype;` and the macro definitions `#define SUNDIALS_INT 0` to `#define SUNDIALS_INT 1` and `#define SUNDIALS_LONG_INT 1` to `#define SUNDIALS_LONG_INT 0` in `sundialtypes.h`. In terms of the problem size  $N$ , and the bandwidths `m1` and `mu` in the case of the band module `BAND`, the largest integer that must be accommodated by the `integertype` type is  $N + m1 + mu$  in the band case, and  $N$  in all other cases. The user can use the type `int` or `long int` in his/her code instead of `integertype` (assuming the typedef for `integertype` matches this choice).

## 11 Description of the NVECTOR Concept

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by a generic machine environment (of type `M_Env`).

The generic `M_Env` type is a pointer to a structure that has an implementation-dependent *content* field containing all data necessary to generate a new vector in that particular implementation, an *ops* field pointing to a structure with generic vector operations, and a *tag* field which is used in some compatibility tests within the SUNDIALS solvers. The type `M_Env` is defined as

```
typedef struct _generic_M_Env *M_Env;

struct _generic_M_Env {
    void *content;
    struct _generic_N_Vector_Ops *ops;
    char tag[8];
};
```

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector and a *menv* field pointing to the `M_Env` structure that was used in constructing the vector. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
    void *content;
    struct _generic_M_Env *menv;
};
```

The `_generic_N_Vector_Ops` structure is defined as

```
struct _generic_N_Vector_Ops {
    N_Vector      (*nvnew)(integertype, M_Env);
    N_Vector_S    (*nvnewS)(integertype, integertype, M_Env);
    void          (*nvfree)(N_Vector);
    void          (*nvfreeS)(integertype, N_Vector_S);
    N_Vector      (*nvmake)(integertype, realtype *, M_Env);
    void          (*nvdispose)(N_Vector);
    realtype*     (*nvgetdata)(N_Vector);
    void          (*nvsetdata)(realtype *, N_Vector);
    void          (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
    void          (*nvconst)(realtype, N_Vector);
    void          (*nvprod)(N_Vector, N_Vector, N_Vector);
    void          (*nvdiv)(N_Vector, N_Vector, N_Vector);
    void          (*nvscale)(realtype, N_Vector, N_Vector);
    void          (*nvabs)(N_Vector, N_Vector);
    void          (*nvinv)(N_Vector, N_Vector);
```

```

void      (*nvaddconst)(N_Vector, realtype, N_Vector);
realtype  (*nvdotprod)(N_Vector, N_Vector);
realtype  (*nvmaxnorm)(N_Vector);
realtype  (*nvwrmsnorm)(N_Vector, N_Vector);
realtype  (*nvmin)(N_Vector);
realtype  (*nvwl2norm)(N_Vector, N_Vector);
realtype  (*nvl1norm)(N_Vector);
void      (*nvonemask)(N_Vector);
void      (*nvcompare)(realtype, N_Vector, N_Vector);
booleantype (*nvinvtest)(N_Vector, N_Vector);
booleantype (*nvconstrprodpos)(N_Vector, N_Vector);
booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
realtype  (*nvminquotient)(N_Vector, N_Vector);
void      (*nvprint)(N_Vector);
};

```

In addition to the above type definitions, the generic NVECTOR module also defines and implements the vector kernels acting on `N_Vector`'s. These routines are nothing but wrappers around the vector kernels defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the structure pointed to by `M_Env`. To illustrate this point we show below the implementation of a typical vector kernel from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```

void N_VScale(realtype c, N_Vector x, N_Vector z)
{
    z->menv->ops->nvscale(c, x, z);
}

```

Table 5 contains a complete list of all vector operations defined by the generic NVECTOR module.

A particular implementation of the NVECTOR module must:

- specify the *content* fields of `M_Env` and `N_Vector`;
- define and implement the vector kernels. Note that the kernel routine names should be unique to that implementation in order to provide the option of using `N_Vector`'s with different internal representations in the same code;
- define and implement user-callable constructor and destructor routines to generate and free a variable of type `M_Env` with the new *content* field and with *ops* pointing to the new vector kernels.

We also strongly recommend that the developer of a new NVECTOR implementation provide as many accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 5: Description of the NVECTOR kernels

Name	Usage and Description
N_VNew	$v = \text{N\_VNew}(n, \text{machEnv});$ Returns a new N_Vector of length $n$ . If there is not enough memory for a new N_Vector, then N_VNew returns NULL.
N_VNew_S	$vs = \text{N\_VNew\_S}(ns, n, \text{machEnv});$ Returns an array of $ns$ new N_Vector's of length $n$ . The parameter <b>machEnv</b> is a pointer to machine environment specific information. If there is not enough memory for a new array of N_Vector's or for one of the components, then N_VNew_S returns NULL.
N_VFree	$\text{N\_VFree}(v);$ Frees the N_Vector $v$ . It is illegal to use $v$ after the call to N_VFree.
N_VFree_S	$\text{N\_VFree\_S}(ns, vs);$ Frees the array of $ns$ N_Vector's $vs$ . It is illegal to use $vs$ after the call to N_VFree_S.
N_VMake	$v = \text{N\_VMake}(n, \text{vdata}, \text{machEnv});$ Creates an N_Vector of length $n$ with component array data $\text{vdata}$ allocated by the user.
N_VDispose	$\text{N\_VDispose}(v);$ Destroys an N_Vector created by a previous call to N_VMake. It is the user's responsibility to free the memory allocated for the data array.
N_VGetData	$\text{vdata} = \text{N\_VGetData}(v);$ Returns a pointer to the data component array from the N_Vector $v$ .
N_VSetData	$\text{N\_VSetData}(\text{vdata}, v);$ Attaches the data component array $\text{vdata}$ to the N_Vector $v$ .
N_VLinearSum	$\text{N\_VLinearSum}(a, x, b, y, z);$ Performs the operation $z = ax + by$ , where $a$ and $b$ are scalars and $x$ and $y$ are N_Vector's: $z_i = ax_i + by_i, i = 0, 1, \dots, n-1$ .
N_VConst	$\text{N\_VConst}(c, z);$ Initializes all components of the N_Vector $z$ to $c$ : $z_i = c, i = 0, 1, \dots, n-1$ .
N_VProd	$\text{N\_VProd}(x, y, z);$ Sets the N_Vector $z$ to be the component-wise product of the N_Vector's $x$ and $y$ : $z_i = x_i y_i, i = 0, 1, \dots, n-1$ .
N_VDiv	$\text{N\_VDiv}(x, y, z);$ Sets the N_Vector $z$ to be the component-wise ratio of the N_Vector's $x$ and $y$ : $z_i = x_i / y_i, i = 0, 1, \dots, n-1$ . The $y_i$ may not be tested for 0 values.
<i>continued on next page</i>	

continued from last page

Name	Usage and Description
N_VScale	N_VScale(c, x, z); Scales the N_Vector x by the scalar c and returns the result in z: $z_i = cx_i$ , $i = 0, 1, \dots, n-1$ .
N_VAbs	N_VAbs(x, y); Sets the components of the N_Vector y to be the absolute values of the components of the N_Vector x: $y_i =  x_i $ , $i = 0, 1, \dots, n-1$ .
N_VInv	N_VInv(x, z); Sets the components of the N_Vector z to be the inverses of the components of the N_Vector x: $z_i = 1.0/x_i$ , $i = 0, 1, \dots, n-1$ . This routine may not check for division by 0. It should be called only with an x which is guaranteed to have all non-zero components.
N_VAddConst	N_VAddConst(x, b, z); Adds the scalar b to all components of x and returns the result in the N_Vector z: $z_i = x_i + b$ , $i = 0, 1, \dots, n-1$ .
N_VDotProd	d = N_VDotProd(x, y); Returns the value of the ordinary dot product of x and y: $d = \sum_{i=0}^{n-1} x_i y_i$ .
N_VMaxNorm	m = N_VMaxNorm(x); Returns the maximum norm of the N_Vector x: $m = \max_i  x_i $ .
N_VWrmsNorm	m = N_VWrmsNorm(x, w) Returns the weighted root mean square norm of the N_Vector x with weight vector w: $m = \sqrt{(\sum_{i=0}^{n-1} (x_i w_i)^2) / n}$ .
N_VMin	m = N_VMin(x); Returns the smallest element of the N_Vector x: $m = \min_i x_i$ .
N_VWL2Norm	m = N_VWL2Norm(x, w); Returns the weighted Euclidean $\ell_2$ norm of the N_Vector x with weight vector w: $m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}$ .
N_VL1Norm	m = N_VL1Norm(x); Returns the $\ell_1$ norm of the N_Vector x: $m = \sum_{i=0}^{n-1}  x_i $ .
N_VOneMask	N_VOneMask(x); Sets the non-zero components of the N_Vector to 1.0: $x_i = 1.0$ , if $x_i \neq 0.0$ , $i = 0, 1, \dots, n-1$ .
N_VCompare	N_VCompare(c, x, z); Compares the components of the N_Vector x to the scalar c and returns an N_Vector z such that: $z_i = 1.0$ if $ x_i  \geq c$ and $z_i = 0.0$ otherwise.
N_VInvTest	t = N_VInvTest(x, z); Sets the components of the N_Vector z to be the inverses of the components of the N_Vector x: $z_i = 1.0/x_i$ , $i = 0, 1, \dots, n-1$ . This routine returns TRUE if all components of x are non-zero (successful inversion) and returns FALSE otherwise.

continued on next page

*continued from last page*

Name	Usage and Description
N_VConstrProdPos	<code>t = N_VConstrProdPos(c, x);</code> Returns a boolean equal to <b>FALSE</b> if, for some $i = 0, 1, \dots, n-1$ , $c_i \neq 0.0$ and $x_i c_i \leq 0.0$ , and <b>TRUE</b> otherwise. This routine is used for constraint checking.
N_VConstrMask	<code>t = N_VConstrMask(c, x, m);</code> Performs the following constraint tests: $x_i > 0$ if $c_i = 2$ , $x_i \geq 0$ if $c_i = 1$ , $x_i \leq 0$ if $c_i = -1$ , $x_i < 0$ if $c_i = -2$ . This routine returns <b>FALSE</b> if any element failed the constraint test, <b>TRUE</b> if all passed. It also sets a mask vector <code>m</code> , with elements equal to 1.0 where the corresponding constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
N_VMinQuotient	<code>minq = N_VMinQuotient(num, denom);</code> This routine returns the minimum of the quotients obtained by term-wise dividing <code>num[i]</code> by <code>denom[i]</code> . A zero element in <code>denom</code> will be skipped. If no such quotients are found, then the large value $10^{99}$ is returned.

### 11.1 The NVECTOR\_SERIAL Implementation of NVECTOR

The NVECTOR\_SERIAL implementation of the NVECTOR module defines the *content* field of `M_Env` to be a structure containing the length of the vector:

```
struct _M_EnvSerialContent {
    integertype length;
};
```

The *tag* field of `M_Env` is set to `serial`. The *content* field of `N_Vector` is defined to be a structure containing the length of the vector and a pointer to the beginning of a contiguous data array:

```
struct _N_VectorSerialContent {
    integertype length;
    realtype    *data;
};
```

The NVECTOR\_SERIAL implementation provides user-callable routines `M_EnvInit_Serial` to create a structure of type `M_Env` whose *content* field is of type `struct _M_EnvSerialContent`, and `M_EnvFree_Serial` to deallocate the space used by such a structure.

The form of the call to `M_EnvInit_Serial` is

```
machenv = M_EnvInit_Serial(vec_length);
```

If successful, `M_EnvInit_Serial` returns a pointer of type `M_Env`. This pointer should in turn be passed in any user calls to `N_VNew` to create a new `N_Vector` of this type. A machine environment variable `machenv` returned by `M_EnvInit_Serial` can be freed by calling:

```
M_EnvFree_Serial(machenv);
```

In addition to these two routines, `NVECTOR_SERIAL` defines serial implementations of all vector kernels listed in Table 5, as well as the following macros that can be used to access the contents of `M_Env` and `N_Vector` or to create and destroy `N_Vector`'s with component array data allocated by the user. The suffix `_S` in the names denotes serial version.

- `ME_CONTENT_S`, `NV_CONTENT_S`

These macros give access to the contents of the serial machine environment and `N_Vector`, respectively.

The assignment `m_cont = ME_CONTENT_S(machenv)` sets `m_cont` to be a pointer to the serial machine environment content structure (of type `struct M_EnvSerialContent`).

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure of type `struct N_VectorSerialContent`.

- `NV_DATA_S`, `NV_LENGTH_S`

These macros give individual access to the parts of the content of a serial `N_Vector`.

The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of `v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

- `NV_Ith_S`

This macro gives access to the individual components of the data array of an `N_Vector`.

The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

- `NV_MAKE_S`, `NV_DISPOSE_S`

These companion macros are used to create and destroy an `N_Vector` with a component array `vdata` allocated by the user.

The call `NV_MAKE_S(v,v_data,machenv)` makes `v` an `N_Vector` with component array `v_data`. The length of the array is taken from `machenv`. `NV_MAKE_S` stores the pointer `v_data` so that changes made by the user to the elements of `v_data` are simultaneously reflected in `v`. There is no copying of elements.

The call `NV_DISPOSE_S(v)` frees all memory associated with `v` except for its component array. This memory was allocated by the user and, therefore, should be deallocated by the user.

- `NVS_MAKE_S`, `NVS_DISPOSE_S`

These companion macros are used to create and destroy an array of `N_Vector`'s with component `vs_data` (of type `realtype **`) allocated by the user.

The call `NVS_MAKE_S(vs,vs_data,ns,machenv)` makes `vs` an array of `ns` `N_Vector`'s, with `vs[i]` having component array `vs_data[i]` and length taken from `machenv`. `NVS_MAKE_S` stores the pointers `vs_data[i]` so that changes made by the user to the elements of `vs_data` are simultaneously reflected in `vs`. There is no copying of elements.



The call `NVS_DISPOSE_S(vs,ns)` frees all memory associated with `vs` except for its components' component array. This memory was allocated by the user and, therefore, should be deallocated by the user.

## Notes

- Users who use the make/dispose macros must `#include<stdlib.h>` since these macros expand to calls to `malloc` and `free`.
- When looping over the components of an `N_Vector` `v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.

`NV_MAKE_S` and `NV_DISPOSE_S` are similar to the `N_VNew` and `N_VFree` implemented by `NVECTOR_SERIAL`, while `NVS_MAKE_S` and `NVS_DISPOSE_S` are similar to `N_VNew_S` and `N_VFree_S`. The difference is one of responsibility for component memory allocation and deallocation. `N_VNew` allocates memory for the `N_Vector` components and `N_VFree` frees the component memory allocated by `N_VNew`. For `NV_MAKE_S` and `NV_DISPOSE_S`, the component memory is allocated and freed by the user of this package. Similar remarks hold for `NVS_MAKE_S`, `NVS_DISPOSE_S` and `N_VNew_S`, `N_VFree_S`.

- To maximize efficiency, vector kernels in the `NVECTOR_SERIAL` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the `M_Env` structure returned by `M_EnvInit.Serial`.

## 11.2 The NVECTOR\_PARALLEL Implementation of NVECTOR

The `NVECTOR_PARALLEL` implementation of the `NVECTOR` module defines the *content* field of `M_Env` to be a structure containing the local and global lengths of the vector, a pointer to the MPI communicator, and a flag showing if the user called `MPI_Init`:

```
struct _M_EnvParallelContent {
    MPI_Comm    comm;
    integertype local_vec_length;
    integertype global_vec_length;
    int         init_by_user;
};
```

The *tag* field of `M_Env` is set to `parallel`. The *content* field of `N_Vector` is defined to be a structure containing the global and local lengths of the vector and a pointer to the beginning of a contiguous local data array:

```
struct _N_VectorParallelContent {
    integertype local_length;
    integertype global_length;
    realtype    *data;
};
```

The NVECTOR\_PARALLEL implementation provides user-callable routines `M_EnvInit_Parallel` to create a structure of type `M_Env` whose *content* field is of type `struct _M_EnvParallelContent`, and `M_EnvFree_Parallel` to deallocate the space used by such a structure.

The form of the call to `M_EnvInit_Parallel` is

```
machenv = M_EnvInit_Parallel(comm, local_vec_length, global_vec_length,
                             argc, argv);
```

Its arguments are:

- `comm` is a pointer to the MPI communicator, of type `MPI_Comm`. Must be non-NULL;
- `local_vec_length` is the length of the piece of the vectors residing on this processor. If the active processor set is a proper subset of the full processor set assigned to the job, the value of `local_vec_length` should be 0 on the inactive processors. Otherwise, the two global length values, input and computed, may differ;
- `global_vec_length` is the global length of the vectors. This must equal the sum of all local lengths over the active processor set. If not, a message is printed;
- `argc`, `argv` are the command line arguments count and the command line argument character array from the main program, respectively. Dummy arguments are acceptable if `MPI_Init` has already been called.

If successful, `M_EnvInit_Parallel` returns a pointer of type `M_Env`. This pointer should in turn be passed in any user calls to `N_VNew` to create a new `N_Vector` of this type. A machine environment variable `machenv` returned by `M_EnvInit_Parallel` can be freed by calling:

```
M_EnvFree_Parallel(machenv);
```

In addition to these two routines, NVECTOR\_PARALLEL defines MPI implementations of all vector kernels listed in Table 5, as well as the following macros that can be used to access the contents of `M_Env` and `N_Vector` or to create and destroy `N_Vector`'s with component array data allocated by the user. The suffix `_P` in the names denotes parallel version.

- `ME_CONTENT_P`, `NV_CONTENT_P`

These macros give access to the contents of the parallel machine environment and `N_Vector`, respectively.

The assignment `m_cont = ME_CONTENT_P(machenv)` sets `m_cont` to be a pointer to the parallel machine environment content structure (of type `struct _M_EnvParallelContent`).

The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` content structure of type `struct _N_VectorParallelContent`.

- `NV_DATA_P`, `NV_LOCLENGTH_P`, `NV_GLOBLENGTH_P`

These macros give individual access to the parts of the content of a parallel `N_Vector`.

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the vector `v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

- **NV\_Ith\_P**

This macro gives access to the individual components of the local data array of an **N\_Vector**.

The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`. The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

- **NV\_MAKE\_P, NV\_DISPOSE\_P**

These companion macros are used to create and destroy an **N\_Vector** with a component array `vdata` allocated by the user.

The call `NV_MAKE_P(v,v_data,machenv)` makes `v` an **N\_Vector** with local component array `v_data`. The local and global lengths of the vector `v` is taken from `machenv`. `NV_MAKE_P` stores the pointer `v_data` so that changes made by the user to the elements of `v_data` are simultaneously reflected in `v`. There is no copying of elements.

The call `NV_DISPOSE_P(v)` frees all memory associated with `v` except for its component array. This memory was allocated by the user and, therefore, should be deallocated by the user.

- **NVS\_MAKE\_P, NVS\_DISPOSE\_P**

These companion macros are used to create and destroy an array of **N\_Vector**'s with component `vs_data` (of type `realtype **`) allocated by the user.

The call `NVS_MAKE_P(vs,vs_data,ns,machenv)` makes `vs` an array of `ns` **N\_Vector**'s, with `vs[i]` having local component array `vs_data[i]` and local and global lengths taken from `machenv`. `NVS_MAKE_P` stores the pointers `vs_data[i]` so that changes made by the user to the elements of `vs_data` are simultaneously reflected in `vs`. There is no copying of elements.

The call `NVS_DISPOSE_P(vs,ns)` frees all memory associated with `vs` except for its components' component array. This memory was allocated by the user and, therefore, should be deallocated by the user.

## Notes

- Users who use the make/dispose macros must `#include<stdlib.h>` since these macros expand to calls to `malloc` and `free`.
- When looping over the components of an **N\_Vector** `v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.

`NV_MAKE_P` and `NV_DISPOSE_P` are similar to `N_VNew` and `N_VFree` implemented by `NVECTOR_PARALLEL`, while `NVS_MAKE_P` and `NVS_DISPOSE_P` are similar to `N_VNew_S` and `N_VFree_S`. The difference is one of responsibility for component memory allocation and deallocation. `N_VNew` allocates memory for the **N\_Vector** components and `N_VFree` frees the component

memory allocated by `N_VNew`. For `NV_MAKE_P` and `NV_DISPOSE_P`, the component memory is allocated and freed by the user of this package. Similar remarks hold for `NVS_MAKE_P`, `NVS_DISPOSE_P` and `N_VNew_S`, `N_VFree_S`.

- To maximize efficiency, vector kernels in the `NVECTOR_PARALLEL` implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the `M_Env` structure returned by `M_EnvInitParallel`.

### 11.3 NVECTOR Kernels Used by CVODES

In Table 6 below, we list the vector kernels in the `NVECTOR` module within the `CVODES` package. The table also shows, for each kernel, which of the code modules uses the kernel. The `CVODES` column shows kernel usage within the main integrator module, while the remaining four columns show kernel usage within each of the four `CVODES` linear solvers.

There is one subtlety in the `CVSPGMR` column hidden by the table. The dot product kernel `N_VDotProd` is not called within the implementation file `cvsspgmr.c` for the `CVSPGMR` solver, yet we have marked it as “used” by `CVSPGMR`. This is because `N_VDotProd` is called within the implementation files `spgmr.c` and `iterative.c` for the generic `SPGMR` solver upon which the `CVSPGMR` solver is implemented. This issue does not arise for the other three `CVODES` linear solvers because the generic `DENSE` and `BAND` solvers (used in the implementation of `CVDENSE` and `CVBAND`) do not make calls to any vector kernels and `CVDIAG` is not implemented using a generic diagonal solver.

The vector kernels `N_VMake`, `N_VDispose`, `N_VGetData`, and `N_VSetData` are only called by one of the `CVODES` direct linear solvers, `CVDENSE` or `CVBAND`, or by a preconditioner solve routine that uses a direct linear solver (such as `CVBANDPRE` or `CVBBDPRE`). They duplicate functionality provided by macros in particular implementations of `NVECTOR`, but are part of the specifications of the generic `NVECTOR` module to insure that the `CVODES` package is not dependent on any particular `NVECTOR` implementation.

At this point, we should emphasize that the `CVODES` user does not need to know anything about the usage of vector kernels by the `CVODES` code modules in order to use `CVODES`. The information is presented as implementation details for the interested reader.

The vector kernels listed in Table 5 that are *not* used by `CVODES` are: `N_VWL2Norm`, `N_VL1Norm`, `N_VOneMask`, `N_VConstrProdPos`, `N_VConstrMask`, and `N_VMinQuotient`. Therefore a user-supplied `NVECTOR` module for `CVODES` could omit these six kernels.

Table 6: List of vector kernels usage by CVODES code modules

Kernel	CVODES	CVDENSE	CVBAND	CVDIAG	CVSPGMR
N_VNew	X			X	X
N_VFree	X			X	X
N_VNew_S	X				
N_VFree_S	X				
N_VMake		X	X		
N_VDispose		X	X		
N_VGetData		X	X		
N_VSetData		X	X		
N_VLinearSum	X	X		X	X
N_VConst	X				X
N_VProd	X			X	X
N_VDiv	X			X	X
N_VScale	X	X	X	X	X
N_VAbs	X				
N_VInv	X			X	
N_VAddConst	X			X	
N_VDotProd					X
N_VMaxNorm	X				
N_VWrmsNorm	X	X	X		X
N_VMin	X				
N_VCompare				X	
N_VInvTest				X	

## 12 Providing Alternate Linear Solver Modules

The central CVODES module interfaces with the linear solver module to be used by way of calls to five routines. These are denoted here by `linit`, `lsetup`, `lsolve`, `lsolveS`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize and allocate memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lsolveS`: solve the linear system;
- `lfree`: free the linear solver memory.

The `lsolveS` routine is intended only for use during forward sensitivity analysis for the solution of linear systems coming from the sensitivity systems, and need not be implemented if CVODES will not be used for forward sensitivity analysis.

A linear solver module must also provide a user-callable specification routine (like those described in §4.3.2) which will attach the above five routines to the main CVODES memory block. The return value of the specification routine should be: `SUCCESS = 0` if the routine was successful, `LMEM_FAIL = -1` if a memory allocation failed, or `LIN_ILL_INPUT = -2` if some input was illegal.

These five routines that interface between CVODES and the linear solver module necessarily have fixed call sequences. Thus a user wishing to implement another linear solver within the CVODES package must adhere to this set of interfaces. The following is a complete description of the call list for each of these routines. Note that the call list of each routine includes a pointer to the main CVODES memory block, by which the routine can access various data related to the CVODES solution. The contents of this memory block are given in the file `cvodes.h` (but not reproduced here, for the sake of space).

**Initialization routine.** The type definition of `linit` is

```
int (*cv_linit)(CNodeMem cv_mem);
```

The purpose of `linit` is to complete initializations for specific linear solver, such as counters and statistics. An `linit` function should return `LINIT_OK = 0` if it has successfully initialized the CVODES linear solver and `LINIT_ERR = -1` otherwise. If an error does occur, an appropriate message should be sent to `cv_mem->errfp`.

**Setup routine.** The type definition of `lsetup` is

```
int (*cv_lsetup)(CNodeMem cv_mem, int convfail, N_Vector ypred,  
                N_Vector fpred, booleantype *jcurPtr, N_Vector vtemp1,  
                N_Vector vtemp2, N_Vector vtemp3);
```

The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`. It may re-compute Jacobian-related data if it deems necessary. Its parameters are as follows:

- **cv\_mem**: problem memory pointer of type **CNodeMem**.
- **convfail**: a flag to indicate any problem that occurred during the solution of the nonlinear equation on the current time step for which the linear solver is being used. This flag can be used to help decide whether the Jacobian data kept by a CVODES linear solver needs to be updated or not. Its possible values are:
  - **NO\_FAILURES**: this value is passed to **lsetup** if either this is the first call for this step, or the local error test failed on the previous attempt at this step (but the Newton iteration converged).
  - **FAIL\_BAD\_J**: this value is passed to **lsetup** if (a) the previous Newton corrector iteration did not converge and the linear solver's setup routine indicated that its Jacobian-related data is not current, or (b) during the previous Newton corrector iteration, the linear solver's solve routine failed in a recoverable manner and the linear solver's setup routine indicated that its Jacobian-related data is not current.
  - **FAIL\_OTHER**: this value is passed to **lsetup** if during the current internal step try, the previous Newton iteration failed to converge even though the linear solver was using current Jacobian-related data.
- **ypred**: the predicted **y** vector for the current CVODES internal step.
- **fpred**: the value of the right-hand side at **ypred**, i.e.  $f(t_n, y_{pred})$ .
- **jcurPtr**: a pointer to a boolean to be filled in by **lsetup**. The function should set **\*jcurPtr** = **TRUE** if its Jacobian data is current after the call and should set **\*jcurPtr** = **FALSE** if its Jacobian data is not current. If **lsetup** calls for re-evaluation of Jacobian data (based on **convfail** and CVODES state data), it should return **\*jcurPtr** = **TRUE** unconditionally; otherwise an infinite loop can result.
- **vtemp1**, **vtemp2**, **vtemp3**: temporary variables of type **N\_Vector** provided for use by **lsetup**.

The **lsetup** routine should return 0 if successful, a positive value for a recoverable error, and a negative value for an unrecoverable error.

**Solve routine.** The type definition of **lsolve** is

```
int (*cv_lsolve)(CNodeMem cv_mem, N_Vector b, N_Vector ycur, N_Vector fcur);
```

The routine **lsolve** must solve the linear equation  $Mx = b$ , where  $M$  is some approximation to  $I - \gamma J$ ,  $J = (\partial f / \partial y)(t_n, y_{cur})$  and the right-hand side vector **b** is input. The vector **ycur** contains the solver's current approximation to  $y(t_n)$  and the vector **fcur** contains  $f(t_n, y_{cur})$ . The solution is to be returned in the vector **b**. **lsolve** returns a positive value for a recoverable error and a negative value for an unrecoverable error. Success is indicated by a 0 return value.

**Sensitivity solve routine.** The type definition of **lsolve** is

```
int (*cv_lsolveS)(CNodeMem cv_mem, N_Vector b, N_Vector ycur,
                  N_Vector fcur, integertype iS);
```

The routine `lsolveS` must solve the linear system  $Mx = b$  corresponding to the `iS`-th sensitivity system. This routine is typically identical to `lsolve` (except for the additional argument `iS`). Of the four linear solvers modules provided with CVODES, only the CVSPGMR module has an `lsolveS` routine that takes into account `iS` to select the correct scaling vectors for left or right preconditioning.

**Memory deallocation routine.** The type definition of `lfree` is

```
void (*cv_lfree)(CVodeMem cv_mem);
```

The routine `lfree` should free up any memory allocated by the linear solver. This routine is called once a problem has been completed and the linear solver is no longer needed.



## 13 Generic Linear Solvers in SUNDIALS

In this section, we describe three generic linear solver code modules that are included in CVODES, but which are of potential use as generic packages in themselves, either in conjunction with the use of CVODES or separately. These modules are:

- The DENSE matrix package, which includes the matrix type **DenseMat**, macros and functions for **DenseMat** matrices, and functions for small dense matrices treated as simple array types.
- The BAND matrix package, which includes the matrix type **BandMat**, macros and functions for **BandMat** matrices, and functions for small band matrices treated as simple array types.
- The SPGMR package, which includes a solver for the scaled preconditioned GMRES method.

For the sake of space, the functions for **DenseMat** and **BandMat** matrices and the functions in SPGMR are only summarized briefly, since they are less likely to be of direct use in connection with CVODES. The functions for small dense matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of CVODE and the CVSPGMR solver.

### 13.1 The DENSE Module

**Type DenseMat.** The type **DenseMat** is defined to be a pointer to a structure with a size and a data field:

```
typedef struct {
    integertype size;
    realtype **data;
} *DenseMat;
```

The *size* field indicates the number of columns (which is the same as the number of rows) of a dense matrix, while the *data* field is a two dimensional array used for component storage. The elements of a dense matrix are stored columnwise (i.e columns are stored one on top of the other in memory). If **A** is of type **DenseMat**, then the (i,j)-th element of **A** (with  $0 \leq i, j \leq \text{size}-1$ ) is given by the expression `(A->data)[j][i]` or by the expression `(A->data)[0][j*size+i]`. The macros below allow a user to access efficiently individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the j-th column of elements can be obtained via the **DENSE\_COL** macro. Users should use these macros whenever possible.

**Accessor Macros.** The following two macros are defined by the DENSE module to provide access to data in the **DenseMat** type:

- **DENSE\_ELEM**

Usage : `DENSE_ELEM(A,i,j) = a_ij`; or `a_ij = DENSE_ELEM(A,i,j)`;

**DENSE\_ELEM** references the (i,j)-th element of the  $N \times N$  **DenseMat** **A**,  $0 \leq i, j \leq N - 1$ .

- **DENSE\_COL**

Usage : `col_j = DENSE_COL(A,j);`

**DENSE\_COL** references the  $j$ -th column of the  $N \times N$  **DenseMat** **A**,  $0 \leq j \leq N - 1$ . The type of the expression **DENSE\_COL(A,j)** is **realtype \***. After the assignment in the usage above, `col_j` may be treated as an array indexed from 0 to  $N - 1$ . The  $(i, j)$ -th element of **A** is referenced by `col_j[i]`.

**Functions.** The following functions for **DenseMat** matrices are available in the **DENSE** package. For full details, see the header file **dense.h**.

- **DenseAllocMat**: allocation of a **DenseMat** matrix;
- **DenseAllocPiv**: allocation of a pivot array for use with **DenseFactor/DenseBacksolve**;
- **DenseFactor**: LU factorization with partial pivoting;
- **DenseBacksolve**: solution of  $Ax = b$  using LU factorization;
- **DenseZero**: load a matrix with zeros;
- **DenseCopy**: copy one matrix to another;
- **DenseScale**: scale a matrix by a scalar;
- **DenseAddI**: increment a matrix by the identity matrix;
- **DenseFreeMat**: free memory for a **DenseMat** matrix;
- **DenseFreePiv**: free memory for a pivot array;
- **DensePrint**: print a **DenseMat** matrix to standard output.

**Small Dense Matrix Functions.** The following functions for small dense matrices are available in the **DENSE** package:

- **denalloc**

**denalloc(n)** allocates storage for an  $n$  by  $n$  dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then **denalloc** returns **NULL**. The underlying type of the dense matrix returned is **realtype\*\***. If we allocate a dense matrix **realtype\*\* a** by `a = denalloc(n)`, then `a[j][i]` references the  $(i,j)$ -th element of the matrix **a**,  $0 \leq i, j \leq n-1$ , and `a[j]` is a pointer to the first element in the  $j$ -th column of **a**. The location `a[0]` contains a pointer to  $n^2$  contiguous locations which contain the elements of **a**;

- **denallocpiv**

**denallocpiv(n)** allocates an array of  $n$  integers. It returns a pointer to the first element in the array if successful. It returns **NULL** if the memory request could not be satisfied;

- **gefa**

**gefa(a,n,p)** factors the  $n$  by  $n$  dense matrix **a**. It overwrites the elements of **a** with its LU factors and keeps track of the pivot rows chosen in the pivot array **p**.

A successful LU factorization leaves the matrix **a** and the pivot array **p** with the following information:

1. **p[k]** contains the row number of the pivot element chosen at the beginning of elimination step **k**,  $k = 0, 1, \dots, n-1$ .
2. If the unique LU factorization of **a** is given by  $Pa = LU$ , where  $P$  is a permutation matrix,  $L$  is a lower triangular matrix with all 1's on the diagonal, and  $U$  is an upper triangular matrix, then the upper triangular part of **a** (including its diagonal) contains  $U$  and the strictly lower triangular part of **a** contains the multipliers,  $I - L$ ;

**gefa** returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization. In this case it returns the column index (numbered from one) at which it encountered the zero;

- **gesl**

**gesl(a,n,p,b)** solves the  $n$  by  $n$  linear system  $ax = b$ . It assumes that **a** has been LU factored and the pivot array **p** has been set by a successful call to **gefa(a,n,p)**. The solution  $x$  is written into the **b** array;

- **denzero**

**denzero(a,n)** sets all the elements of the  $n$  by  $n$  dense matrix **a** to be 0.0;

- **dencopy**

**dencopy(a,b,n)** copies the  $n$  by  $n$  dense matrix **a** into the  $n$  by  $n$  dense matrix **b**;

- **denscale**

**denscale(c,a,n)** scales every element in the  $n$  by  $n$  dense matrix **a** by **c**;

- **denaddI**

**denaddI(a,n)** increments the  $n$  by  $n$  dense matrix **a** by the identity matrix;

- **denfreepiv**

**denfreepiv(p)** frees the pivot array **p** allocated by **denallocpiv**;

- **denfree**

**denfree(a)** frees the dense matrix **a** allocated by **denalloc**;

- **denprint**

**denprint(a,n)** prints the  $n$  by  $n$  dense matrix **a** to standard output as it would normally appear on paper. It is intended as a debugging tool with small values of **n**. The elements are printed using the **%g** option. A blank line is printed before and after the matrix.

## 13.2 The BAND Module

**Type BandMat.** The type **BandMat** is the type of a large band matrix A (possibly distributed). It is defined to be a pointer to a structure defined by:

```
typedef struct {
    integertype size;
    integertype mu, ml, smu;
    realtype **data;
} *BandMat;
```

The fields in the above structure are:

- *size* is the number of columns (which is the same as the number of rows);
- *mu* is the upper half-bandwidth,  $0 \leq mu \leq size-1$ ;
- *ml* is the lower half-bandwidth,  $0 \leq ml \leq size-1$ ;
- *smu* is the storage upper half-bandwidth,  $mu \leq smu \leq size-1$ . The **BandFactor** routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper half-bandwidth as big as  $\min(size-1, mu+ml)$  because of partial pivoting. The *smu* field holds the upper half-bandwidth allocated for A.
- *data* is a two dimensional array used for component storage. The elements of a band matrix of type **BandMat** are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored.

If we number rows and columns in the band matrix starting from 0, then

- *data[0]* is a pointer to  $(smu+ml+1)*size$  contiguous locations which hold the elements within the band of A
- *data[j]* is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from  $smu-mu$  (to access the uppermost element within the band in the j-th column) to  $smu+ml$  (to access the lowest element within the band in the j-th column). Indices from 0 to  $smu-mu-1$  give access to extra storage elements required by **BandFactor**.
- *data[j][i-j+smu]* is the (i,j)-th element,  $j-mu \leq i \leq j+ml$ .

The macros below allow a user to access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer into the j-th column of elements can be obtained via the **BAND\_COL** macro. Users should use these macros whenever possible.

See Figure 4 for a diagram of the **BandMat** type.

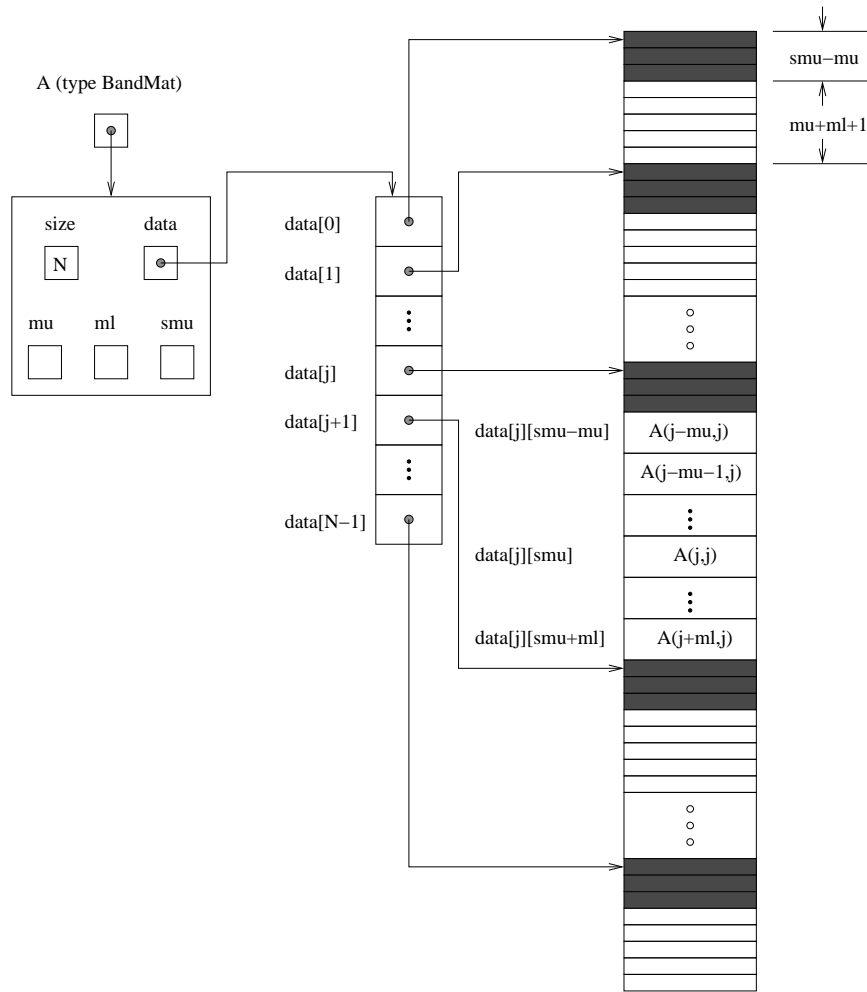


Figure 4: Diagram of the storage for a band matrix of type **BandMat**. Here  $A$  is an  $N \times N$  band matrix of type **BandMat** with upper and lower half-bandwidths  $\mu$  and  $\text{ml}$ , respectively. The rows and columns of  $A$  are numbered from 0 to  $N - 1$  and the  $(i, j)$ -th element of  $A$  is denoted  $A(i, j)$ . The greyed out areas of the underlying component storage are used by the **BandFactor** and **BandBacksolve** routines.

**Accessor Macros.** The following three macros are defined by the BAND module to provide access to data in the `BandMat` type:

- **BAND\_ELEM**

Usage : `BAND_ELEM(A,i,j) = a_ij`; or `a_ij = BAND_ELEM(A,i,j)`;

`BAND_ELEM` references the  $(i,j)$ -th element of the  $N \times N$  band matrix `A`, where  $0 \leq i, j \leq N - 1$ . The location  $(i,j)$  should further satisfy  $j - (A \rightarrow \text{mu}) \leq i \leq j + (A \rightarrow \text{ml})$ .

- **BAND\_COL**

Usage : `col_j = BAND_COL(A,j)`;

`BAND_COL` references the diagonal element of the  $j$ -th column of the  $N \times N$  band matrix `A`,  $0 \leq j \leq N - 1$ . The type of the expression `BAND_COL(A,j)` is `realtype *`. The pointer returned by the call `BAND_COL(A,j)` can be treated as an array which is indexed from  $-(A \rightarrow \text{mu})$  to  $(A \rightarrow \text{ml})$ .

- **BAND\_COL\_ELEM**

Usage : `BAND_COL_ELEM(col_j,i,j) = a_ij`; or `a_ij = BAND_COL_ELEM(col_j,i,j)`;

This macro references the  $(i,j)$ -th entry of the band matrix `A` when used in conjunction with `BAND_COL` to reference the  $j$ -th column through `col_j`. The index  $(i,j)$  should satisfy  $j - (A \rightarrow \text{mu}) \leq i \leq j + (A \rightarrow \text{ml})$ .

**Functions.** The following functions for `BandMat` matrices are available in the BAND package. For full details, see the header file `band.h`.

- `BandAllocMat`: allocation of a `BandMat` matrix;
- `BandAllocPiv`: allocation of a pivot array for use with `BandFactor`/`BandBacksolve`;
- `BandFactor`: LU factorization with partial pivoting;
- `BandBacksolve`: solution of  $Ax = b$  using LU factorization;
- `BandZero`: load a matrix with zeros;
- `BandCopy`: copy one matrix to another;
- `BandScale`: scale a matrix by a scalar;
- `BandAddI`: increment a matrix by the identity matrix;
- `BandFreeMat`: free memory for a `BandMat` matrix;
- `BandFreePiv`: free memory for a pivot array;
- `BandPrint`: print a `BandMat` matrix to standard output.

### 13.3 The SPGMR Module

The SPGMR package, in the files `spgmr.h` and `spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, `iterativ.h` and `iterativ.c`, contains auxiliary functions that support SPGMR, and also other Krylov solvers to be added later. For full details, including usage instructions, see the files `spgmr.h` and `iterativ.h`.

**Functions.** The following functions are available in the SPGMR package:

- `SpgmrMalloc`: allocation of memory for `SpgmrSolve`;
- `SpgmrSolve`: solution of  $Ax = b$  by the SPGMR method;
- `SpgmrFree`: free memory allocated by `SpgmrMalloc`.

The following functions are available in the support package `iterativ.h` and `iterativ.c`:

- `ModifiedGS`: performs modified Gram-Schmidt procedure;
- `ClassicalGS`: performs classical Gram-Schmidt procedure;
- `QRfact`: performs QR factorization of Hessenberg matrix;
- `QRsol`: solves a least squares problem with a Hessenberg matrix factored by `QRfact`.

## References

- [1] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, *VODE, a Variable-Coefficient ODE Solver*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1038–1051.
- [2] P. N. Brown and A. C. Hindmarsh, *Reduced Storage Matrix Methods in Stiff ODE Systems*, J. Appl. Math. & Comp. 31 (1989), pp. 40–91.
- [3] G. D. Byrne, *Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting*, in *Computational Ordinary Differential Equations*, J. R. Cash and I. Gladwell (Eds.), Oxford University Press, Oxford, 1992, pp. 323–356.
- [4] G. D. Byrne and A. C. Hindmarsh, *PVODE, An ODE Solver for Parallel Computers*, Intl. J. High Perf. Comput. Apps., 1999, 13(4), pp. 254–365.
- [5] Y. Cao, S. Li, L. R. Petzold, and R. Serban, *Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE and its Numerical Solution*, SIAM J. Sci. Comp., to appear.
- [6] M. Carcotsios and W. E. Stewart, *Sensitivity Analysis of Initial Value Problems with Mixed ODEs and Algebraic Constraints*, Comput. Chem. Eng., 1985, 9, pp. 359–365.
- [7] S. D. Cohen and A. C. Hindmarsh, *CVODE User Guide*, LLNL Report UCRL-MA-118618, Sept. 1994
- [8] S. D. Cohen and A. C. Hindmarsh, *CVODE, a Stiff/Nonstiff ODE Solver in C*, Computers in Physics. 10(2) (1996), pp. 138–143.
- [9] W. F. Feehery, J. E. Tolsma, and P. I. Barton, *Efficient Sensitivity Analysis of Large-Scale Differential-Algebraic Systems*, Appl. Num. Math., 1997, 25, pp. 41–54.
- [10] A. C. Hindmarsh, *Detecting Stability Barriers in BDF Solvers*, in *Computational Ordinary Differential Equations*, J. R. Cash and I. Gladwell (Eds.), Oxford Univ. Press, 1992, pp. 87–96. Also available as LLNL Report UCRL-101197, June 1989.
- [11] A. C. Hindmarsh, *Avoiding BDF Stability Barriers in the MOL Solution of Advection-Dominated Problems*, Appl. Num. Math., 1995, 17, pp. 311–318.
- [12] A. C. Hindmarsh and A. G. Taylor, *PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems*, LLNL Report UCRL-ID-129739, February 1998.
- [13] T. Maly and L. R. Petzold, *Numerical Methods and Software for Sensitivity Analysis of Differential-Algebraic Systems*, Appl. Numer. Math., 1996, 20, pp. 57–79.
- [14] K. Radhakrishnan and A. C. Hindmarsh, *Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*, NASA Reference Publication 1327, 1993, and LLNL Report UCRL-ID-113855, March 1994.
- [15] Y. Saad and M. H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comp. 7 (1986), pp. 856–869.



# Index

## A

ADAMS ..... 22, 30, 31  
 Adams method ..... 4  
 adjoint sensitivity analysis  
   check-pointing ..... 11  
   examples *see* examples, adjoint sensitivity  
   implementation in CVODES ..... 12, 15  
   mathematical background ..... 10–12  
   right hand side evaluation ..... 57–58  
 ALLSENS ..... 46, 50

## B

BAD\_DKY ..... 31, 48, 49  
 BAD\_IS ..... 49  
 BAD\_K ..... 29, 48  
 BAD\_T ..... 29, 47  
 BAND generic linear solver  
   functions ..... 110  
   macros ..... 110  
   type `BandMat` ..... 108  
 BAND\_COL ..... 35, **110**  
 BAND\_COL\_ELEM ..... 35, **110**  
 BAND\_ELEM ..... 35, **110**  
 BAND\_LIW ..... 25  
 BAND\_LRW ..... 25  
 BAND\_NJE ..... 25  
 BandMat ..... 19, 35, 59, **108**  
 BDF ..... 22, 30, 31  
 BDF method ..... 4

## C

CONV\_FAILURE ..... 28  
 CVadjCheckPointsList ..... **57**  
 CVadjFree ..... **57**  
 CVadjMalloc ..... 53, **54**  
 CVBAND linear solver  
   Jacobian approximation used by ..... 25  
   memory requirements ..... 25  
   NVECTOR compatibility ..... 25, 33  
   optional outputs ..... 25  
   reinitialization ..... 33  
   selection of ..... 24–25  
   usage with adjoint module ..... 55–56

CVBand ..... 22, 23, **24**, 35  
 CVBandB ..... **56**, 58  
 CVBandDQJac ..... 25, 56  
 CVBandJacFn ..... **35**  
 CVBandJacFnB ..... **58**  
 CVBANDPRE preconditioner  
   description ..... 38  
   usage ..... 39  
   usage with adjoint module ..... 60–61  
 CVBandPreAlloc ..... 39  
 CVBandPrecon ..... 39  
 CVBandPreFree ..... 39  
 CVBandPSol ..... 39  
 CVBBD\_IPWSIZE ..... 42  
 CVBBD\_NGE ..... 42  
 CVBBD\_RPWSIZE ..... 42  
 CVBBDAlloc ..... 41  
 CVBBDFree ..... 42  
 CVBBDPRE preconditioner  
   additional user-supplied functions ..... 41  
   description ..... 40–41  
   optional output ..... 42  
   usage ..... 41–42  
 CVBBDPrecon ..... 41  
 CVBBDPSol ..... 41  
 CVBM\_NO\_MEM ..... 55  
 CVDENSE linear solver  
   Jacobian approximation used by ..... 24  
   memory requirements ..... 24  
   NVECTOR compatibility ..... 24, 33  
   optional outputs ..... 24  
   reinitialization ..... 32–33  
   selection of ..... 23–24  
   usage with adjoint module ..... 55  
 CVDense ..... 22, 23, **24**, 34  
 CVDenseB ..... 58  
 CVDenseDQJac ..... 24, 55  
 CVDenseJacFn ..... **34**  
 CVDenseJacFnB ..... **58**  
 CVDIAG linear solver  
   Jacobian approximation used by ..... 26  
   memory requirements ..... 26

optional outputs.....	26
selection of.....	25–26
CVDiag.....	22, 23, <b>25</b>
CVODE.....	1
CVode.....	22, <b>28</b> , 44
CVODE_NO_MEM.....	28
cvodea.h.....	52
CVodeB.....	53, <b>56</b>
CVodeDky.....	<b>29</b>
CVodeF.....	53, <b>54</b>
CVODEF_MEM_FAIL.....	54
CVodeFree.....	22, 45
CVodeMalloc.....	<b>22</b> , 31, 43, 52
CVodeMallocB.....	53, <b>54</b>
CVodeMemExtract.....	<b>48</b>
CVODES	
brief description of.....	1
motivation for writing in C.....	1–2
package structure.....	15
relationship to CVODE, PVODE.....	1
relationship to VODE, VODPK.....	1
CVODES linear solvers	
built on generic solvers.....	23
CVBAND.....	24, 33
CVDENSE.....	23, 32
CVDIAG.....	25
CVSPGMR.....	26, 33
header files.....	19
implementation details.....	17
list of.....	15
NVECTOR compatibility.....	19
reinitializing one.....	32
selecting one.....	23
usage with adjoint module.....	55
cvodes.h.....	19
CVodeSensDky.....	<b>49</b>
CVodeSensDkyAll.....	<b>48</b>
CVodeSensExtract.....	44, <b>47</b>
CVodeSensMalloc.....	44, <b>45</b> , 49
CVReInit.....	<b>31</b> , 32
CVReInitBand.....	<b>33</b>
CVReInitDense.....	<b>32</b>
CVReInitSpgmr.....	<b>33</b>
cvband.h.....	19
cvsdense.h.....	19

cvdiag.h.....	19
CVSensReInit.....	<b>49</b>
CVSensRhs1DQ.....	46, 50
CVSensRhsDQ.....	46, 50
CVSPGMR linear solver	
Jacobian approximation used by.....	27
memory requirements.....	27
optional inputs.....	27
optional outputs.....	27
preconditioner setup routine.....	27, 37
preconditioner solve routine.....	27, 36
reinitialization.....	33
selection of.....	26
usage with adjoint module.....	56
CVSpgmr.....	22, 23, <b>26</b>
CVSpgmrB.....	<b>56</b> , 61
CVSpgmrDQJtimes.....	27, 56
CVSpgmrJtimesFn.....	<b>36</b>
CVSpgmrJtimesFnB.....	<b>59</b>
CVSpgmrPrecondFn.....	<b>37</b>
CVSpgmrPrecondFnB.....	<b>60</b>
CVSpgmrPSolveFn.....	<b>36</b>
CVSpgmrPSolveFnB.....	<b>60</b>
cvsspgmr.h.....	20

## D

denaddI.....	<b>107</b>
denalloc.....	<b>106</b>
denallocpiv.....	<b>106</b>
dencopy.....	<b>107</b>
denfree.....	<b>107</b>
denfreepiv.....	<b>107</b>
denprint.....	<b>107</b>
denscale.....	<b>107</b>
DENSE generic linear solver	
functions	
large matrix.....	106
small matrix.....	106–107
macros.....	105–106
type DenseMat.....	105
DENSE_COL.....	34, <b>106</b>
DENSE_ELEM.....	34, <b>105</b>
DENSE_LIW.....	24
DENSE_LRW.....	24
DENSE_NJE.....	24

DenseMat ..... 19, 34, 58, **105**  
denzero ..... **107**  
DIAG\_LIW ..... 26  
DIAG\_LRW ..... 26  
DKY\_NO\_MEM ..... 31, 47  
DKY\_NO\_SENFI ..... 47

## E

ERR\_FAILURE ..... 28  
error control ..... 5–6, 8  
examples, adjoint sensitivity  
    list of ..... 82  
    parallel sample program  
        code `pvanx.c` ..... 179–189  
        explanation of ..... 86–87  
        output ..... 87–88  
        problem solved by ..... 85–86  
    serial sample program  
        code `cvadx.c` ..... 172–178  
        explanation of ..... 83–84  
        output ..... 84–85  
        problem solved by ..... 83  
        user-defined accessor macros ..... 83  
examples, forward sensitivity  
    list of ..... 69  
    parallel sample program  
        code `pvfkx.c` ..... 150–171  
        explanation of ..... 75–76  
        output ..... 76–81  
        problem solved by ..... 75  
    serial sample program  
        code `cvfdx.c` ..... 142–149  
        explanation of ..... 70–71  
        output ..... 71–75  
        problem solved by ..... 69–70  
        user data ..... 70  
examples, simulation  
    list of ..... 62–63  
    parallel sample program  
        code `pvkx.c` ..... 123–141  
        explanation of ..... 66  
        output ..... 66–68  
        problem solved by ..... 65–66  
    serial sample program  
        code `cvdx.c` ..... 118–122

explanation of ..... 63–65  
output ..... 65  
problem solved by ..... 63  
user-defined accessor macros ..... 63–65

## F

forward sensitivity analysis  
    absolute tolerance selection ..... 8  
    correction strategies ..... 6–8, 15, 45  
    examples ..... *see* examples, forward  
        sensitivity  
    mathematical background ..... 6–9  
    right hand side evaluation 8–9, 46, 50–51  
FULL ..... 46  
FUNCTIONAL ..... 22, 30

## G

gefa ..... **107**  
generic linear solvers  
    BAND ..... 108  
    DENSE ..... 105  
    SPGMR ..... 111  
    use in CVODES ..... 17  
gesl ..... **107**  
GMRES method ..... 27, 111  
Gram-Schmidt procedure ..... 27

## I

ILL\_INPUT ..... 28  
interpolated output ..... 29, 48  
iopt ..... 23, **29**, 30, 32, 47

## J

Jacobian approximation routine  
    band  
        difference quotient ..... 25  
        user-supplied ..... 25, 35–36  
    dense  
        difference quotient ..... 24  
        user-supplied ..... 24, 34–35  
    Jacobian times vector  
        difference quotient ..... 27  
        user-supplied ..... 27, 36

## L

LIN\_ILL\_INPUT ..... 33  
LINIT\_ERR ..... **102**

LINIT_OK .....	102
LMEM_FAIL .....	32, 33
LSODE .....	1

## M

M_Env .....	19, 91, <b>91</b>
M_EnvFree_Parallel .....	22, <b>98</b>
M_EnvFree_Serial .....	22, <b>95</b>
M_EnvInit_Parallel .....	20, <b>98</b>
M_EnvInit_Serial .....	20, <b>95</b>
maxord .....	31
ME_CONTENT_P .....	<b>98</b>
ME_CONTENT_S .....	<b>96</b>
MEXT_NO_MEM .....	48
MPI .....	3, 97, 98

## N

N_Vector .....	19, 91, <b>91</b>
N_VFree .....	22
N_VFree_S .....	44
N_VNew .....	20
N_VNew_S .....	44
NEWTON .....	22, 30, 31
NORMAL .....	22, 53
NV_CONTENT_P .....	<b>98</b>
NV_CONTENT_S .....	<b>96</b>
NV_DATA_P .....	<b>98</b>
NV_DATA_S .....	<b>96</b>
NV_DISPOSE_P .....	<b>99</b>
NV_DISPOSE_S .....	<b>96</b>
NV_GLOBLLENGTH_P .....	<b>98</b>
NV_Ith_P .....	<b>99</b>
NV_Ith_S .....	<b>96</b>
NV_LENGTH_S .....	<b>96</b>
NV_LOCLENGTH_P .....	<b>98</b>
NV_MAKE_P .....	<b>99</b>
NV_MAKE_S .....	<b>96</b>
nvector.h .....	19
nvector_parallel.h .....	19
nvector_serial.h .....	19
NVS_DISPOSE_P .....	<b>99</b>
NVS_DISPOSE_S .....	<b>96</b>
NVS_MAKE_P .....	<b>99</b>
NVS_MAKE_S .....	<b>96</b>

## O

OKAY .....	29, 48
ONE_STEP .....	22, 53
ONESENS .....	46, 51

## P

PARTIAL .....	46
preconditioning	
advice on .....	15, 26
setup and solve phases .....	15
PVODE .....	1

## R

RCONST .....	89
reinitialization .....	31, 32, 49
RhsFn .....	<b>34</b>
RhsFnB .....	<b>57</b>
right hand side function	
adjoint backward problem .....	57–58
forward sensitivity .....	50–51
initial value problem .....	34
ropt .....	23, <b>29</b> , 31

## S

SCVM_ILL_INPUT .....	47
SCVM_MEM_FAIL .....	47
SCVM_NO_MEM .....	47
SCVREI_ILL_INPUT .....	49
SCVREI_MEM_FAIL .....	49
SCVREI_NO_MEM .....	49
SCVREI_NO_SENSE .....	49
SensRhs1Fn .....	46, <b>51</b>
SensRhsFn .....	46, <b>50</b>
SETUP_FAILURE .....	29
SIMULTANEOUS .....	15, <b>45</b> , 50
SOLVE_FAILURE .....	29
SPGMR generic linear solver	
description of .....	111
functions .....	27–28, 111
support functions .....	111
SPGMR_LIW .....	27
SPGMR_LRW .....	27
SPGMR_NCFL .....	27
SPGMR_NLI .....	27
SPGMR_NPE .....	27
SPGMR_NPS .....	27

SS ..... 23, 46  
 STAGGERED ..... 15, **45**, 50  
 STAGGERED1 ..... 15, **45**, 48, 51  
 SUCCESS ..... 28, 32, 33, 47, 49, 54, 55  
 sundialtypes.h ..... 19, **89**  
 SV ..... 23, 46

## T

TOO\_MUCH\_ACC ..... 28  
 TOO\_MUCH\_WORK ..... 28  
 TSTOP\_RETURN ..... 28, 54, 57

## V

VODE ..... 1  
 VODPK ..... 1

## A Listings of CVODES IVP Solution Examples

### A.1 A Serial Sample Problem - cvdx.c

```

/*****
 *
 * File      : cvdx.c
 * Programmers: Scott D. Cohen and Alan C. Hindmarsh @LLNL
 * Version of : 5 March 2002
 *-----*
 * Modified by R. Serban to work with new serial nvector (5/3/2002)
 *-----*
 * Example problem.
 * The following is a simple example problem, with the coding
 * needed for its solution by CVODE. The problem is from chemical
 * kinetics, and consists of the following three rate equations..
 *   dy1/dt = -.04*y1 + 1.e4*y2*y3
 *   dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*(y2)^2
 *   dy3/dt = 3.e7*(y2)^2
 * on the interval from t = 0.0 to t = 4.e10, with initial conditions
 * y1 = 1.0, y2 = y3 = 0. The problem is stiff.
 * This program solves the problem with the BDF method, Newton
 * iteration with the CVODE dense linear solver, and a user-supplied
 * Jacobian routine.
 * It uses a scalar relative tolerance and a vector absolute tolerance.
 * Output is printed in decades from t = .4 to t = 4.e10.
 * Run statistics (optional outputs) are printed at the end.
 *****/

#include <stdio.h>

/* CVODE header files with a description of contents used in cvdx.c */

#include "sundialtypes.h" /* definitions of types realtype and          */
                          /* integertype, and the constant FALSE      */
#include "cvodes.h"       /* prototypes for CVMalloc, CVMalloc, and CVMallocFree, */
                          /* constants OPT_SIZE, BDF, NEWTON, SV, SUCCESS,      */
                          /* NST, NFE, NSETUPS, NNI, NCFN, NETF                */
#include "cvdense.h"      /* prototype for CVDense, constant DENSE_NJE          */
#include "nvector_serial.h" /* definitions of type N_Vector and macro NV_Ith_S,   */
                          /* prototypes for N_VNew, N_VFree                     */
#include "dense.h"        /* definitions of type DenseMat, macro DENSE_ELEM     */

/* User-defined vector and matrix accessor macros: Ith, IJth */

/* These macros are defined in order to write code which exactly matches
   the mathematical problem description given above.

   Ith(v,i) references the ith component of the vector v, where i is in
```

the range [1..NEQ] and NEQ is defined below. The Ith macro is defined using the N\_VIth macro in nvector.h. N\_VIth numbers the components of a vector starting from 0.

IJth(A,i,j) references the (i,j)th element of the dense matrix A, where i and j are in the range [1..NEQ]. The IJth macro is defined using the DENSE\_ELEM macro in dense.h. DENSE\_ELEM numbers rows and columns of a dense matrix starting from 0. \*/

```
#define Ith(v,i)    NV_Ith_S(v,i-1)      /* Ith numbers components 1..NEQ */
#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* IJth numbers rows,cols 1..NEQ */
```

/\* Problem Constants \*/

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

/\* Private Helper Function \*/

```
static void PrintFinalStats(long int iopt[]);
```

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

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

```
static void Jac(integertype N, DenseMat J, RhsFn f, void *f_data, realtype t,
               N_Vector y, N_Vector fy, N_Vector ewt, realtype h,
               realtype uround, void *jac_data, long int *nfePtr,
               N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
```

/\*\*\*\*\*\* Main Program \*\*\*\*\*/

```
int main()
{
    M_Env machEnv;
    realtype ropt[OPT_SIZE], reltol, t, tout;
    long int iopt[OPT_SIZE];
    N_Vector y, abstol;
```

```

void *cvmem;
int iout, flag;

/* Initialize serial machine environment */
machEnv = M_EnvInit_Serial(NEQ);

y = N_VNew(NEQ, machEnv); /* Allocate y, abstol vectors */
abstol = N_VNew(NEQ, machEnv);

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

reltol = RTOL; /* Set the scalar relative tolerance */
Ith(abstol,1) = ATOL1; /* Set the vector absolute tolerance */
Ith(abstol,2) = ATOL2;
Ith(abstol,3) = ATOL3;

/* Call CVMalloc to initialize CVMODE:

    NEQ      is the problem size = number of equations
    f        is the user's right hand side function in y'=f(t,y)
    T0       is the initial time
    y        is the initial dependent variable vector
    BDF      specifies the Backward Differentiation Formula
    NEWTON    specifies a Newton iteration
    SV       specifies scalar relative and vector absolute tolerances
    &reltol   is a pointer to the scalar relative tolerance
    abstol   is the absolute tolerance vector
    FALSE    indicates there are no optional inputs in iopt and ropt
    iopt      is an array used to communicate optional integertype input and output
    ropt      is an array used to communicate optional realtype input and output

    A pointer to CVMODE problem memory is returned and stored in cvmem. */

cvmem = CVMalloc(NEQ, f, T0, y, BDF, NEWTON, SV, &reltol, abstol,
                NULL, NULL, FALSE, iopt, ropt, machEnv);
if (cvmem == NULL) { printf("CVMalloc failed.\n"); return(1); }

/* Call CVDense to specify the CVMODE dense linear solver with the
   user-supplied Jacobian routine Jac. */

flag = CVDense(cvmem, Jac, NULL);
if (flag != SUCCESS) { printf("CVDense failed.\n"); return(1); }

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

printf(" \n3-species kinetics problem\n\n");
for (iout=1, tout=T1; iout <= NOUT; iout++, tout += TMULT) {
    flag = CVMODE(cvmem, tout, y, &t, NORMAL);
    printf("At t = %0.4e      y =%14.6e %14.6e %14.6e\n",

```



```

        t, Ith(y,1), Ith(y,2), Ith(y,3));
    if (flag != SUCCESS) { printf("Cvode failed, flag=%d.\n", flag); break; }
}

N_VFree(y);                /* Free the y and abstol vectors */
N_VFree(abstol);
CvodeFree(cvode_mem);      /* Free the CVODE problem memory */
M_EnvFree_Serial(machEnv); /* Free the machine environment memory */

PrintFinalStats(iopt);      /* Print some final statistics */
return(0);
}

/***** Private Helper Function *****/

/* Print some final statistics located in the iopt array */

static void PrintFinalStats(long int iopt[])
{
    printf("\nFinal Statistics.. \n\n");
    printf("nst = %-6ld nfe = %-6ld nsetups = %-6ld nje = %ld\n",
        iopt[NST], iopt[NFE], iopt[NSETUPS], iopt[DENSE_NJE]);
    printf("nni = %-6ld ncnf = %-6ld netf = %ld\n \n",
        iopt[NNI], iopt[NCFN], iopt[NETF]);
}

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

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

static void f(integertype N, realtype t, N_Vector y, N_Vector ydot, void *f_data)
{
    realtype y1, y2, y3, yd1, yd3;

    y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

    yd1 = Ith(ydot,1) = -0.04*y1 + 1e4*y2*y3;
    yd3 = Ith(ydot,3) = 3e7*y2*y2;
    Ith(ydot,2) = -yd1 - yd3;
}

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

static void Jac(integertype N, DenseMat J, RhsFn f, void *f_data, realtype t,
    N_Vector y, N_Vector fy, N_Vector ewt, realtype h,
    realtype uround, void *jac_data, long int *nfePtr,
    N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
    realtype y1, y2, y3;

```

```

y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

IJth(J,1,1) = -0.04;  IJth(J,1,2) = 1e4*y3;          IJth(J,1,3) = 1e4*y2;
IJth(J,2,1) =  0.04;  IJth(J,2,2) = -1e4*y3-6e7*y2;  IJth(J,2,3) = -1e4*y2;
                    IJth(J,3,2) = 6e7*y2;
}

```

## A.2 A Parallel Sample Program - pvkx.c

```

/*****
*
* File      : pvkx.c
* Programmers: S. D. Cohen, A. C. Hindmarsh, M. R. Wittman @ LLNL
* Version of : 6 March 2002
*-----*
* Modified by R. Serban to work with new parallel nvector (6/3/2002) *
*-----*
* Example problem.
* An ODE system is generated from the following 2-species diurnal
* kinetics advection-diffusion PDE system in 2 space dimensions:
*
*  $dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)$ 
*               +  $Ri(c1,c2,t)$  for  $i = 1,2$ , where
*  $R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2$ ,
*  $R2(c1,c2,t) = q1*c1*c3 - q2*c1*c2 - q4(t)*c2$ ,
*  $Kv(y) = Kv0*exp(y/5)$ ,
*  $Kh, V, Kv0, q1, q2$ , and  $c3$  are constants, and  $q3(t)$  and  $q4(t)$ 
* vary diurnally. The problem is posed on the square
*  $0 \leq x \leq 20$ ,  $30 \leq y \leq 50$  (all in km),
* with homogeneous Neumann boundary conditions, and for time  $t$  in
*  $0 \leq t \leq 86400$  sec (1 day).
* The PDE system is treated by central differences on a uniform
* mesh, with simple polynomial initial profiles.
*
* The problem is solved by CVODE on NPE processors, treated as a
* rectangular process grid of size NPEX by NPEY, with  $NPE = NPEX*NPEY$ .
* Each processor contains a subgrid of size MXSUB by MYSUB of the
* (x,y) mesh. Thus the actual mesh sizes are  $MX = MXSUB*NPEX$  and
*  $MY = MYSUB*NPEY$ , and the ODE system size is  $neq = 2*MX*MY$ .
*
* The solution with CVODE is done with the BDF/GMRES method (i.e.
* using the CVSPGMR linear solver) and the block-diagonal part of the
* Newton matrix as a left preconditioner. A copy of the block-diagonal
* part of the Jacobian is saved and conditionally reused within the
* Precond routine.
*
* Performance data and sampled solution values are printed at selected
* output times, and all performance counters are printed on completion.
*
* This version uses MPI for user routines, and the MPI_PVODE solver.
* Execution: pvkx -npes N with  $N = NPEX*NPEY$  (see constants below).
*****/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sundialstypes.h"      /* definitions of realtype, integertype, */
                                /* booleantype, and consants TRUE, FALSE */

```

```

#include "cvodes.h"          /* main CVODE header file */
#include "iterativ.h"        /* contains the enum for types of preconditioning */
#include "cvsspgmr.h"        /* use CVSPGMR linear solver each internal step */
#include "smalldense.h"      /* use generic DENSE solver in preconditioning */
#include "nvector_parallel.h" /* definitions of type N_Vector, macro NV_DATA_P */
#include "sundialsmath.h"    /* contains SQR macro */
#include "mpi.h"

/* Problem Constants */

#define NVARs      2          /* number of species */
#define KH         4.0e-6     /* horizontal diffusivity Kh */
#define VEL        0.001      /* advection velocity V */
#define KVO        1.0e-8     /* coefficient in Kv(y) */
#define Q1         1.63e-16   /* coefficients q1, q2, c3 */
#define Q2         4.66e-16
#define C3         3.7e16
#define A3         22.62      /* coefficient in expression for q3(t) */
#define A4         7.601      /* coefficient in expression for q4(t) */
#define C1_SCALE   1.0e6      /* coefficients in initial profiles */
#define C2_SCALE   1.0e12

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

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

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

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

/* CVMalloc Constants */

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

```

```

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

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

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

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

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

typedef struct {
    realtype q4, om, dx, dy, hdco, haco, vdco;
    realtype uext[NVARs*(MXSUB+2)*(MYSUB+2)];
    integertype my_pe, isubx, isuby, nvmsub, nvmsub2;
    MPI_Comm comm;
} *UserData;

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

/* Private Helper Functions */

static PreconData AllocPreconData(UserData data);
static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
static void FreePreconData(PreconData pdata);
static void SetInitialProfiles(N_Vector u, UserData data);
static void PrintOutput(integertype my_pe, MPI_Comm comm, long int iopt[],
                        realtype ropt[], N_Vector u, realtype t);
static void PrintFinalStats(long int iopt[]);
static void BSend(MPI_Comm comm, integertype my_pe, integertype isubx,
                  integertype isuby, integertype dsize, integertype dsizey,
                  realtype udata[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[], integertype my_pe,
                     integertype isubx, integertype isuby,
                     integertype dsize, integertype dsizey,
                     realtype uext[], realtype buffer[]);
static void BRecvWait(MPI_Request request[], integertype isubx, integertype isuby,
                     integertype dsize, realtype uext[], realtype buffer[]);
static void ucomm(integertype N, realtype t, N_Vector u, UserData data);

```

```

static void fcalc(integertype N, realtype t, realtype udata[], realtype dudata[],
                  UserData data);

/* Functions Called by the CVODE Solver */

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

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

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

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

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

    /* Set problem size neq */

    neq = NVAR*MX*MY;

    /* Get processor number and total number of pe's */

    MPI_Init(&argc, &argv);
    comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &my_pe);

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

    /* Set local length */

```

```

local_N = NVAR*MXSUB*MYSUB;

/* Allocate and load user data block; allocate preconditioner block */

data = (UserData) malloc(sizeof *data);
InitUserData(my_pe, comm, data);
predata = AllocPreconData (data);

/* Set machEnv block */

machEnv = M_EnvInit_Parallel(comm, local_N, neq, &argc, &argv);
if (machEnv == NULL) return(1);

/* Allocate u, and set initial values and tolerances */

u = N_VNew(neq, machEnv);
SetInitialProfiles(u, data);
abstol = ATOL; reltol = RTOL;

/* Call CVodeMalloc to initialize CVODE:

    neq      is the problem size = number of equations
    f        is the user's right hand side function in u'=f(t,u)
    T0       is the initial time
    u        is the initial dependent variable vector
    BDF       specifies the Backward Differentiation Formula
    NEWTON    specifies a Newton iteration
    SS       specifies scalar relative and absolute tolerances
    &reltol and &abstol are pointers to the scalar tolerances
    data     is the pointer to the user-defined block of coefficients
    FALSE    indicates there are no optional inputs in iopt and ropt
    iopt     and ropt arrays communicate optional integer and real input/output

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

ccode_mem = CVodeMalloc(neq, f, T0, u, BDF, NEWTON, SS, &reltol,
                        &abstol, data, NULL, FALSE, iopt, ropt, machEnv);
if (ccode_mem == NULL) { printf("CVodeMalloc failed."); return(1); }

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

flag = CVSpgrmr(ccode_mem, LEFT, MODIFIED_GS, 0, 0.0, Precond, PSolve,
                predata, NULL, NULL);
if (flag != SUCCESS) { printf("CVSpgrmr failed."); return(1); }

```

```

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

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

for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CNode(cnode_mem, tout, u, &t, NORMAL);
    PrintOutput(my_pe, comm, iopt, ropt, u, t);
    if (flag != SUCCESS) {
        if (my_pe == 0) printf("CNode failed, flag=%d.\n", flag);
        break;
    }
}

/* Free memory and print final statistics */

N_VFree(u);
free(data);
FreePreconData(predata);
CNodeFree(cnode_mem);
if (my_pe == 0) PrintFinalStats(iopt);
M_EnvFree_Parallel(machEnv);
MPI_Finalize();

return(0);
}

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

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

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

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

    pdata->f_data = fdata;

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

    return(pdata);
}

```



```

/* Load constants in data */

static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
{
    integertype isubx, isuby;

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

    /* Set machine-related constants */
    data->comm = comm;
    data->my_pe = my_pe;
    /* isubx and isuby are the PE grid indices corresponding to my_pe */
    isuby = my_pe/NPEX;
    isubx = my_pe - isuby*NPEX;
    data->isubx = isubx;
    data->isuby = isuby;
    /* Set the sizes of a boundary x-line in u and uest */
    data->nvmxsub = NVAR*MXSUB;
    data->nvmxsub2 = NVAR*(MXSUB+2);
}

/* Free preconditioner data memory */

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

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

    free(pdata);
}

/* Set initial conditions in u */

static void SetInitialProfiles(N_Vector u, UserData data)
{
    integertype isubx, isuby, lx, ly, jx, jy, offset;

```

```

realtype dx, dy, x, y, cx, cy, xmid, ymid;
realtype *udata;

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

udata = NV_DATA_P(u);

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

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

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

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

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

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

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

    /* Send c1,c2 at top right mesh point to PE 0 */
    if (my_pe == npelast) {
        i0 = NVAR*MXSUB*MYSUB - 2;

```

```

    i1 = i0 + 1;
    if (npelast != 0)
        MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
        tempu[0] = udata[i0];
        tempu[1] = udata[i1];
    }
}

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

/* Print final statistics contained in iopt */

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

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

static void BSend(MPI_Comm comm, integertype my_pe, integertype isubx,
                  integertype isuby, integertype dsize, integertype dsizey,
                  realtype udata[])
{
    int i, ly;
    integertype offsetu, offsetbuf;
    realtype buyleft[NVARS*MYSUB], bufright[NVARS*MYSUB];

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

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

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

```

```

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

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

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

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

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

}

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

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

    /* If isuby > 0, receive data for bottom x-line of uext */
    if (isuby != 0)
        MPI_Irecv(&uext[NVAR], dsizex, PVEC_REAL_MPI_TYPE,

```

```

        my_pe-NPEX, 0, comm, &request[0]));

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

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

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

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

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

    dsizex2 = dsizex + 2*NVAR;

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

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

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

```

```

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

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

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

}

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

static void ucomm(integertype N, realtype t, N_Vector u, UserData data)
{
    realtype *udata, *uext, buffer[2*NVAR*MYSUB];
    MPI_Comm comm;
    integertype my_pe, isubx, isuby, nvmsub, nvmysub;
    MPI_Request request[4];

    udata = NV_DATA_P(u);

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

    comm = data->comm; my_pe = data->my_pe;
    isubx = data->isubx; isuby = data->isuby;
    nvmsub = data->nvmsub;
    nvmysub = NVAR*MYSUB;
    uext = data->uext;

    /* Start receiving boundary data from neighboring PEs */

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

```

```

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

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

/* Finish receiving boundary data from neighboring PEs */

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

}

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

static void fcalc(integertype N, realtype t, realtype udata[], realtype dudata[],
                  UserData data)
{
    realtype *uext;
    realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
    realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
    realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
    realtype q4coef, dely, verdco, hordco, horaco;
    int i, lx, ly, jx, jy;
    integertype isubx, isuby, nvmsub, nvmsub2, offsetu, offsetue;

    /* Get subgrid indices, data sizes, extended work array uext */

    isubx = data->isubx;    isuby = data->isuby;
    nvmsub = data->nvmsub; nvmsub2 = data->nvmsub2;
    uext = data->uext;

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

    offsetu = 0;
    offsetue = nvmsub2 + Nvars;
    for (ly = 0; ly < MYSUB; ly++) {
        for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
        offsetu = offsetu + nvmsub;
        offsetue = offsetue + nvmsub2;
    }

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

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

```

```

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

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

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

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

dely = data->dy;
verdco = data->vdco;
hordco = data->hdco;
horaco = data->haco;

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

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

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

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

    jy = ly + isuby*MYSUB;

```



```

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

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

    jx = lx + isubx*MXSUB;

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

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

    /* Set vertical diffusion terms */

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

    /* Set horizontal diffusion and advection terms */

    c1lt = uext[offsetue-2];
    c2lt = uext[offsetue-1];
    c1rt = uext[offsetue+2];
    c2rt = uext[offsetue+3];
    hord1 = hordco*(c1rt - 2.0*c1 + c1lt);
    hord2 = hordco*(c2rt - 2.0*c2 + c2lt);
    horad1 = horaco*(c1rt - c1lt);
    horad2 = horaco*(c2rt - c2lt);

    /* Load all terms into dudata */

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

```

```

}

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

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

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

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

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

    ucomm (N, t, u, data);

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

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

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

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

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

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

```

```

Jbd = predata->Jbd;
pivot = predata->pivot;
udata = NV_DATA_P(u);
isubx = data->isubx;  isuby = data->isuby;
nvmxsub = data->nvmxsub;

if (jok) {

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

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

*jcurPtr = FALSE;

}

else {

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

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

q4coef = data->q4;
dely = data->dy;
verdco = data->vdco;
hordco = data->hdco;

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

for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    ydn = YMIN + (jy - .5)*dely;
    yup = ydn + dely;
    cydn = verdco*exp(0.2*ydn);
    cyup = verdco*exp(0.2*yup);
    diag = -(cydn + cyup + 2.0*hordco);
    for (lx = 0; lx < MXSUB; lx++) {
        jx = lx + isubx*MXSUB;
        offset = lx*NVARs + ly*nvmxsub;
        c1 = udata[offset];
        c2 = udata[offset+1];
        j = Jbd[lx][ly];
        a = P[lx][ly];
        IJth(j,1,1) = (-Q1*c3 - Q2*c2) + diag;
        IJth(j,1,2) = -Q2*c1 + q4coef;
        IJth(j,2,1) = Q1*c3 - Q2*c2;
        IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
        dencopy(j, a, NVARs);
    }
}

```

```

    }
}

*jcurPtr = TRUE;

}

/* Scale by -gamma */

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

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

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

return(0);
}

/* Preconditioner solve routine */

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

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

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

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

```

```

N_VScale(1.0, r, z);

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

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

return(0);
}

```

## B Listings of CVODES Forward Sensitivity Examples

### B.1 A Serial Sample Problem - cvfdx.c

```

/*****
 *
 * File      : cvfdx.c
 * Programmers: Scott D. Cohen, Alan C. Hindmarsh, and Radu Serban
 *           @ LLNL
 * Version of : 20 March 2002
 *-----*
 * Example problem.
 * The following is a simple example problem, with the coding
 * needed for its solution by CVODES. The problem is from chemical
 * kinetics, and consists of the following three rate equations..
 *   dy1/dt = -p1*y1 + p2*y2*y3
 *   dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
 *   dy3/dt =  p3*(y2)^2
 * on the interval from t = 0.0 to t = 4.e10, with initial conditions
 * y1 = 1.0, y2 = y3 = 0. The reaction rates are: p1=0.04, p2=1e4, and
 * p3=3e7. The problem is stiff.
 * This program solves the problem with the BDF method, Newton
 * iteration with the CVODES dense linear solver, and a user-supplied
 * Jacobian routine.
 * It uses a scalar relative tolerance and a vector absolute tolerance.
 * Output is printed in decades from t = .4 to t = 4.e10.
 * Run statistics (optional outputs) are printed at the end.
 *
 * Optionally, CVODES can compute sensitivities with respect to the
 * problem parameters p1, p2, and p3.
 * Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
 * STAGGERED1) can be used and sensitivities may be included in the
 * error test or not (error control set on FULL or PARTIAL,
 * respectively).
 *
 * Execution:
 *
 * If no sensitivities are desired:
 *   % cvsdx -nosensi
 * If sensitivities are to be computed:
 *   % cvsdx -sensi sensi_meth err_con
 * where sensi_meth is one of {sim, stg, stg1} and err_con is one of
 * {full, partial}.
 *
 *****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sundialstypes.h" /* definitions of types realtype and
                          /* integertype, and the constant FALSE
                          */

```

```

#include "cvodes.h"          /* prototypes for CNodeMalloc, CNode, and CNodeFree, */
                             /* constants OPT_SIZE, BDF, NEWTON, SV, SUCCESS, */
                             /* NST, NFE, NSETUPS, NNI, NCFN, NETF */
#include "cvsdense.h"        /* prototype for CVDense, constant DENSE_NJE */
#include "nvector_serial.h"   /* definitions of type N_Vector and macro NV_Ith_S, */
                             /* prototypes for N_VNew, N_VFree */
#include "dense.h"           /* definitions of type DenseMat, macro DENSE_ELEM */

#define Ith(v,i)    NV_Ith_S(v,i-1)      /* Ith numbers components 1..NEQ */
#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* IJth numbers rows,cols 1..NEQ */

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

#define NP     3
#define NS     3

#define ZERO   0.0

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

/* Private Helper Function */

static void WrongArgs(char *argv[]);
static void PrintFinalStats(booleantype sensi, int sensi_meth, int err_con,
                           long int iopt[]);
static void PrintOutput(long int iopt[], realtype ropt[], realtype t, N_Vector u);
static void PrintOutputS(N_Vector *uS);

/* Functions Called by the CVODES Solver */

static void f(integertype N, realtype t, N_Vector y, N_Vector ydot, void *f_data);
static void Jac(integertype N, DenseMat J, RhsFn f, void *f_data, realtype t,
               N_Vector y, N_Vector fy, N_Vector ewt, realtype h, realtype around,
               void *jac_data, long int *nfePtr, N_Vector vtemp1,
               N_Vector vtemp2, N_Vector vtemp3);

```

```

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

```

```

int main(int argc, char *argv[])
{
    M_Env machEnv;
    UserData data;
    realtype ropt[OPT_SIZE], reltol, t, tout;
    long int iopt[OPT_SIZE];
    N_Vector y, abstol;
    void *cnode_mem;
    int iout, flag;

    realtype pbar[NP], rhomax;
    integertype is, *plist;
    N_Vector *yS;
    booleantype sensi;
    int sensi_meth, err_con, ifS;

    /* Process arguments */
    if (argc < 2)
        WrongArgs(argv);

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

    if (sensi) {

        if (argc != 4)
            WrongArgs(argv);

        if (strcmp(argv[2], "sim") == 0)
            sensi_meth = SIMULTANEOUS;
        else if (strcmp(argv[2], "stg") == 0)
            sensi_meth = STAGGERED;
        else if (strcmp(argv[2], "stg1") == 0)
            sensi_meth = STAGGERED1;
        else
            WrongArgs(argv);

        if (strcmp(argv[3], "full") == 0)
            err_con = FULL;
        else if (strcmp(argv[3], "partial") == 0)
            err_con = PARTIAL;
        else
            WrongArgs(argv);
    }
}

```



```

}

/* Initialize serial machine environment */
machEnv = M_EnvInit_Serial(NEQ);

/* USER DATA STRUCTURE */
data = (UserData) malloc(sizeof *data);
data->p[0] = 0.04;
data->p[1] = 1.0e4;
data->p[2] = 3.0e7;

/* INITIAL STATES */
y = N_VNew(NEQ, machEnv);
abstol = N_VNew(NEQ, machEnv);

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

/* TOLERANCES */
/* Set the scalar relative tolerance */
reltol = RTOL;
/* Set the vector absolute tolerance */
Ith(abstol,1) = ATOL1;
Ith(abstol,2) = ATOL2;
Ith(abstol,3) = ATOL3;

/* CVOICE_MALLOC */
cvoice_mem = CVoiceMalloc(NEQ, f, TO, y, BDF, NEWTON, SV, &reltol, abstol,
                        data, NULL, FALSE, iopt, ropt, machEnv);
if (cvoice_mem == NULL) {
    printf("CVoiceMalloc failed.\n");
    return(1);
}

/* CVDENSE */
flag = CVDense(cvoice_mem, Jac, NULL);
if (flag != SUCCESS) { printf("CVDense failed.\n"); return(1); }

/* SENSITIVITY */
if(sensi) {
    pbar[0] = data->p[0];
    pbar[1] = data->p[1];
    pbar[2] = data->p[2];
    plist = (integertype *) malloc(NS * sizeof(integertype));
    for(is=0;is<NS;is++) plist[is] = is+1;

    yS = N_VNew_S(NS, NEQ, machEnv);
    for(is=0;is<NS;is++)

```

```

    N_VConst(0.0, yS[is]);

    ifS = ALLSENS;
    if(sensi_meth==STAGGERED1) ifS = ONESENS;

    rhomax = ZERO;
    flag = CNodeSensMalloc(cvode_mem, NS, sensi_meth, data->p, pbar, plist,
                          ifS, NULL, err_con, rhomax, yS, NULL, NULL);
    if (flag != SUCCESS) {
        printf("CNodeSensMalloc failed, flag=%d\n",flag);
        return(1);
    }
}

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

printf("\n3-species chemical kinetics problem\n\n");
printf("=====");
printf("=====\n");
printf("      T      Q      H      NST                      y1");
printf("      y2          y3      \n");
printf("=====");
printf("=====\n");

for (iout=1, tout=T1; iout <= NOUT; iout++, tout *= TMULT) {
    flag = CNode(cvode_mem, tout, y, &t, NORMAL);
    if (flag != SUCCESS) {
        printf("CNode failed, flag=%d.\n", flag);
        break;
    }
    PrintOutput(iopt, ropt, t, y);
    if (sensi) {
        flag = CNodeSensExtract(cvode_mem, t, yS);
        if (flag != SUCCESS) {
            printf("CNodeSensExtract failed, flag=%d.\n", flag);
            break;
        }
        PrintOutputS(yS);
    }
    printf("-----");
    printf("-----\n");
}

/* Print final statistics */
PrintFinalStats(sensi,sensi_meth,err_con,iopt);

/* Free memory */
N_VFree(y);                /* Free the y and abstol vectors */
N_VFree(abstol);
if(sensi) N_VFree_S(NS, yS); /* Free the yS vectors */
free(data);                /* Free user data */

```

```

    CVodeFree(cvode_mem);          /* Free the CVODES problem memory */
    M_EnvFree_Serial(machEnv);     /* Free the machine environment memory */

    return(0);
}

/***** Private Helper Function *****/

/* ===== */
/* Exit if arguments are incorrect */

static void WrongArgs(char *argv[])
{
    printf("\nUsage: %s [-nosensi] [-sensi sensi_meth err_con]\n",argv[0]);
    printf("          sensi_meth = sim, stg, or stg1\n");
    printf("          err_con      = full or partial\n");

    exit(0);
}

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

static void PrintOutput(long int iopt[], realtype ropt[], realtype t, N_Vector u)
{
    realtype *udata;

    udata = NV_DATA_S(u);

    printf("%8.3e %2ld  %8.3e %5ld\n", t,iopt[QU],ropt[HU],iopt[NST]);
    printf("          Solution          ");
    printf("%12.4e %12.4e %12.4e \n", udata[0], udata[1], udata[2]);
}

/* ===== */
/* Print sensitivities */

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

    sdata = NV_DATA_S(uS[0]);
    printf("          Sensitivity 1  ");
    printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);

    sdata = NV_DATA_S(uS[1]);
    printf("          Sensitivity 2  ");
    printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);
}

```

```

    sdata = NV_DATA_S(uS[2]);
    printf("                                Sensitivity 3  ");
    printf("%12.4e %12.4e %12.4e \n", sdata[0], sdata[1], sdata[2]);

}

/* ===== */
/* Print some final statistics located in the iopt array */

static void PrintFinalStats(boolean_type sensi, int sensi_meth, int err_con,
                           long int iopt[])
{
    printf("\n\n=====");
    printf("\nFinal Statistics");
    printf("\nSensitivity: ");

    if(sensi) {
        printf("YES ");
        if(sensi_meth == SIMULTANEOUS)
            printf("( SIMULTANEOUS +");
        else
            if(sensi_meth == STAGGERED) printf("( STAGGERED +");
            else printf("( STAGGERED1 +");
        if(err_con == FULL) printf(" FULL ERROR CONTROL ");
        else printf(" PARTIAL ERROR CONTROL ");
    } else {
        printf("NO");
    }

    printf("\n\n");
    /*
    printf("lenrw   = %5ld    leniw = %5ld\n", iopt[LENRW], iopt[LENIW]);
    printf("llrw    = %5ld    lliw  = %5ld\n", iopt[SPGMR_LRW], iopt[SPGMR_LIW]);
    */
    printf("nst      = %5ld          \n\n", iopt[NST]);
    printf("nfe      = %5ld    nfSe  = %5ld \n", iopt[NFE], iopt[NFSE]);
    printf("nni      = %5ld    nniS   = %5ld \n", iopt[NNI], iopt[NNIS]);
    printf("ncfn     = %5ld    ncfns  = %5ld \n", iopt[NCFN], iopt[NCFNS]);
    printf("netf     = %5ld    netfS  = %5ld\n\n", iopt[NETF], iopt[NETFS]);
    printf("nsetups  = %5ld          \n", iopt[NSETUPS]);
    printf("nje      = %5ld          \n", iopt[DENSE_NJE]);

    printf("=====\n");

}

/***** Functions Called by the CVODES Solver *****/

```

```

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

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

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

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

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

static void Jac(integertype N, DenseMat J, RhsFn f, void *f_data, realtype t,
                N_Vector y, N_Vector fy, N_Vector ewt, realtype h, realtype uround,
                void *jac_data, long int *nfePtr, N_Vector vtemp1,
                N_Vector vtemp2, N_Vector vtemp3)
{
    realtype y1, y2, y3;
    UserData data;
    realtype p1, p2, p3;

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

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

```

## B.2 A Parallel Sample Program - pvfkx.c

```

/*****
*
* File      : pvfkx.c
* Programmers: S. D. Cohen, A. C. Hindmarsh, Radu Serban, and
*              M. R. Wittman @ LLNL
* Version of : 21 March 2002
*-----*
* Example problem.
* An ODE system is generated from the following 2-species diurnal
* kinetics advection-diffusion PDE system in 2 space dimensions:
*
*  $dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy$ 
*  $+ Ri(c1,c2,t)$  for  $i = 1,2$ , where
*  $R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2$ ,
*  $R2(c1,c2,t) = q1*c1*c3 - q2*c1*c2 - q4(t)*c2$ ,
*  $Kv(y) = Kv0*exp(y/5)$ ,
*  $Kh$ ,  $V$ ,  $Kv0$ ,  $q1$ ,  $q2$ , and  $c3$  are constants, and  $q3(t)$  and  $q4(t)$ 
* vary diurnally. The problem is posed on the square
*  $0 \leq x \leq 20$ ,  $30 \leq y \leq 50$  (all in km),
* with homogeneous Neumann boundary conditions, and for time  $t$  in
*  $0 \leq t \leq 86400$  sec (1 day).
* The PDE system is treated by central differences on a uniform
* mesh, with simple polynomial initial profiles.
*
* The problem is solved by CVODES on NPE processors, treated as a
* rectangular process grid of size NPEX by NPEY, with  $NPE = NPEX*NPEY$ .
* Each processor contains a subgrid of size MXSUB by MYSUB of the
* (x,y) mesh. Thus the actual mesh sizes are  $MX = MXSUB*NPEX$  and
*  $MY = MYSUB*NPEY$ , and the ODE system size is  $neq = 2*MX*MY$ .
*
* The solution with CVODES is done with the BDF/GMRES method (i.e.
* using the CVSPGMR linear solver) and the block-diagonal part of the
* Newton matrix as a left preconditioner. A copy of the block-diagonal
* part of the Jacobian is saved and conditionally reused within the
* Precond routine.
*
* Performance data and sampled solution values are printed at selected
* output times, and all performance counters are printed on completion.
*
* Optionally, CVODES can compute sensitivities with respect to the
* problem parameters  $q1$  and  $q2$ .
* Any of three sensitivity methods (SIMULTANEOUS, STAGGERED, and
* STAGGERED1) can be used and sensitivities may be included in the
* error test or not (error control set on FULL or PARTIAL,
* respectively).
*
* Execution:
*
* NOTE: This version uses MPI for user routines, and the CVODES
*****/

```

```

*      solver. In what follows, N is the number of processors,      *
*      N = NPEX*NPEY (see constants below) and it is assumed that  *
*      the MPI script mpirun is used to run a parallel application. *
* If no sensitivities are desired:                                   *
*   % mpirun -np N pvfkx -nosensi                                  *
* If sensitivities are to be computed:                             *
*   % mpirun -np N pvfkx -sensi sensi_meth err_con                 *
* where sensi_meth is one of {sim, stg, stg1} and err_con is one of *
* {full, partial}.                                                *
*                                                                    *
*****/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "sundialstypes.h" /* definitions of realtype, integertype, */
                          /* booleantype, and constants TRUE, FALSE */
#include "cvodes.h" /* main CVODES header file */
#include "iterativ.h" /* contains the enum for types of preconditioning */
#include "cvsspghmr.h" /* use CVSPGMR linear solver each internal step */
#include "smalldense.h" /* use generic DENSE solver in preconditioning */
#include "nvector_parallel.h" /* definitions of type N_Vector, macro N_VDATA */
#include "sundialsmath.h" /* contains SQR macro */
#include "mpi.h"

/* Problem Constants */

#define NVAR 2 /* number of species */
#define C1_SCALE 1.0e6 /* coefficients in initial profiles */
#define C2_SCALE 1.0e12

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

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

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

#define MX (NPEX*MXSUB) /* MX = number of x mesh points */

```

```

#define MY          (NPEY*MYSUB)  /* MY = number of y mesh points      */
                                  /* Spatial mesh is MX by MY          */
/* CVMalloc Constants */

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

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

#define ZERO        RCONST(0.0)

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

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

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

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

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

typedef struct {
    realtype *p;
    realtype q4, om, dx, dy, hdco, haco, vdco;
    realtype uext[NVARs*(MXSUB+2)*(MYSUB+2)];
    integertype my_pe, isubx, isuby, nvmsub, nvmsub2;
    MPI_Comm comm;
} *UserData;

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

/* Private Helper Functions */

static void WrongArgs(integertype my_pe, char *argv[]);
static PreconData AllocPreconData(UserData data);
static void InitUserData(integertype my_pe, MPI_Comm comm, UserData data);
static void FreePreconData(PreconData pdata);

```



```

static void SetInitialProfiles(N_Vector u, UserData data);
static void PrintOutput(integertype my_pe, MPI_Comm comm, long int iopt[],
                        realtype ropt[], realtype t, N_Vector u);
static void PrintOutputS(integertype my_pe, MPI_Comm comm, N_Vector *uS);
static void PrintFinalStats(booleanype sensi, int sensi_meth, int err_con,
                            long int iopt[]);
static void BSend(MPI_Comm comm, integertype my_pe, integertype isubx,
                  integertype isuby, integertype dsizeX, integertype dsizeY,
                  realtype udata[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[], integertype my_pe,
                      integertype isubx, integertype isuby,
                      integertype dsizeX, integertype dsizeY,
                      realtype uext[], realtype buffer[]);
static void BRecvWait(MPI_Request request[], integertype isubx, integertype isuby,
                      integertype dsizeX, realtype uext[], realtype buffer[]);
static void ucomm(integertype N, realtype t, N_Vector u, UserData data);
static void fcalc(integertype N, realtype t, realtype udata[],
                  realtype dudata[], UserData data);

/* Functions Called by the CVODES Solver */

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

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

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

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

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

    realtype *pbar, rhomax;
    integertype is, *plist;
    N_Vector *uS;

```

```

booleantype sensi;
int sensi_meth, err_con, ifS;

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

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

/* Process arguments */
if (argc < 2)
    WrongArgs(my_pe,argv);

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

if (sensi) {

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

    if (strcmp(argv[2],"sim") == 0)
        sensi_meth = SIMULTANEOUS;
    else if (strcmp(argv[2],"stg") == 0)
        sensi_meth = STAGGERED;
    else if (strcmp(argv[2],"stg1") == 0)
        sensi_meth = STAGGERED1;
    else
        WrongArgs(my_pe,argv);

    if (strcmp(argv[3],"full") == 0)
        err_con = FULL;
    else if (strcmp(argv[3],"partial") == 0)
        err_con = PARTIAL;
    else
        WrongArgs(my_pe,argv);
}

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

```

```

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

/* Allocate and load user data block; allocate preconditioner block */
data = (UserData) malloc(sizeof *data);
data->p = (realtype *) malloc(NP*sizeof(realtype));
InitUserData(my_pe, comm, data);
predata = AllocPreconData (data);

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

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

for (is=0; is<OPT_SIZE; is++) {
    iopt[is] = 0;
    ropt[is] = 0.0;
}
iopt[MXSTEP] = 2000;

/* CVMEM_MALLOC */
cvmem = CVodeMalloc(neq, f, T0, u, BDF, NEWTON, SS, &reltol,
                    &abstol, data, NULL, TRUE, iopt, ropt, machEnv);
if (cvmem == NULL) {
    printf("CVodeMalloc failed.");
    return(1);
}

/* CVSPGMR */
flag = CVSpGmr(cvmem, LEFT, MODIFIED_GS, 0, 0.0,
               Precond, PSolve, predata, NULL, NULL);
if (flag != SUCCESS) { printf("CVSpGmr failed.\n"); return(1); }

/* SENSITIVITY */
if(sensi) {
    pbar = (realtype *) malloc(NP*sizeof(realtype));
    for(is=0; is<NP; is++) pbar[is] = data->p[is];
    plist = (integertype *) malloc(NS * sizeof(integertype));
    for(is=0; is<NS; is++) plist[is] = is+1;

    uS = N_VNew_S(NS,neq,machEnv);
    for(is=0;is<NS;is++)
        N_VConst(ZERO,uS[is]);

    rhomax = ZERO;
    ifS = ALLSENS;
}

```

```

    if(sensi_meth==STAGGERED1) ifS = ONESENS;

    flag = CNodeSensMalloc(cvode_mem,NS,sensi_meth,data->p,pbar,plist,
                          ifS,NULL,err_con,rhox,uS,NULL,NULL);
    if (flag != SUCCESS) {
        if (my_pe == 0) printf("CNodeSensMalloc failed, flag=%d\n",flag);
        return(1);
    }
}

if (my_pe == 0) {
    printf("\n2-species diurnal advection-diffusion problem\n\n");
    printf("=====\n");
    printf("      T      Q      H      NST      Bottom left  Top right \n");
    printf("=====\n");
}

/* In loop over output points, call CNode, print results, test for error */
for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CNode(cvode_mem, tout, u, &t, NORMAL);
    if (flag != SUCCESS) {
        if (my_pe == 0) printf("CNode failed, flag=%d.\n", flag);
        break;
    }
    PrintOutput(my_pe, comm, iopt, ropt, t, u);
    if (sensi) {
        flag = CNodeSensExtract(cvode_mem, t, uS);
        if (flag != SUCCESS) {
            printf("CNodeSensExtract failed, flag=%d.\n", flag);
            break;
        }
        PrintOutputS(my_pe, comm, uS);
    }
    if (my_pe == 0)
        printf("-----\n");
}

/* Print final statistics */
if (my_pe == 0) PrintFinalStats(sensi,sensi_meth,err_con,iopt);

/* Free memory */
N_VFree(u);
if(sensi)
    N_VFree_S(NS, uS);
free(data->p);
free(data);
FreePreconData(predata);
CNodeFree(cvode_mem);
M_EnvFree_Parallel(machEnv);
MPI_Finalize();

```

```

    return(0);
}

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

/* ===== */
/* Exit if arguments are incorrect */

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

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

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

    pdata = (PreconData) malloc(sizeof *pdata);
    pdata->f_data = fdata;

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

    return(pdata);
}

/* ===== */
/* Load constants in data */

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

```

```

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

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

/* Set machine-related constants */
data->comm = comm;
data->my_pe = my_pe;
/* isubx and isuby are the PE grid indices corresponding to my_pe */
isuby = my_pe/NPEX;
isubx = my_pe - isuby*NPEX;
data->isubx = isubx;
data->isuby = isuby;
/* Set the sizes of a boundary x-line in u and uext */
data->nvmxsub = NVAR*MXSUB;
data->nvmxsub2 = NVAR*(MXSUB+2);

}

/* ===== */
/* Free data memory */

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

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

    free(pdata);
}

/* ===== */

```

```

/* Set initial conditions in u */

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

    /* Set pointer to data array in vector u */
    udata = NV_DATA_P(u);

    /* Get mesh spacings, and subgrid indices for this PE */
    dx = data->dx;      dy = data->dy;
    isubx = data->isubx; isuby = data->isuby;

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

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

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

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

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

```

```

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

/* On PE 0, receive c at top right, then print performance data
and sampled solution values */
if (my_pe == 0) {
    if (npelast != 0)
        MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);

    printf("%8.3e %2ld %8.3e %5ld\n", t, iopt[QU], ropt[HU], iopt[NST]);
    printf("                Solution                ");
    printf("%12.4e %12.4e \n", udata[0], tempu[0]);
    printf("                ");
    printf("%12.4e %12.4e \n", udata[1], tempu[1]);
}
}

/* ===== */
/* Print sampled sensitivity values */

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

    npelast = NPEX*NPEY - 1;

    sdata = NV_DATA_P(uS[0]);
    /* Send s1 at top right mesh point to PE 0 */
    if (my_pe == npelast) {
        i0 = NVAR*MXSUB*MYSUB - 2;
        i1 = i0 + 1;
        if (npelast != 0)
            MPI_Send(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
        else {
            temps[0] = sdata[i0];
            temps[1] = sdata[i1];
        }
    }
}

/* On PE 0, receive s1 at top right, then print sampled sensitivity values */

```



```

if (my_pe == 0) {
    if (npelast != 0)
        MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    printf("-----\n");
    printf("Sensitivity 1 ");
    printf("%12.4e %12.4e \n", sdata[0], temps[0]);
    printf(" ");
    printf("%12.4e %12.4e \n", sdata[1], temps[1]);
}

sdata = NV_DATA_P(uS[1]);
/* Send s2 at top right mesh point to PE 0 */
if (my_pe == npelast) {
    i0 = NVAR*MXSUB*MYSUB - 2;
    i1 = i0 + 1;
    if (npelast != 0)
        MPI_Send(&sdata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
        temps[0] = sdata[i0];
        temps[1] = sdata[i1];
    }
}

/* On PE 0, receive s2 at top right, then print sampled sensitivity values */
if (my_pe == 0) {
    if (npelast != 0)
        MPI_Recv(&temps[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    printf("-----\n");
    printf("Sensitivity 2 ");
    printf("%12.4e %12.4e \n", sdata[0], temps[0]);
    printf(" ");
    printf("%12.4e %12.4e \n", sdata[1], temps[1]);
}

}

/* ===== */
/* Print final statistics contained in iopt */

static void PrintFinalStats(boolean_type sensi, int sensi_meth, int err_con,
                           long int iopt[])
{
    printf("\n\n=====");
    printf("\nFinal Statistics");
    printf("\nSensitivity: ");

    if(sensi) {
        printf("YES ");
        if(sensi_meth == SIMULTANEOUS)
            printf("( SIMULTANEOUS +");
        else

```

```

        if(sensi_meth == STAGGERED) printf("( STAGGERED +");
        else                        printf("( STAGGERED1 +");
    if(err_con == FULL) printf(" FULL ERROR CONTROL ");
    else                  printf(" PARTIAL ERROR CONTROL ");
} else {
    printf("NO");
}

printf("\n\n");
printf("nst      = %5ld          \n\n", iopt[NST]);
printf("nfe      = %5ld      nfSe = %5ld \n", iopt[NFE], iopt[NFSE]);
printf("nni      = %5ld      nniS = %5ld \n", iopt[NNI], iopt[NNIS]);
printf("ncfn      = %5ld      ncfnS = %5ld \n", iopt[NCFN], iopt[NCFNS]);
printf("netf      = %5ld      netfS = %5ld\n\n", iopt[NETF], iopt[NETFS]);
printf("nsetups = %5ld          \n", iopt[NSETUPS]);
printf("nli      = %5ld      ncfl = %5ld \n", iopt[SPGMR_NLI], iopt[SPGMR_NCFL]);
printf("npe      = %5ld      nps  = %5ld \n", iopt[SPGMR_NPE], iopt[SPGMR_NPS]);

printf("=====\n");

}

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

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

    /* If isuby > 0, send data from bottom x-line of u */
    if (isuby != 0)
        MPI_Send(&udata[0], dsize, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

    /* If isuby < NPEY-1, send data from top x-line of u */
    if (isuby != NPEY-1) {
        offsetu = (MYSUB-1)*dsize;
        MPI_Send(&udata[offsetu], dsize, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
    }

    /* If isubx > 0, send data from left y-line of u (via bufleft) */
    if (isubx != 0) {
        for (ly = 0; ly < MYSUB; ly++) {
            offsetbuf = ly*NVARS;
            offsetu = ly*dsize;
            for (i = 0; i < NVARS; i++)
                bufleft[offsetbuf+i] = udata[offsetu+i];
        }
    }
}

```

```

    MPI_Send(&bufleft[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
}

/* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
if (isubx != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
        offsetbuf = ly*NVAR;
        offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVAR;
        for (i = 0; i < NVAR; i++)
            bufright[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufright[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
}

}

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

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

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

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

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

```

```

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

}

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

static void BRecvWait(MPI_Request request[], integertype isubx, integertype isuby,
                     integertype dsizey, realtype uext[], realtype buffer[])
{
    int i, ly;
    integertype dsizey2, offsetue, offsetbuf;
    realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
    MPI_Status status;

    dsizey2 = dsizey + 2*NVARs;

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

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

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

        /* Copy the buffer to uext */
        for (ly = 0; ly < MYSUB; ly++) {
            offsetbuf = ly*NVARs;
            offsetue = (ly+1)*dsizey2;
            for (i = 0; i < NVARs; i++)
                uext[offsetue+i] = bufleft[offsetbuf+i];
        }
    }

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

```

```

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

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

static void ucomm(integertype N, realtype t, N_Vector u, UserData data)
{
    realtype *udata, *uext, buffer[2*NVAR*MYSUB];
    MPI_Comm comm;
    integertype my_pe, isubx, isuby, nvxsub, nvmysub;
    MPI_Request request[4];

    udata = NV_DATA_P(u);

    /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
    comm = data->comm; my_pe = data->my_pe;
    isubx = data->isubx; isuby = data->isuby;
    nvxsub = data->nvxsub;
    nvmysub = NVAR*MYSUB;
    uext = data->uext;

    /* Start receiving boundary data from neighboring PEs */
    BRecvPost(comm, request, my_pe, isubx, isuby, nvxsub, nvmysub, uext, buffer);

    /* Send data from boundary of local grid to neighboring PEs */
    BSend(comm, my_pe, isubx, isuby, nvxsub, nvmysub, udata);

    /* Finish receiving boundary data from neighboring PEs */
    BRecvWait(request, isubx, isuby, nvxsub, uext, buffer);
}

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

static void fcalc(integertype N, realtype t, realtype udata[], realtype dudata[],
                  UserData data)
{
    realtype *uext;
    realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;

```

```

realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
realtype q4coef, dely, verdco, hordco, horaco;
int i, lx, ly, jx, jy;
integertype isubx, isuby, nvmxsub, nvmxsub2, offsetu, offsetue;
realtype Q1, Q2, C3, A3, A4, KH, VEL, KVO;

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

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

/* Copy local segment of u vector into the working extended array uext */
offsetu = 0;
offsetue = nvmxsub2 + NVARs;
for (ly = 0; ly < MYSUB; ly++) {
    for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
    offsetu = offsetu + nvmxsub;
    offsetue = offsetue + nvmxsub2;
}

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

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

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

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

```

```

    for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
}
}

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

/* Make local copies of problem variables, for efficiency */
dely = data->dy;
verdco = data->vdco;
hordco = data->hdco;
horaco = data->haco;

/* Set diurnal rate coefficients as functions of t, and save q4 in
data block for use by preconditioner evaluation routine */
s = sin((data->om)*t);
if (s > 0.0) {
    q3 = exp(-A3/s);
    q4coef = exp(-A4/s);
} else {
    q3 = 0.0;
    q4coef = 0.0;
}
data->q4 = q4coef;

/* Loop over all grid points in local subgrid */
for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    /* Set vertical diffusion coefficients at jy +/- 1/2 */
    ydn = YMIN + (jy - .5)*dely;
    yup = ydn + dely;
    cydn = verdco*exp(0.2*ydn);
    cyup = verdco*exp(0.2*yup);
    for (lx = 0; lx < MXSUB; lx++) {
        jx = lx + isubx*MXSUB;
        /* Extract c1 and c2, and set kinetic rate terms */
        offsetue = (lx+1)*NVAR + (ly+1)*nvmxsub2;
        c1 = uext[offsetue];
        c2 = uext[offsetue+1];
        qq1 = Q1*c1*C3;
        qq2 = Q2*c1*c2;
        qq3 = q3*C3;
        qq4 = q4coef*c2;
        rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
        rkin2 = qq1 - qq2 - qq4;
    }
}

```

```

        /* Set vertical diffusion terms */
        c1dn = uext[offsetue-nvmxsub2];
        c2dn = uext[offsetue-nvmxsub2+1];
        c1up = uext[offsetue+nvmxsub2];
        c2up = uext[offsetue+nvmxsub2+1];
        vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
        vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
        /* Set horizontal diffusion and advection terms */
        c1lt = uext[offsetue-2];
        c2lt = uext[offsetue-1];
        c1rt = uext[offsetue+2];
        c2rt = uext[offsetue+3];
        hord1 = hordco*(c1rt - 2.0*c1 + c1lt);
        hord2 = hordco*(c2rt - 2.0*c2 + c2lt);
        horad1 = horaco*(c1rt - c1lt);
        horad2 = horaco*(c2rt - c2lt);
        /* Load all terms into dudata */
        offsetu = lx*NVARs + ly*nvmxsub;
        dudata[offsetu] = vertd1 + hord1 + horad1 + rkin1;
        dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
    }
}

}

/***** Functions Called by the CVODES Solver *****/

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

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

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

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

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

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

static int Precond(integertype N, realtype tn, N_Vector u, N_Vector fu,

```



```

        booleantype jok, booleantype *jcurPtr,
        realtype gamma, N_Vector ewt, realtype h,
        realtype uration, long int *nfePtr, void *P_data,
        N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
    realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
    realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
    integertype nvmxsub, *(*pivot)[MYSUB], ier, offset;
    int lx, ly, jx, jy, isubx, isuby;
    realtype *udata, **a, **j;
    PreconData predata;
    UserData data;
    realtype Q1, Q2, C3, A3, A4, KH, VEL, KVO;

    /* Make local copies of pointers in P_data, pointer to u's data,
       and PE index pair */
    predata = (PreconData) P_data;
    data = (UserData) (predata->f_data);
    P = predata->P;
    Jbd = predata->Jbd;
    pivot = predata->pivot;
    udata = NV_DATA_P(u);
    isubx = data->isubx;    isuby = data->isuby;
    nvmxsub = data->nvmxsub;

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

    if (jok) { /* jok = TRUE: Copy Jbd to P */

        for (ly = 0; ly < MYSUB; ly++)
            for (lx = 0; lx < MXSUB; lx++)
                dencopy(Jbd[lx][ly], P[lx][ly], NVARs);
        *jcurPtr = FALSE;

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

        /* Make local copies of problem variables, for efficiency */
        q4coef = data->q4;
        dely = data->dy;
        verdco = data->vdco;
        hordco = data->hdco;

        /* Compute 2x2 diagonal Jacobian blocks (using q4 values

```

```

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

*jcurPtr = TRUE;

}

/* Scale by -gamma */
for (ly = 0; ly < MYSUB; ly++)
    for (lx = 0; lx < MXSUB; lx++)
        denscale(-gamma, P[lx][ly], NVARs);

/* Add identity matrix and do LU decompositions on blocks in place */
for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
        denaddI(P[lx][ly], NVARs);
        ier = gefa(P[lx][ly], NVARs, pivot[lx][ly]);
        if (ier != 0) return(1);
    }
}

return(0);
}

/* ===== */
/* Preconditioner solve routine */

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

```

```

{
    realtype **(*P)[MYSUB];
    integertype nvmxsub, *(*pivot)[MYSUB];
    int lx, ly;
    realtype *zdata, *v;
    PreconData predata;
    UserData data;

    /* Extract the P and pivot arrays from P_data */
    predata = (PreconData) P_data;
    data = (UserData) (predata->f_data);
    P = predata->P;
    pivot = predata->pivot;

    /* Solve the block-diagonal system Px = r using LU factors stored
       in P and pivot data in pivot, and return the solution in z.
       First copy vector r to z. */
    N_VScale(1.0, r, z);

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

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

    return(0);
}

```

## C Listings of CVODES Adjoint Sensitivity Examples

### C.1 A Serial Sample Problem - cvadx.c

```

/*****
 *
 * File      : cvadx.c
 * Programmers: Radu Serban @ LLNL
 * Version of : 22 March 2002
 *-----*
 * Adjoint sensitivity example problem.
 * The following is a simple example problem, with the coding
 * needed for its solution by CVODES. The problem is from chemical
 * kinetics, and consists of the following three rate equations..
 *   dy1/dt = -p1*y1 + p2*y2*y3
 *   dy2/dt =  p1*y1 - p2*y2*y3 - p3*(y2)^2
 *   dy3/dt =  p3*(y2)^2
 * on the interval from t = 0.0 to t = 4.e10, with initial conditions
 * y1 = 1.0, y2 = y3 = 0. The reaction rates are: p1=0.04, p2=1e4, and
 * p3=3e7. The problem is stiff.
 * This program solves the problem with the BDF method, Newton
 * iteration with the CVODE dense linear solver, and a user-supplied
 * Jacobian routine.
 * It uses a scalar relative tolerance and a vector absolute tolerance.
 * Output is printed in decades from t = .4 to t = 4.e10.
 * Run statistics (optional outputs) are printed at the end.
 *
 * Optionally, CVODES can compute sensitivities with respect to the
 * problem parameters p1, p2, and p3 of the following quantity:
 *   G = int_t0^t1 g(t,p,y) dt
 * where
 *   g(t,p,y) = y1 + p2 * y2 * y3
 *
 * The gradient dG/dp is obtained as:
 *   dG/dp = int_t0^t1 (g_p - lambda^T f_p ) dt - lambda^T(t0)*y0_p
 *           = - xi^T(t0) - lambda^T(t0)*y0_p
 * where lambda and xi are solutions of:
 *   d(lambda)/dt = - (f_y)^T * lambda + (g_y)^T
 *   lambda(t1) = 0
 * and
 *   d(xi)/dt = - (f_p)^T * lambda + (g_p)^T
 *   xi(t1) = 0
 *
 * During the backward integration, CVODES also evaluates G as
 *   G = - phi(t0)
 * where
 *   d(phi)/dt = g(t,y,p)
 *   phi(t1) = 0
 *
 *****/
```

```

#include <stdio.h>
#include <stdlib.h>
#include "sundialstypes.h"
#include "cvodea.h"
#include "cvsdense.h"
#include "nvector_serial.h"
#include "dense.h"

#define Ith(v,i)    NV_Ith_S(v,i-1)      /* Ith numbers components 1..NEQ */
#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* IJth numbers rows,cols 1..NEQ */

/* Problem Constants */
#define NEQ          3                  /* number of equations */
#define RTOL         1e-4              /* scalar relative tolerance */
#define ATOL1         1e-8              /* vector absolute tolerance components */
#define ATOL2         1e-14
#define ATOL3         1e-6

#define ATOL1         1e-5              /* absolute tolerance for adjoint vars. */
#define ATOLq         1e-6              /* absolute tolerance for quadratures */

#define TO           0.0                /* initial time */
#define TOUT          4e7               /* final time */

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

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

#define ZERO          0.0

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

/* Functions Called by the CVODES Solver */

/* f is of type RhsFn */
static void f(integertype N, realtype t, N_Vector y, N_Vector ydot, void *f_data);
/* Jac is of type CVDenseJacFn */
static void Jac(integertype N, DenseMat J, RhsFn f, void *f_data, realtype t,
                N_Vector y, N_Vector fy, N_Vector ewt, realtype h, realtype around,
                void *jac_data, long int *nfePtr, N_Vector vtemp1,
                N_Vector vtemp2, N_Vector vtemp3);
/* fB is of type RhsFnB */
static void fB(integertype NB, realtype t, N_Vector y,
                N_Vector yB, N_Vector yBdot, void *f_dataB);
/* JacB is of type CVDenseJacFnB */

```

```

static void JacB(integertype NB, DenseMat JB, RhsFnB fB, void *f_dataB, realtype t,
                N_Vector y, N_Vector yB, N_Vector fyB, N_Vector ewtB, realtype hB,
                realtype uroundB, void *jac_dataB, long int *nfePtrB, N_Vector vtemp1B,
                N_Vector vtemp2B, N_Vector vtemp3B);

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

int main(int argc, char *argv[])
{
    M_Env machEnvF, machEnvB;

    UserData data;

    void *cvadj_mem;
    void *cvode_mem;

    realtype reltol;
    N_Vector y, abstol;

    realtype reltolB;
    N_Vector yB, abstolB;

    realtype time;
    int flag, ncheck;

    /* USER DATA STRUCTURE */
    data = (UserData) malloc(sizeof *data);
    data->p[0] = 0.04;
    data->p[1] = 1.0e4;
    data->p[2] = 3.0e7;

    /* Initialize serial machine environment for forward integration */
    machEnvF = M_EnvInit_Serial(NEQ);

    /* Initialize y */
    y = N_VNew(NEQ, machEnvF);
    Ith(y,1) = 1.0;
    Ith(y,2) = 0.0;
    Ith(y,3) = 0.0;

    /* Set the scalar relative tolerance reltol */
    reltol = RTOL;
    /* Set the vector absolute tolerance abstol */
    abstol = N_VNew(NEQ, machEnvF);
    Ith(abstol,1) = ATOL1;
    Ith(abstol,2) = ATOL2;
    Ith(abstol,3) = ATOL3;

    /* Allocate CVODE memory for forward run */
    printf("\nAllocate CVODE memory for forward runs\n");

```

```

cnode_mem = CNodeMalloc(NEQ, f, T0, y, BDF, NEWTON, SV, &reltol, abstol,
                        data, NULL, FALSE, NULL, NULL, machEnvF);
flag = CVDense(cnode_mem, Jac, NULL);
if (flag != SUCCESS) { printf("CVDense failed.\n"); return(1); }

/* Allocate global memory */
printf("\nAllocate global memory\n");
cvadj_mem = CVadjMalloc(cnode_mem, STEPS);
if (cvadj_mem == NULL) { printf("CVadjMalloc failed.\n"); return(1); }

/* Perform forward run */
printf("\nForward integration\n");

flag = CNodeF(cvadj_mem, TOUT, y, &time, NORMAL, &ncheck);
if (flag != SUCCESS) { printf("CNodeF failed.\n"); return(1); }

/* Test check point linked list */
printf("\nList of Check Points (ncheck = %d)\n", ncheck);
CVadjCheckPointsList(cvadj_mem);

/* Initialize serial machine environment for backward run */
machEnvB = M_EnvInit_Serial(NEQ+NP+1);

/* Initialize yB */
yB = N_VNew(NEQ+NP+1, machEnvB);
Ith(yB,1) = 0.0;
Ith(yB,2) = 0.0;
Ith(yB,3) = 0.0;
Ith(yB,4) = 0.0;
Ith(yB,5) = 0.0;
Ith(yB,6) = 0.0;
Ith(yB,7) = 0.0;

/* Set the scalar relative tolerance reltolB */
reltolB = RTOL;
/* Set the vector absolute tolerance abstolB */
abstolB = N_VNew(NEQ+NP+1, machEnvB);
Ith(abstolB,1) = ATOLl;
Ith(abstolB,2) = ATOLl;
Ith(abstolB,3) = ATOLl;
Ith(abstolB,4) = ATOLq;
Ith(abstolB,5) = ATOLq;
Ith(abstolB,6) = ATOLq;
Ith(abstolB,7) = ATOLq;

/* Allocate CNODE memory for backward run */
printf("\nAllocate CNODE memory for backward run\n");
flag = CNodeMallocB(cvadj_mem, NEQ+NP+1, fB, yB, BDF, NEWTON, SV,
                    &reltolB, abstolB, data, NULL,
                    FALSE, NULL, NULL, machEnvB);
if (flag != SUCCESS) { printf("CNodeMallocB failed.\n"); return(1); }

```

```

flag = CVDenseB(cvadj_mem, JacB, NULL);
if (flag != SUCCESS) { printf("CVDenseB failed.\n"); return(1); }

/* Backward Integration */
flag = CNodeB(cvadj_mem, yB);
if (flag < 0) { printf("CNodeB failed.\n"); return(1); }

printf("\n\n=====\\n");
printf("G:          %12.4e \\n",-Ith(yB,7));
printf("Gp:          %12.4e %12.4e %12.4e\\n",
      -Ith(yB,4), -Ith(yB,5), -Ith(yB,6));
printf("=====\\n");
printf("lambda(t0): %12.4e %12.4e %12.4e\\n",
      Ith(yB,1), Ith(yB,2), Ith(yB,3));
printf("=====\\n");

/* Free memory */
printf("\\nFree memory\\n");
CNodeFree(cnode_mem);
CVadjFree(cvadj_mem);
N_VFree(y);
N_VFree(yB);
N_VFree(abstol);
N_VFree(abstolB);
free(data);
M_EnvFree_Serial(machEnvF);
M_EnvFree_Serial(machEnvB);

return(0);
}

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

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

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

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

```



```

/* Jacobian routine. Compute J(t,y). */
static void Jac(integertype N, DenseMat J, RhsFn f, void *f_data, realtype t,
               N_Vector y, N_Vector fy, N_Vector ewt, realtype h, realtype uround,
               void *jac_data, long int *nfePtr, N_Vector vtemp1,
               N_Vector vtemp2, N_Vector vtemp3)
{
    realtype y1, y2, y3;
    UserData data;
    realtype p1, p2, p3;

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

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

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

    data = (UserData) f_dataB;

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

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

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

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

    /* Load yBdot */
    Ith(yBdot,1) = - p1*l21 - 1.0;
    Ith(yBdot,2) = p2*y3*l21 - 2.0*p3*y2*l32 - p2*y3;
    Ith(yBdot,3) = p2*y2*l21 - p2*y2;
    Ith(yBdot,4) = y1*l21;
    Ith(yBdot,5) = - y23*l21 + y23;
}

```

```

    Ith(yBdot,6) = y2*y2*132;
    Ith(yBdot,7) = y1+p2*y23;
}

/* JacB routine. Compute JB(t,y,yB). */
static void JacB(integertype NB, DenseMat JB, RhsFnB fB, void *f_dataB, realtype t,
                N_Vector y, N_Vector yB, N_Vector fyB, N_Vector ewtB, realtype hB,
                realtype uroundB, void *jac_dataB, long int *nfePtrB, N_Vector vtemp1B,
                N_Vector vtemp2B, N_Vector vtemp3B)
{
    UserData data;
    realtype y1, y2, y3;
    realtype p1, p2, p3;

    data = (UserData) f_dataB;

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

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

    /* Load JB */
    IJth(JB,1,1) = p1;      IJth(JB,1,2) = -p1;
    IJth(JB,2,1) = -p2*y3; IJth(JB,2,2) = p2*y3+2.0*p3*y2; IJth(JB,2,3) = -2.0*p3*y2;
    IJth(JB,3,1) = -p2*y2; IJth(JB,3,2) = p2*y2;
    IJth(JB,4,1) = -y1;     IJth(JB,4,2) = y1;
    IJth(JB,5,1) = y2*y3;   IJth(JB,5,2) = -y2*y3;
                                IJth(JB,6,2) = -y2*y2;          IJth(JB,6,3) = y2*y2;
}

```

## C.2 A Parallel Sample Program - pvanx.c

```

/*****
*
* File      : pvanx.c
* Programmers: Radu Serban @ LLNL
* Version of : 21 May 2002
*-----*
* Example problem.
* The following is a simple example problem, with the program for its
* solution by CVODE. The problem is the semi-discrete form of the
* advection-diffusion equation in 1-D:
*    $du/dt = p1 * d^2u / dx^2 + p2 * du / dx$ 
* on the interval  $0 \leq x \leq 2$ , and the time interval  $0 \leq t \leq 5$ .
* Homogeneous Dirichlet boundary conditions are posed, and the
* initial condition is
*    $u(x,t=0) = x(2-x)\exp(2x)$  .
* The nominal values of the two parameters are:  $p1=1.0$ ,  $p2=0.5$ 
* The PDE is discretized on a uniform grid of size  $MX+2$  with
* central differencing, and with boundary values eliminated,
* leaving an ODE system of size  $NEQ = MX$ .
* This program solves the problem with the option for nonstiff systems:
* ADAMS method and functional iteration.
* It uses scalar relative and absolute tolerances.
*
* In addition to the solution, sensitivities with respect to  $p1$  and  $p2$ 
* as well as with respect to initial conditions are computed for the
* quantity:
*    $g(t, u, p) = \int_x u(x,t)$  at  $t = 5$ 
* These sensitivities are obtained by solving the adjoint system:
*    $dv/dt = -p1 * d^2 v / dx^2 + p2 * dv / dx$ 
* with homogeneous Dirichlet boundary conditions and the final
* condition
*    $v(x,t=5) = 1.0$ 
* Then,  $v(x, t=0)$  represents the sensitivity of  $g(5)$  with respect to
*  $u(x, t=0)$  and the gradient of  $g(5)$  with respect to  $p1, p2$  is
*    $(dg/dp)^T = [ \int_t \int_x (v * d^2u / dx^2) dx dt ]$ 
*                 $[ \int_t \int_x (v * du / dx) dx dt ]$ 
*
* This version uses MPI for user routines.
* Execute with Number of Processors = N, with  $1 \leq N \leq MX$ .
*****/

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

```

```

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

/* Adjoint Problem Constants */
#define NP 2              /* number of parameters    */
#define STEPS 100         /* steps between check points */

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

/* Private Helper Functions */
static void SetIC(N_Vector u, realtype dx, integertype my_length,
                 integertype my_base);
static void SetICback(N_Vector uB, integertype my_base);
static realtype Xintgr(realtype *z, integertype l, realtype dx);
static realtype Compute_g(N_Vector u, UserData data);

/* Functions Called by the CVODES Solver */
static void f(integertype N, realtype t, N_Vector u, N_Vector udot, void *f_data);
static void fB(integertype NB, realtype t, N_Vector u,
               N_Vector uB, N_Vector uBdot, void *f_dataB);

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

int main(int argc, char *argv[])
{
    M_Env machEnvF, machEnvB;

    UserData data;

    void *cvadj_mem;
    void *cvmem;

    long int iopt[OPT_SIZE];
    realtype ropt[OPT_SIZE];
    N_Vector u;
    realtype reltol, abstol;

```

```

N_Vector uB;
realtype *uBdata;

realtype dx, t, umax, g_val;
int flag, my_pe, nprocs, npes, ncheck;
integertype local_N, nperpe, nrem, my_base, i, iglobal;

MPI_Comm comm;

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

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

/* Allocate and load user data structure */
data = (UserData) malloc(sizeof *data);
data->p[0] = 1.0;
data->p[1] = 0.5;
dx = data->dx = XMAX/((realtype)(MX+1));
data->hdcoef = data->p[0]/(dx*dx);
data->hacoef = data->p[1]/(2.0*dx);
data->comm = comm;
data->npes = npes;
data->my_pe = my_pe;

/* Set local vector length. */
if (my_pe < npes) {
    nperpe = NEQ/npes;
    nrem = NEQ - npes*nperpe;
    local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
    my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;
}

/*-----
FORWARD INTEGRATION PHASE
-----*/

/* Make last process inactive for forward phase */
if (my_pe == npes) local_N = 0;

/* Initialize machine environment for forward phase */
machEnvF = M_EnvInit_Parallel(comm, local_N, NEQ, &argc, &argv);
if (machEnvF == NULL) {
    if(my_pe == 0) printf("M_EnvInit_Parallel failed.\n");
    return(1);
}

```

```

/* Set relative and absolute tolerances for forward phase */
reltol = 0.0;
abstol = ATOL;

/* Allocate and initialize forward variables */
u = N_VNew(NEQ, machEnvF);
SetIC(u, dx, local_N, my_base);

/* Allocate CVODES memory for forward integration */
cnode_mem = CNodeMalloc(NEQ, f, T0, u, ADAMS, FUNCTIONAL, SS, &reltol,
                        &abstol, data, NULL, FALSE, iopt, ropt, machEnvF);
if (cnode_mem == NULL) {
    if(my_pe == 0) printf("CNodeMalloc failed.\n");
    return(1);
}

/* Allocate combined forward/backward memory */
cvadj_mem = CVadjMalloc(cnode_mem, STEPS);

/* Integrate to TOUT and collect check point information */
flag = CNodeF(cvadj_mem, TOUT, u, &t, NORMAL, &ncheck);
if (flag != SUCCESS) {
    if(my_pe == 0) printf("CNode failed, flag=%d.\n", flag);
    return(1);
}

/* Compute and print value of g(5) */
g_val = Compute_g(u, data);
if(my_pe == npes) {
    printf("\n (PE# %d)\n", my_pe);
    printf("    g(t1) = %g\n", g_val);
}

/*-----
   BACKWARD INTEGRATION PHASE
   -----*/

/* Allocate work space */
data->z1 = (realtype *)malloc(local_N*sizeof(realtype));
data->z2 = (realtype *)malloc(local_N*sizeof(realtype));

/* Activate last process for integration of the quadrature equations */
if(my_pe == npes) local_N = NP;

/* Initialize machine environment for backward phase */
machEnvB = M_EnvInit_Parallel(comm, local_N, NEQ+NP, &argc, &argv);
if (machEnvB == NULL) {
    if(my_pe == 0) printf("M_EnvInit_Parallel failed.\n");
    return(1);
}

```

```

/* Allocate and initialize backward variables */
uB = N_VNew(NEQ+NP, machEnvB);
SetICback(uB, my_base);

/* Allocate CVODES memory for the backward integration */
flag = CVodeMallocB(cvadj_mem, NEQ+NP, fB, uB, ADAMS, FUNCTIONAL, SS, &reltol,
                  &abstol, data, NULL, FALSE, NULL, NULL, machEnvB);
if (flag != SUCCESS) {
    if(my_pe == 0) printf("CVodeMallocB failed, flag=%d.\n", flag);
    return(1);
}

/* Integrate to T0 */
flag = CVodeB(cvadj_mem, uB);
if (flag < 0) {
    if(my_pe == 0) printf("CVodeB failed, flag=%d.\n", flag);
    return(1);
}

/* Print results (adjoint states and quadrature variables) */
uBdata = NV_DATA_P(uB);
printf("\n (PE# %d)\n", my_pe);
if (my_pe == npes) {
    printf("    dgdp(t1) = [ %g %g ]\n", -uBdata[0], -uBdata[1]);
} else {
    for (i=1; i<=local_N; i++) {
        iglobal = my_base + i;
        printf("    mu(t0)[%2d] = %g\n", iglobal, uBdata[i-1]);
    }
}

/* Clean-Up */
N_VFree(u);
N_VFree(uB);
CVodeFree(cvode_mem);
CVadjFree(cvadj_mem);
free(data->z1); free(data->z2); free(data);
M_EnvFree_Parallel(machEnvF);
M_EnvFree_Parallel(machEnvB);

/* Finalize MPI */
MPI_Finalize();

return(0);
}

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

/* Set initial conditions in u vector */
static void SetIC(N_Vector u, realtype dx, integertype my_length,

```

```

        integertype my_base)
{
    int i;
    integertype iglobal;
    realtype x;
    realtype *udata;

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

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

/* Set final conditions in uB vector */
static void SetICback(N_Vector uB, integertype my_base)
{
    int i;
    realtype *uBdata;
    integertype my_length;

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

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

/* Compute local value of the space integral  $\int_x z(x) dx$  */
static realtype Xintgr(realtype *z, integertype l, realtype dx)
{
    realtype my_intgr;
    integertype i;

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

    return(my_intgr);
}

/* Compute value of  $g(u)$  */
static realtype Compute_g(N_Vector u, UserData data)

```



```

{
    realtype integr, my_intgr, dx, *udata;
    integertype my_length;
    int npes, my_pe, i;
    MPI_Status status;
    MPI_Comm comm;

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

    dx = data->dx;

    if (my_pe == npes) { /* Loop over all other processes and sum */
        integr = 0.0;
        for (i=0; i<npes; i++) {
            MPI_Recv(&my_intgr, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
            integr += my_intgr;
        }
        return(integr);
    } else { /* Compute local portion of the integral */
        udata = NV_DATA_P(u);
        my_length = NV_LOCLENGTH_P(u);
        my_intgr = Xintgr(udata, my_length, dx);
        MPI_Send(&my_intgr, 1, PVEC_REAL_MPI_TYPE, npes, 0, comm);
        return(my_intgr);
    }
}

/***** Function Called by the CVODE Solver *****/

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

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

    /* If this process is inactive, return now */

```

```

if (my_pe == npes) return;

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

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

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

/* Pass needed data to processes before and after current process. */
if (my_pe != 0)
    MPI_Send(&udata[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
if (my_pe != last_pe)
    MPI_Send(&udata[my_length-1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);

/* Receive needed data from processes before and after current process. */
if (my_pe != 0)
    MPI_Recv(&uLeft, 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
else uLeft = 0.0;
if (my_pe != last_pe)
    MPI_Recv(&uRight, 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
            &status);
else uRight = 0.0;

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

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

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

/* fB routine. Compute right hand side of backward problem */
static void fB(integertype NB, realtype t, N_Vector u,
              N_Vector uB, N_Vector uBdot, void *f_dataB)
{

```

```

realtype *uBdata, *duBdata, *udata, *zB;
realtype uBLeft, uBRight, uBi, uBlt, uBrt;
realtype uLeft, uRight, ui, ult, urt;
realtype dx, hordc, horac, hdiff, hadv;
realtype *z1, *z2, intgr1, intgr2;
integertype i, my_length;
int npes, my_pe, my_pe_m1, my_pe_p1, last_pe, my_last;
UserData data;
MPI_Status status;
MPI_Comm comm;

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

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

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

    /* Loop over all other processes and load right hand side of quadrature eqs. */
    duBdata[0] = 0.0;
    duBdata[1] = 0.0;
    for (i=0; i<npes; i++) {
        MPI_Recv(&intgr1, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
        duBdata[0] += intgr1;
        MPI_Recv(&intgr2, 1, PVEC_REAL_MPI_TYPE, i, 0, comm, &status);
        duBdata[1] += intgr2;
    }

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

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

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

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

```

```

udata = NV_DATA_P(u);
my_length = NV_LOCLENGTH_P(uB);

/* Pass needed data to processes before and after current process. */
if (my_pe != 0) {
    MPI_Send(&udata[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
    MPI_Send(&uBdata[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
}
if (my_pe != last_pe) {
    MPI_Send(&udata[my_length-1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
    MPI_Send(&uBdata[my_length-1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
}

/* Receive needed data from processes before and after current process. */
if (my_pe != 0) {
    MPI_Recv(&uLeft, 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
    MPI_Recv(&uBLeft, 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
} else {
    uLeft = 0.0;
    uBLeft = 0.0;
}
if (my_pe != last_pe) {
    MPI_Recv(&uRight, 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm, &status);
    MPI_Recv(&uBRight, 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm, &status);
} else {
    uRight = 0.0;
    uBRight = 0.0;
}

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

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

    /* Set diffusion and advection terms and load into udot */
    hdiff = hordc*(uBlt - 2.0*uBi + uBrt);
    hadv = horac*(uBrt - uBlt);
    duBdata[i] = - hdiff + hadv;

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

    /* Load integrands of the two space integrals */
    z1[i] = uBdata[i]*(ult - 2.0*ui + urt)/(dx*dx);
    z2[i] = uBdata[i]*(urt - ult)/(2.0*dx);
}

```

```

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

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

```