CVODE User Guide

Scott D. Cohen   Alan C. Hindmarsh

October 5, 1994

Contents

List of Figures

List of Tables

1 Introduction

1.1 Historical Background

CVODE is a solver for stiff and nonstiff initial value problems for systems of ordinary differential equation (ODEs). It is written in C, and based on two older ODE solvers written in Fortran.

Fortran solvers for ODE initial value problems are widespread and heavily used. Two solvers that have been written at LLNL in recent years 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 [4]. 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.

In addition to translating the VODE and VODPK algorithms into C, we have reorganized the overall algorithm 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 will facilitate the planned extension of CVODE to multiprocessor environments with minimal impacts on the rest of the solver.

There are several motivations for choosing the C language for CVODE. 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 CVODE 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 Mathematical Setting

An ODE initial value problem can be written as

$$\dot{y} = f(t, y), \qquad y(t_0) = y_0, \qquad y \in \mathbf{R}^N, \tag{1}$$

where $\dot{y}$ denotes the derivative dy/dt. This problem is stiff if it contains one or more strongly damped modes (i.e. the Jacobian matrix $J = \partial f / \partial y$ has an eigenvalue with large negative real part). The underlying integration methods used in CVODE are variable-coefficient forms of the Adams and BDF (Backward Differentiation Formula) methods. The numerical solution to (1) is generated as discrete values y_n at time points t_n. The computed values y_n obey a linear multistep formula

$$\sum_{i=0}^{K_1} \alpha_{n,i} y_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}_{n-i} = 0. \tag{2}$$

Here the $\dot{y}_n$ are computed approximations to $\dot{y}(t_n)$, $h_n = t_n - t_{n-1}$ is the stepsize, and $\alpha_{n,0} = -1$. For use on nonstiff problems, the Adams-Moulton formula is characterized by $K_1 = 1$ and $K_2 = q$, and the order q varies between 1 and 12. For stiff problems, the BDF formula has $K_1 = q$ and $K_2 = 0$, and the order q varies between 1 and 5. In either case, the nonlinear system

$$G(y_n) \equiv y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0 \ , \quad \text{where} \quad a_n \equiv \sum_{i>0} (\alpha_{n,i} y_{n-i} + h_n \beta_{n,i} \dot{y}_{n-i}), \tag{3}$$

must be solved (approximately) at each time step. In the nonstiff case, this is usually done with simple functional (or fixed point) iteration. In the stiff case, the solution of (3) is usually done with some variant of Newton iteration. This requires the solution of linear systems of the form

$$M[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}), \tag{4}$$

where M is an approximation to the Newton matrix $I - h\beta_{n,0} J$ and $J = \partial f / \partial y$ is the ODE system Jacobian. The linear systems arising in the Newton iteration are solved by either a direct or an iterative method. The direct solvers currently available in CVODE include dense (full) and banded methods, and a diagonal approximate Jacobian method. The iterative method is the *Generalized Minimal Residual* method [5], or GMRES. With scaling and preconditioning included, we refer to this method as SPGMR.

The estimation and control of errors is an extremely important feature of any ODE solver. In CVODE, local tructation errors in the computed values of y are estimated, and the solver will control the vector e of estimated local errors in accordance with a combination of input relative and absolute tolerances. Specifically, the vector e is made to satisfy an inequality of the form

$$\|e\|_{WRMS,ewt} \le 1, \tag{5}$$

where the weighted root-mean-square norm $\|v\|_{WRMS,w}$ with weight vector w is defined as

$$\|v\|_{WRMS,w} = \sqrt{\frac{\sum_{i=1}^{N} (v_i w_i)^2}{N}}.$$

The CVODE error weight vector ewt has components

$$ewt_i = \frac{1}{RTOL \cdot |y_i| + ATOL_i},$$

where the non-negative relative and absolute tolerances $RTOL$ and $ATOL$ are specified by the user. Here $RTOL$ is a scalar, but $ATOL$ can be either a scalar or a vector. The local error test (5) controls the estimated local error e_i in component y_i in such a way that, roughly, $|e_i|$ will be less than $ewt_i^{-1} = RTOL \cdot |y_i| + ATOL_i$. The local error test passes if in each component the absolute error $|e_i|$ is less than or equal to $ATOL_i$, or the relative error $|e_i|/|y_i|$ is less than or equal to $RTOL$. Use $RTOL = 0$ for pure absolute error control, and $ATOL_i = 0$ for pure relative error control. Actual (global) errors, being an accumulation of local errors, may well exceed these local tolerances, so choose $RTOL$ and $ATOL$ conservatively.

1.3 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.

In Section 2, we begin with a short overview of the usage of CVODE. Readers can then either read the more detailed usage description that follows, or skip to the example programs.

The subsequent sections contain three sample programs that illustrate the usage of CVODE with different choices of the linear system solvers. The examples include the use of the dense solver, the band solver, and the SPGMR solver, in Sections 3, 4, and 5, respectively. Each of these sections is self-contained. 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 CVODE by example, and each can serve as a template for user programs.

Section 6 describes in more detail the call that the user must make to select a linear system solver, and gives a brief summary of the CVODE linear solvers currently available. Section 7 describes the integer and real data types used in CVODE, and how those can be altered to accomodate different precisions.

Section 8 describes a pair of "demonstration programs" that are also provided with the package. These can be viewed as illustrations of the use of CVODE, but, more importantly, provide a means of checking the validity of the installed package.

In Section 9 we give a brief description of the overall organization of the CVODE package, together with short descriptions of the files in it.

Most of this user guide is taken up with the reference guide given in Section 10. It contains full documentation for all functions, macros, types, enumerations, etc. that the user needs to use CVODE. Keep the reference guide handy when writing or modifying your code, so that you can easily look up things like exact function names, and the meaning and ordering of parameters. Throughout the user guide, the reader is referred to sections in the reference guide for "full details" on the usage of CVODE.

Finally, the reader should be aware of a couple of notational issues in this user guide. Program listings and identifiers (such as `CVodeMalloc`) within textual explanations appear in typewriter type style, and packages or modules, such as CVDENSE, are written in all capitals. The one exception is the CVode main integrator module (CVODE is reserved for the name of the entire software package). Thus, for example, we write the identifier `CVode` for the function that is part of the CVode integrator module within the CVODE software package.

2 Usage of CVODE

In this section we describe the usage of CVODE at two levels. First we give a brief overview. This overview is sufficient for an understanding of the sample programs given in the following sections, and therefore the reader may want to skip to those sections. The remainder of this section is a more detailed summary of usage. Even that description omits some of the usage details. However, both in this section and in the descriptions of the sample programs, we refer to later sections for complete information on CVODE usage.

2.1 Overview

A high-level view of the combined user program and CVODE package is shown in Figure 1. It shows the user's program and the various CVODE modules. The user's program has the following essential parts:

- `#include` lines to access CVODE header files;
- allocation and loading of CVODE inputs, including the input `y` of type `N_Vector` (allocated with `N_VNew`);
- a call to `CVodeMalloc` to allocate memory for CVODE;
- if Newton iteration is selected, a call to one of `CVDense`, `CVBand`, `CVDiag`, or `CVSpgmr`, to specify which linear system module is to be used;
- a loop of calls to `CVode` for integration to prescribed output points;
- a call to `CVodeFree` to free all CVODE memory;
- freeing of user-allocated memory when the integration is done, e.g. freeing `y` with `N_VFree`;
- the user-supplied function `f` defining $f(t, y)$; and
- either an optional user-supplied function `Jac` for a Jacobian approximation or a pair of user-supplied routines, `Precond` and `PSolve`, for preconditioning of the Krylov method.

This list includes the user-supplied functions that define the ODE problem and its Jacobian (or approximation, as needed), and it includes calls to certain CVODE functions to carry out the solution. The list also includes items that deal with two other aspects of the CVODE user interface: One is the selection of a linear system solver for use in the stiff case. The other deals with CVODE vector; all vectors of length N, such as y, are communicated as objects of type `N_Vector`, a type defined with the CVODE package.

With this overview, the reader may find it more instructive at this point to skip to the sample programs in the following three sections. There most of the necessary usage details not shown above are covered in the programs and the accompanying explanations. However, for the sake of completeness, we give also the following summary of the general usage instructions.

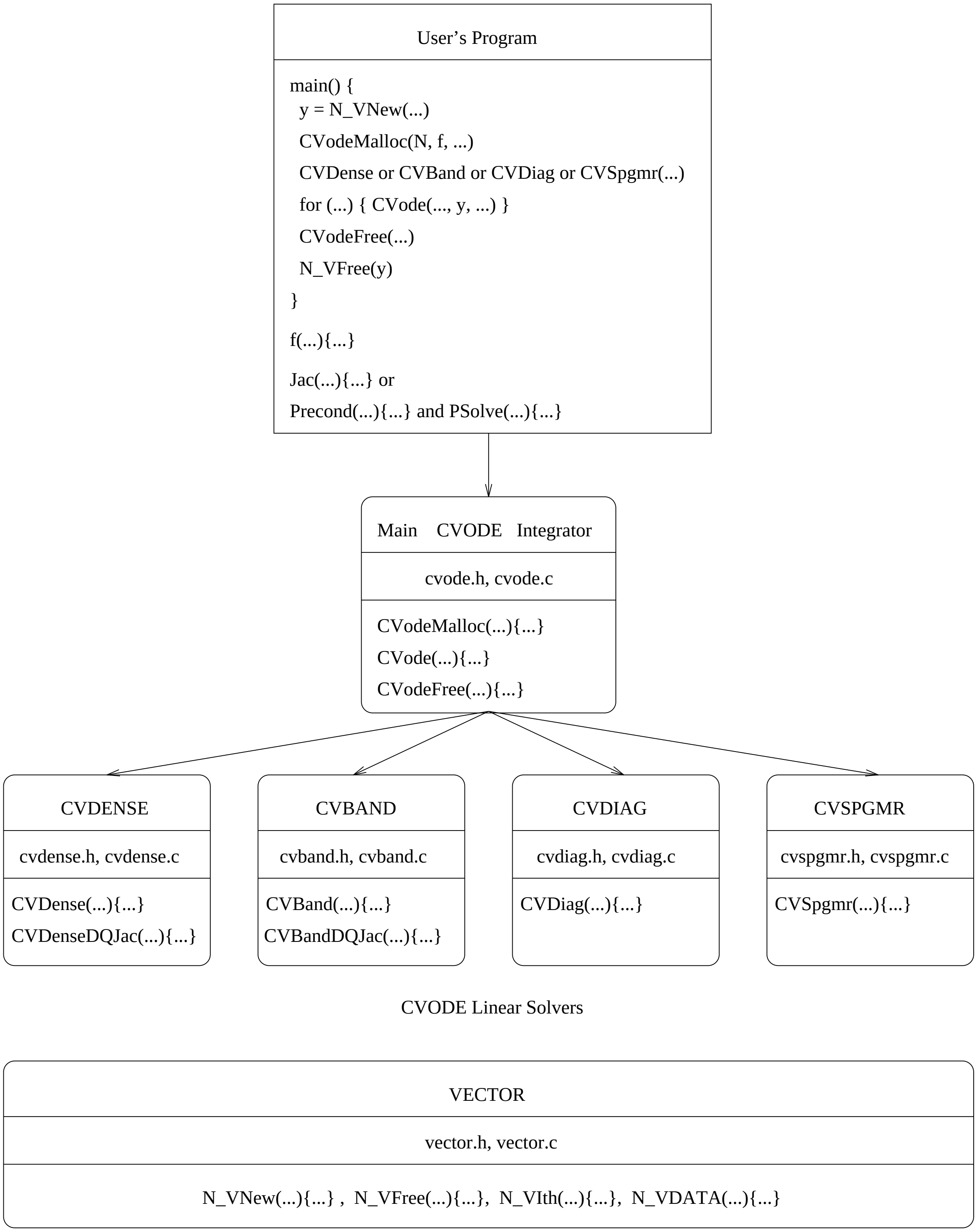

Figure 1: Diagram of the user program and CVODE package

2.2 Usage Summary

A program which uses CVODE must include certain CVODE header files, make calls to the CVODE solver functions, and define the problem through user-supplied functions. We summarize these features in turn, below.

To begin with, the calling program should also contain the following `#include` lines, in order to gain access to various constants, macros, and function prototypes:

```
#include "llnltyps.h" /* for types real, integer, bool; constants TRUE, FALSE */
#include "llnlmath.h" /* for power functions, MIN, MAX, ABS, SQR, RSqrt */
#include "cvode.h"    /* for function prototypes and constants */
#include "vector.h"   /* for definitions of N_Vector type and macros */
```

For details on the types `real` and `integer`, see Section 7, and for detailed contents of `vector.h`, see Section 10.3.1. For documentation of the functions in the LLNLMATH module, see the file `llnlmath.h`.

In addition, the calling program should contain `#include` lines for one or more header files associated with the linear system solver selected, namely:

```
#include "cvdense.h" /* for the CVDENSE case */
#include "cvband.h"  /* for the CVBAND case  */
#include "cvdiag.h"  /* for the CVDIAG case  */
#include "cvspgmr.h" /* for the CVSPGMR case */
#include "dense.h"   /* for use of the generic dense matrix routines */
#include "band.h"    /* for use of the generic band matrix routines  */
```

Generic solver modules called DENSE, BAND, and SPGMR may also need to be accessed by the CVODE user. Details on these generic solvers appear in Sections 10.3.2, 10.3.3, and 10.3.4, respectively.

The calling program itself must call four different functions (three if functional iteration is chosen instead of Newton). These are summarized as follows:

- `CVodeMalloc` allocates memory for the main CVODE integrator, initializes certain data, and returns a pointer to the CVODE memory structure;
- If Newton iteration is selected, a call to `CV__` specifies which linear system module is to be used, where `CV__` is one of `CVDense`, `CVBand`, `CVDiag`, or `CVSpgmr` (referring to the modules CVDENSE, CVBAND, CVDIAG, and CVSPGMR, respectively);
- `CVode` is called for each point at which output is desired;
- `CVodeFree` frees all memory that was allocated by `CVodeMalloc`.

Specification of inputs that are independent of the CVODE linear system solver is done through `CVodeMalloc`. The call has the form

```
cvode_mem = CVodeMalloc(N, f, t0, y0, lmm, iter, itol, &reltol, abstol,
                        f_data, errfp, optIn, iopt, ropt, NULL);
```

These inputs include the problem size N, the function defining f, the initial conditions t_0 and y_0, the choices of linear multistep method (Adams or BDF) for the integration and of iteration method (functional or Newton), and three tolerance parameters. The tolerance type `itol` is either `SS`, to indicate scalar relative and absolute tolerances, or `SV`, to indicate a scalar relative tolerance and a vector of absolute tolerances. In the call above, the `abstol` argument is an `N_Vector` in the `SV` case, and a pointer to a real scalar in the `SS` case. Following the tolerances in the call list are: a pointer to user data (described below), an error file pointer `errfp` (pass `NULL` to specify standard output), an optional input flag `optIn` (`FALSE` indicates no optional inputs), and two arrays `iopt` and `ropt` for integer and real optional inputs and outputs (discussed below). The final `NULL` argument is for a machine environment agrument not used in the sequential version of CVODE. The arguments to `CVodeMalloc` are fully described in Sections 10.1.1-10.1.3.

The second call above, to `CV__`, selects the linear solver to be used in connection with Newton iteration, and also supplies user inputs associated with that linear solver. The `CV__` routine also does memory allocation specific to the particular linear solver. The call takes one of the following four forms:

```
CVDense(cvode_mem, Jac, jac_data);
CVBand(cvode_mem, mupper, mlower, Jac, jac_data);
CVDiag(cvode_mem);
CVSpgmr(cvode_mem, pretype, gstype, maxl, delt, Precond, PSolve, P_data);
```

In the direct CVDENSE and CVBAND cases, inputs in this call include the user's Jacobian function (if present), and bandwidths in the band case. In the CVSPGMR case, they include the user's preconditioner setup and solve functions, and a flag `pretype` indicating whether preconditioning is on the left, on the right, on both sides, or absent. These `CV__` calls are more fully described in Section 6.

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

```
flag = CVode(cvode_mem, tout, yout, &t, itask);
```

In addition to the CVODE 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 $t =$`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 `yout` and a corresponding independent variable value $t =$`*t`, such that `yout` is the computed value of $y(t)$. In the normal mode, with no failures, `*t` will be equal to `tout`. For a detailed description of the `CVode` call and return values, see Section 10.1.4.

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 CVODE linear solver may or may not have optional inputs, which are passed through the associated `CV__` 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 CVODE 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 Section 10.1.7.

In addition to a main or calling program containing the sequence of calls described above, the user must always supply a function `f` that defines the right-hand side $f(t, y)$. (For its detailed specification, see Section 10.1.2.) In the direct CVDENSE and CVBAND cases, an optional Jacobian function `Jac` may be supplied. If it is not, a difference quotient approximation is used. In the CVSPGMR case, a pair of functions, `Precond` and `PSolve`, are supplied if the GMRES iteration is to be preconditioned. The `Precond` function may be absent if no setup operations are involved in the preconditioning. For details, see Section 6.

Typically, a user application involves data that is defined by the user and is to be shared between the calling program and the user-supplied functions. This is handled with pointers supplied by the user in the various calls. A pointer to a user-defined space called `f_data` is passed to `CVodeMalloc`, and this pointer is then passed in all calls to the user's function `f`, for use there as desired. Similarly, in the CVDENSE or CVBAND case, a pointer to user data (which can be identical to `f_data`) associated with the `Jac` routine is passed to `CVDense` or `CVBand` and then passed to `Jac`, and in the CVSPGMR case a pointer for preconditioning data is passed to `CVSpgmr` and then to `Precond` and `PSolve`. In this way, any data required by the user-supplied routines can be stored in a manner determined entirely by the user and retrieved as needed, and the user's program need not have any global data.

Error conditions are reported by the CVODE package through error messages. These are sent to standard output, or to a file specified by the user in the call to `CVodeMalloc` if desired. Error messages may be issued by `CVodeMalloc` in the course of its checking of all input, by the linear solver function `CV__` in its input checking, or by `CVode` from illegal input or following a numerical failure (e.g. in Newton iteration convergence or the error test).

The memory storage requirements of the CVODE package can be broken into three categories: fixed requirements, problem-dependent storage for the central integrator CVode, and problem-dependent storage for the linear solver. The problem-dependent storage for CVode consists of $N * (maxord + 5)$ real words, where N is the size of the ODE system and $maxord$ is the maximum method order. For the default value of $maxord$, i.e. 12 for the Adams method and 5 for the BDF method, this requirement is $17N$ or $10N$ real words. The problem-dependent storage requirements for the linear solvers are given in Section 6. The actual lengths of the various problem-dependent workspaces are also provided as optional outputs.

An optionally callable function `CVodeDky` is available to obtain additional output values. This function 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 CVODE. See Section 10.1.5 for details.

3 Dense Solver Sample Program

As an initial illustration of the use of the CVODE package, the following is a sample program provided as part of the package. It uses the CVODE dense linear solver module CVDENSE in the solution of a small chemical kinetics problem. Following the source, we give a rather detailed explanation of the parts of the program and their interaction with CVODE.

```
/************************************************************************
 *                                                                      *
 * File: cvdx.c                                                         *
 * Programmers: Scott D. Cohen and Alan C. Hindmarsh @ LLNL             *
 * Version of 1 September 1994                                          *
 *----------------------------------------------------------------------*
 * 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 "llnltyps.h" /* definitions of types real (set to double) and     */
                      /* integer (set to int), and the constant FALSE      */
#include "cvode.h"    /* prototypes for CVodeMalloc, CVode, and CVodeFree, */
                      /* constants OPT_SIZE, BDF, NEWTON, SV, SUCCESS,     */
                      /* NST, NFE, NSETUPS, NNI, NCFN, NETF                */
#include "cvdense.h"  /* prototype for CVDense, constant DENSE_NJE         */
#include "vector.h"   /* definitions of type N_Vector and macro N_VIth,    */
                      /* 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 vector.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)    N_VIth(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(integer N, real t, N_Vector y, N_Vector ydot, void *f_data);

static void Jac(integer N, DenseMat J, RhsFn f, void *f_data, real t,
                N_Vector y, N_Vector fy, N_Vector ewt, real h, real uround,
                void *jac_data, long int *nfePtr, N_Vector vtemp1,
                N_Vector vtemp2, N_Vector vtemp3);


/***************************** Main Program ******************************/

main()
{
  real ropt[OPT_SIZE], reltol, t, tout;
  long int iopt[OPT_SIZE];
  N_Vector y, abstol;
  void *cvode_mem;
  int iout, flag;

  y = N_VNew(NEQ, NULL);       /* Allocate y, abstol vectors */
  abstol = N_VNew(NEQ, NULL);

  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 CVodeMalloc to initialize CVODE:

     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 integer input and output
     ropt    is an array used to communicate optional real input and output

     A pointer to CVODE problem memory is returned and stored in cvode_mem. */

  cvode_mem = CVodeMalloc(NEQ, f, T0, y, BDF, NEWTON, SV, &reltol, abstol,
                          NULL, NULL, FALSE, iopt, ropt, NULL);
  if (cvode_mem == NULL) { printf("CVodeMalloc failed.\n"); return(1); }

  /* Call CVDense to specify the CVODE dense linear solver with the
     user-supplied Jacobian routine Jac. */

  CVDense(cvode_mem, Jac, NULL);

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

  printf(" \n3-species kinetics problem\n\n");
  for (iout=1, tout=T1; iout <= NOUT; iout++, tout *= TMULT) {
    flag = CVode(cvode_mem, 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 */
  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 ncfn = %-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(integer N, real t, N_Vector y, N_Vector ydot, void *f_data)
{
  real 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(integer N, DenseMat J, RhsFn f, void *f_data, real t,
                N_Vector y, N_Vector fy, N_Vector ewt, real h, real uround,
                void *jac_data, long int *nfePtr, N_Vector vtemp1,
                N_Vector vtemp2, N_Vector vtemp3)
{
  real 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;
}
```

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in CVODE header files. The `llnltyps.h` file provides the definitions of the types `real` and `integer`; see Section 7 for details. For now, it suffices to read `real` as `double` and `integer` as `int`. The `cvode.h` file provides prototypes for the three CVODE 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. See Section 10.1 for details. The `cvdense.h` file provides the prototype for the `CVDense` function, and a constant `DENSE_NJE` for accessing optional output specific to CVDENSE. See Section 10.2.1 for details. The `vector.h` file includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the vector-related memory allocation and freeing functions. See Section 10.3.1 for details. Finally, the `dense.h` file provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements. See Section 10.3.2 for details. 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`. It is defined using the CVODE accessor macro `N_VIth` 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 CVODE accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The CVODE macros `N_VIth` and `DENSE_ELEM` are fully described in Sections 10.3.1.2 and 10.3.2.2, 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 CVODE.

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 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 dependendent variable vector into y and set the absolute tolerance vector abstol using the Ith macro. (The NULL argument is not relevant here.) See Section 10.3.1.3.1 for details on N_VNew.

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 three 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 CVODE error messages to be sent to standard output; a file pointer (of type FILE *) may be given in this position otherwise. The last NULL parameter is passed for the CVodeMalloc formal parameter machEnv. This is intended for use in parallel versions of CVODE which are currently being developed. For now, the user should always pass NULL for machEnv.

The return value of CVodeMalloc is a pointer to a CVODE solver memory block for the problem and solver options specified by the inputs. In the case if failure, the return value is NULL. This pointer must be passed in the remaining calls to CVODE functions. See Section 10.1.3 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 Section 10.2.1.1 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 $t =$tout, putting the computed value of $y($tout$)$ into y. The program prints t and y, and tests for a return value other than SUCCESS by CVode. See Section 10.1.4 for full details of the call to CVode.

Finally, the main program calls N_VFree to free the vectors y and abstol, calls CVodeFree to free the CVODE memory block, and prints all of the statistical quantities in the private helper function PrintFinalStats. See Section 10.3.1.3.1 for details on N_VFree, and Section 10.1.6 for details on CVodeFree.

The function PrintFinalStats used here is actually suitable for general use in connection with the use of CVODE 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 (Newton) iterations (nni), the number of nonlinear convergence failures (ncnf), and the number of local error test failures (netf). These optional outputs are described in Section 10.1.7, except for nje = iopt[DENSE_NJE], which is described in

Section 10.2.1.3.

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 Section 10.1.2 for a detailed description 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 `N_VIth` 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 Section 10.2.1.2 for a detailed description of the dense `Jac` function.

The output generated by `cvdx.c` 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 =  7.158016e-01    9.185045e-06    2.841893e-01
At t = 4.0000e+02      y =  4.505207e-01    3.222813e-06    5.494761e-01
At t = 4.0000e+03      y =  1.832269e-01    8.943736e-07    8.167722e-01
At t = 4.0000e+04      y =  3.899012e-02    1.622106e-07    9.610097e-01
At t = 4.0000e+05      y =  4.940021e-03    1.985699e-08    9.950600e-01
At t = 4.0000e+06      y =  5.169551e-04    2.068877e-09    9.994830e-01
At t = 4.0000e+07      y =  5.204938e-05    2.082082e-10    9.999480e-01
At t = 4.0000e+08      y =  5.226977e-06    2.090802e-11    9.999948e-01
At t = 4.0000e+09      y =  5.152617e-07    2.061048e-12    9.999995e-01
At t = 4.0000e+10      y =  4.721229e-08    1.888492e-13    1.000000e+00

Final Statistics..

nst = 513    nfe  = 693    nsetups = 100    nje = 11
nni = 690    ncfn = 0      netf = 15
```

4 Band Solver Sample Program

This section includes an example that illustrates the use of CVODE band linear solver CVBAND. The source code is followed by a textual explanation and program output.

The `cvbx.c` program given below solves the semi-discretized form of the 2-D advection-diffusion equation

$$\partial u/\partial t = \partial^2 u/\partial x^2 + .5\partial u/\partial x + \partial^2 u/\partial y^2 \tag{6}$$

on a rectangle, with zero Dirichlet boundary conditions. The PDE is discretized spatially with standard central finite differences on a (MX+2) $\times$ (MY+2) mesh, giving an ODE system of size MX*MY. The discrete value u_{ij} approximates u at $x = i\Delta x$, $y = j\Delta y$. The ODEs are

$$\frac{du_{ij}}{dt} = f_{ij} = \frac{u_{i-1,j} - 2u_{ij} + u_{i+1,j}}{(\Delta x)^2} + .5\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{u_{i,j-1} - 2u_{ij} + u_{i,j+1}}{(\Delta y)^2}, \tag{7}$$

where $1 \leq i \leq$MX and $1 \leq j \leq$MY. The boundary conditions are imposed by taking $u_{ij} = 0$ above if $i = 0$ or MX+1, or if $j = 0$ or MY+1. If we set $v_{(j-1)+(i-1)*MY} = u_{ij}$, then the system Jacobian $J = \partial f/\partial v$ is a band matrix with upper and lower bandwidths both equal to MY. In the example, we take `MX` = 10 and `MY` = 5.

```
/************************************************************************
 *                                                                      *
 * File: cvbx.c                                                         *
 * Programmers: Scott D. Cohen and Alan C. Hindmarsh @ LLNL             *
 * Version of 1 September 1994                                          *
 *----------------------------------------------------------------------*
 * Example problem.                                                     *
 * The following is a simple example problem with a banded Jacobian,    *
 * with the program for its solution by CVODE.                          *
 * The problem is the semi-discrete form of the advection-diffusion     *
 * equation in 2-D:                                                     *
 *   du/dt = d^2 u / dx^2 + .5 du/dx + d^2 u / dy^2                     *
 * on the rectangle 0 <= x <= 2, 0 <= y <= 1, and the time              *
 * interval 0 <= t <= 1.  Homogeneous Dirichlet boundary conditions     *
 * are posed, and the initial condition is                              *
 *   u(x,y,t=0) = x(2-x)y(1-y)exp(5xy) .                                *
 * The PDE is discretized on a uniform MX+2 by MY+2 grid with           *
 * central differencing, and with boundary values eliminated,           *
 * leaving an ODE system of size NEQ = MX*MY.                           *
 * This program solves the problem with the BDF method, Newton          *
 * iteration with the CVODE band linear solver, and a user-supplied     *
 * Jacobian routine.                                                    *
 * It uses scalar relative and absolute tolerances.                     *
 * Output is printed at t = .1, .2, ..., 1.                             *
 * Run statistics (optional outputs) are printed at the end.            *
 ************************************************************************/

#include <stdio.h>
#include <math.h>

/* CVODE header files with a description of contents used in cvbx.c */

#include "llnltyps.h" /* definitions of real, integer, FALSE             */
#include "cvode.h"    /* prototypes for CVodeMalloc, CVode, and CVodeFree, */
                      /* constants OPT_SIZE, BDF, NEWTON, SS, SUCCESS,     */
                      /* NST, NFE, NSETUPS, NNI, NCFN, NETF                */
```

```
#include "cvband.h"   /* prototype for CVBand, constant BAND_NJE          */
#include "vector.h"   /* definitions of type N_Vector and macro N_VDATA,   */
                      /* prototypes for N_VNew, N_VFree, N_VMaxNorm        */
#include "band.h"     /* definitions of type BandMat, macros               */


/* Problem Constants */

#define XMAX  2.0          /* domain boundaries */
#define YMAX  1.0
#define MX    10           /* mesh dimensions   */
#define MY     5
#define NEQ   MX*MY        /* number of equations       */
#define ATOL  1.e-5        /* scalar absolute tolerance */
#define T0    0.0          /* initial time              */
#define T1    0.1          /* first output time         */
#define DTOUT 0.1          /* output time increment     */
#define NOUT  10           /* number of output times    */

/* User-defined vector access macro IJth */

/* IJth is defined in order to isolate the translation from the
   mathematical 2-dimensional structure of the dependent variable vector
   to the underlying 1-dimensional storage.
   IJth(vdata,i,j) references the element in the vdata array for
   u at mesh point (i,j), where 1 <= i <= MX, 1 <= j <= MY.
   The vdata array is obtained via the macro call vdata = N_VDATA(v),
   where v is an N_Vector.
   The variables are ordered by the y index j, then by the x index i. */

#define IJth(vdata,i,j) (vdata[(j-1) + (i-1)*MY])


/* Type : UserData
   contains grid constants */

typedef struct {
  real dx, dy, hdcoef, hacoef, vdcoef;
} *UserData;


/* Private Helper Functions */

static void SetIC(N_Vector u, UserData data);

static void PrintFinalStats(long int iopt[]);

/* Functions Called by the CVODE Solver */

static void f(integer N, real t, N_Vector u, N_Vector udot, void *f_data);

static void Jac(integer N, integer mu, integer ml, BandMat J, RhsFn f,
 void *f_data, real t, N_Vector u, N_Vector fu, N_Vector ewt,
 real h, real uround, void *jac_data, long int *nfePtr,
 N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);


/***************************** Main Program ******************************/

main()
{
  real ropt[OPT_SIZE], dx, dy, reltol, abstol, t, tout, umax;
```

```
long int iopt[OPT_SIZE];
N_Vector u;
UserData data;
void *cvode_mem;
int iout, flag;

u = N_VNew(NEQ, NULL);        /* Allocate u vector */

reltol = 0.0;                 /* Set the tolerances */
abstol = ATOL;

data = (UserData) malloc(sizeof *data);  /* Allocate data memory */
dx = data->dx = XMAX/(MX+1);     /* Set grid coefficients in data */
dy = data->dy = YMAX/(MY+1);
data->hdcoef = 1.0/(dx*dx);
data->hacoef = 0.5/(2.0*dx);
data->vdcoef = 1.0/(dy*dy);

SetIC(u, data);               /* Initialize u vector */

/* 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 cvode_mem.  */

cvode_mem = CVodeMalloc(NEQ, f, T0, u, BDF, NEWTON, SS, &reltol, &abstol,
                        data, NULL, FALSE, iopt, ropt, NULL);
if (cvode_mem == NULL) { printf("CVodeMalloc failed.\n"); return(1); }

/* Call CVBand to specify the CVODE band linear solver with the
   user-supplied Jacobian routine Jac, bandwidths ml = mu = MY,
   and the pointer to the user-defined block data. */

CVBand(cvode_mem, MY, MY, Jac, data);

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

printf(" \n2-D advection-diffusion equation, mesh dimensions =%3d %3d\n\n",
       MX,MY);
umax = N_VMaxNorm(u);
printf("At t = %4.2f    max.norm(u) =%14.6e \n", T0,umax);
for (iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
  flag = CVode(cvode_mem, tout, u, &t, NORMAL);
  umax = N_VMaxNorm(u);
  printf("At t = %4.2f    max.norm(u) =%14.6e   nst =%4d \n",
          t,umax,iopt[NST]);
  if (flag != SUCCESS) { printf("CVode failed, flag=%d.\n", flag); break; }
}

N_VFree(u);                   /* Free the u vector */
CVodeFree(cvode_mem);         /* Free the CVODE problem memory */
```

```
  PrintFinalStats(iopt);       /* Print some final statistics   */
  return(0);
}


/************************ Private Helper Functions **********************/

/* Set initial conditions in u vector */

static void SetIC(N_Vector u, UserData data)
{
  int i, j;
  real x, y, dx, dy;
  real *udata;

  /* Extract needed constants from data */

  dx = data->dx;
  dy = data->dy;

  /* Set pointer to data array in vector u. */

  udata = N_VDATA(u);

  /* Load initial profile into u vector */

  for (j=1; j <= MY; j++) {
    y = j*dy;
    for (i=1; i <= MX; i++) {
      x = i*dx;
      IJth(udata,i,j) = x*(XMAX - x)*y*(YMAX - y)*exp(5.0*x*y);
    }
  }
}


/* 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[BAND_NJE]);
  printf("nni = %-6ld ncfn = %-6ld netf = %ld\n \n",
 iopt[NNI], iopt[NCFN], iopt[NETF]);
}


/***************** Functions Called by the CVODE Solver *****************/

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

static void f(integer N, real t, N_Vector u, N_Vector udot, void *f_data)
{
  real uij, udn, uup, ult, urt, hordc, horac, verdc, hdiff, hadv, vdiff;
  real *udata, *dudata;
  int i, j;
  UserData data;

  udata = N_VDATA(u);
  dudata = N_VDATA(udot);
```

```
  /* Extract needed constants from data */

  data = (UserData) f_data;
  hordc = data->hdcoef;
  horac = data->hacoef;
  verdc = data->vdcoef;

  /* Loop over all grid points. */

  for (j=1; j <= MY; j++) {

    for (i=1; i <= MX; i++) {

      /* Extract u at x_i, y_j and four neighboring points */

      uij = IJth(udata, i, j);
      udn = (j == 1)  ? 0.0 : IJth(udata, i, j-1);
      uup = (j == MY) ? 0.0 : IJth(udata, i, j+1);
      ult = (i == 1)  ? 0.0 : IJth(udata, i-1, j);
      urt = (i == MX) ? 0.0 : IJth(udata, i+1, j);

      /* Set diffusion and advection terms and load into udot */

      hdiff = hordc*(ult - 2.0*uij + urt);
      hadv = horac*(urt - ult);
      vdiff = verdc*(uup - 2.0*uij + udn);
      IJth(dudata, i, j) = hdiff + hadv + vdiff;
    }
  }
}


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

static void Jac(integer N, integer mu, integer ml, BandMat J, RhsFn f,
 void *f_data, real t, N_Vector u, N_Vector fu, N_Vector ewt,
 real h, real uround, void *jac_data, long int *nfePtr,
 N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
  integer i, j, k;
  real *kthCol, hordc, horac, verdc;
  UserData data;

  /*
    The components of f = udot that depend on u(i,j) are
    f(i,j), f(i-1,j), f(i+1,j), f(i,j-1), f(i,j+1), with
      df(i,j)/du(i,j) = -2 (1/dx^2 + 1/dy^2)
      df(i-1,j)/du(i,j) = 1/dx^2 + .25/dx  (if i > 1)
      df(i+1,j)/du(i,j) = 1/dx^2 - .25/dx  (if i < MX)
      df(i,j-1)/du(i,j) = 1/dy^2           (if j > 1)
      df(i,j+1)/du(i,j) = 1/dy^2           (if j < MY)
  */

  data = (UserData) jac_data;
  hordc = data->hdcoef;
  horac = data->hacoef;
  verdc = data->vdcoef;

  for (j=1; j <= MY; j++) {
    for (i=1; i <= MX; i++) {
      k = j-1 + (i-1)*MY;
      kthCol = BAND_COL(J,k);
```

```
        /* set the kth column of J */

        BAND_COL_ELEM(kthCol,k,k) = -2.0*(verdc+hordc);
        if (i != 1)  BAND_COL_ELEM(kthCol,k-MY,k) = hordc + horac;
        if (i != MX) BAND_COL_ELEM(kthCol,k+MY,k) = hordc - horac;
        if (j != 1)  BAND_COL_ELEM(kthCol,k-1,k)  = verdc;
        if (j != MY) BAND_COL_ELEM(kthCol,k+1,k)  = verdc;
      }
    }
  }
```

The cvbx.c program includes the files cvband.h and band.h in order to use the CVBAND linear solver. The cvband.h file contains the prototype for the CVBand routine, as well as the index constant BAND_NJE used in accessing optional output specific to CVBAND. See Section 10.2.2. The band.h file contains the definition for band matrix type BandMat and the BAND_COL and BAND_COL_ELEM macros for accessing matrix elements. See Section 10.3.3. We have explicitly included band.h, but this is not necessary because it is included by cvband.h. The file vector.h is included for the definition of the type N_Vector and the accessor macro N_VDATA. This macro is fully documented in Section 10.3.1.2.

The include lines at the top of the file are followed by definitions of problem constants which include the x and y mesh dimensions, MX and MY, the number of equations NEQ, the scalar absolute tolerance ATOL, the initial time T0, and the initial output time T1.

Spatial discretization of the partial differential equation naturally produces an ODE system in which equations are numbered by mesh coordinates (i, j). The user-defined macro IJth isolates the translation for the mathematical two-dimensional index to the one-dimensional N_Vector index and allows the user to write clean, readable code to access components of the dependent variable. The N_VDATA macro returns the component array for a given N_Vector, and this array is passed to IJth in order to do the actual N_Vector access.

The type UserData is a pointer to a structure containing problem data that the user needs in his/her f and Jac functions. This structure is allocated and initialized at the beginning of main. The pointer to this structure, called data, is passed to both CVodeMalloc and CVBand. Passing data to CVodeMalloc for the formal parameter f_data means that data will be passed to the f function each time f is called. Passing data to CVBand for the formal parameter jac_data means that data will also be passed to the Jac function each time Jac is called. The use of the data pointer eliminates the need for global program data.

The four functions other than main in the cvbx.c file are SetIC, PrintFinalStats, f, and Jac. The first two functions are called only from within the cvbx.c file. The SetIC function sets the initial dependent variable vector, and PrintFinalStats prints out statistics at the end of the run. The latter include various counters, such as the total number of steps (nst), f evaluations (nfe), LU decompositions (nsetups), Jacobian evaluations (nje), and nonlinear iterations (nni). These optional outputs are described in Sections 10.1.7 and 10.2.2.3.

The main function is straightforward. The CVodeMalloc call specifies the BDF method with a NEWTON iteration. The SS actual parameter indicates scalar relative and absolute tolerances, and pointers &reltol and &abstol to these values are passed. The call to CVBand with a non-NULL Jacobian function Jac specifies the CVBAND linear solver with a user-supplied Jacobian. The constant MY is passed for both the upper and lower bandwidths of the Jacobian. See Section 10.2.2.1 for full details on CVBand. The actual solution of the problem is performed by the call to CVode within the loop over the output times tout. The max-norm of the solution vector and the number of time steps is printed at each output time. Finally, the calls to N_VFree, CVodeFree, and PrintFinalStats free problem memory and print statistics.

The user's f function implements the central difference approximation (7) with u identically zero on the boundary.

The user's Jac function is an expression of the derivatives

$$\begin{aligned}
\partial f_{ij}/\partial u_{ij} &= -2[(\Delta x)^{-2} + (\Delta y)^{-2}] \\
\partial f_{ij}/\partial u_{i+1,j} &= (\Delta x)^{-2} + .5(2\Delta x)^{-1}, \quad \partial f_{ij}/\partial u_{i-1,j} = (\Delta x)^{-2} - .5(2\Delta x)^{-1} \\
\partial f_{ij}/\partial u_{i,j+1} &= (\Delta y)^{-2}, \quad \partial f_{ij}/\partial u_{i,j-1} = (\Delta y)^{-2} \; .
\end{aligned}$$

The values $(\Delta x)^{-2}$, $.5(2\Delta x)^{-1}$, and $(\Delta y)^{-2}$ are computed only once at the beginning of main, and stored in the locations data->hdcoef, data->hacoef, and data->vdcoef, respectively. When Jac receives the data pointer, it pulls out these values from storage in the local variables hordc, horac, and verdc. The Jac function loads the Jacobian by columns. It loops over the mesh points (i,j). For each such mesh point, the one-dimensional index k = j-1 + (i-1)*MY is computed and the kth column of the Jacobian matrix J is set. The row index k' of each component $f_{i',j'}$ that depends on $u_{i,j}$ must be identified in order to load the corresponding element. The elements are loaded with the BAND_COL_ELEM macro. Note that the formula for the global index k implies that decreasing (increasing) i by 1 corresponds to decreasing (increasing) k by MY, while decreasing (increasing) j by 1 corresponds of decreasing (increasing) k by 1. These statements are reflected in the calls to BAND_COL_ELEM. The first argument passed to the BAND_COL_ELEM macro is a pointer to the diagonal element in the column to be accessed. This pointer is obtained via a call to the BAND_COL macro and is stored in kthCol in the Jac function. When setting the components of J we must be careful not to index out of bounds. The guards (i != 1), (i != MX), (j != 1), and (j != MY) in front of the calls to BAND_COL_ELEM prevent illegal indexing.

The output generated by cvbx.c is shown below.

```
2-D advection-diffusion equation, mesh dimensions = 10   5

At t = 0.00    max.norm(u) =  8.954716e+01
At t = 0.10    max.norm(u) =  4.132889e+00   nst =  85
At t = 0.20    max.norm(u) =  1.039294e+00   nst = 103
At t = 0.30    max.norm(u) =  2.979829e-01   nst = 113
At t = 0.40    max.norm(u) =  8.765774e-02   nst = 120
At t = 0.50    max.norm(u) =  2.625637e-02   nst = 126
At t = 0.60    max.norm(u) =  7.830425e-03   nst = 130
At t = 0.70    max.norm(u) =  2.329387e-03   nst = 134
At t = 0.80    max.norm(u) =  6.953434e-04   nst = 137
At t = 0.90    max.norm(u) =  2.115983e-04   nst = 140
At t = 1.00    max.norm(u) =  6.556853e-05   nst = 142

Final Statistics..

nst = 142    nfe  = 173    nsetups = 23     nje = 3
nni = 170    ncfn = 0      netf = 3
```

5 Krylov Solver Sample Program

We give here an example that illustrates the use of CVODE with the Krylov method SPGMR, in the CVSPGMR module, as the linear system solver. Following the source are explanatory comments on the program.

The following program solves the semi-discretized form of a pair of kinetics-advection-diffusion partial differential equations. The problem is defined on a rectangle, and discretized spatially with standard central finite differences on a 10×10 mesh, giving an ODE system of size 200.

```
/************************************************************************
 *                                                                      *
 * File: cvkx.c                                                         *
 * Programmers: Scott D. Cohen and Alan C. Hindmarsh @ LLNL             *
 * Version of 1 September 1994                                          *
 *----------------------------------------------------------------------*
 * 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/dz)(Kv(z)*dc(i)/dz)    *
 *                 + 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(z) = Kv0*exp(z/5) ,                                             *
 * Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)        *
 * vary diurnally.   The problem is posed on the square                 *
 *   0 <= x <= 20,    30 <= z <= 50   (all in km),                      *
 * with homogeneous Neumann boundary conditions, and for time t in      *
 *   0 <= t <= 86400 sec (1 day).                                       *
 * The PDE system is treated by central differences on a uniform        *
 * 10 x 10 mesh, with simple polynomial initial profiles.               *
 * The problem is solved with CVODE, 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.                                                     *
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "llnltyps.h"  /* definitions of real, integer, bool, TRUE, FALSE */
#include "cvode.h"     /* main CVODE header file                          */
#include "iterativ.h"  /* contains the enum for types of preconditioning  */
#include "cvspgmr.h"   /* use CVSPGMR linear solver each internal step    */
#include "dense.h"     /* use generic DENSE solver for preconditioning    */
#include "vector.h"    /* definitions of type N_Vector, macro N_VDATA     */
#include "llnlmath.h"  /* contains SQR macro                              */


/* Problem Constants */

#define NUM_SPECIES  2            /* number of species         */
#define KH           4.0e-6       /* horizontal diffusivity Kh */
#define VEL          0.001        /* advection velocity V      */
#define KV0          1.0e-8       /* coefficient in Kv(z)      */
#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 ZMIN        30.0           /* grid boundaries in z  */
#define ZMAX        50.0
#define XMID        10.0           /* grid midpoints in x,z */
#define ZMID        40.0

#define MX          10              /* MX = number of x mesh points */
#define MZ          10              /* MZ = number of z mesh points */
#define NSMX        20              /* NSMX = NUM_SPECIES*MX */
#define MM          (MX*MZ)         /* MM = MX*MZ */

/* CVodeMalloc 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 */
#define NEQ     (NUM_SPECIES*MM)  /* NEQ = number of equations */


/* User-defined vector and matrix accessor macros: IJKth, IJth */

/* IJKth is defined in order to isolate the translation from the
   mathematical 3-dimensional structure of the dependent variable vector
   to the underlying 1-dimensional storage. IJth is defined in order to
   write code which indexes into small dense matrices with a (row,column)
   pair, where 1 <= row, column <= NUM_SPECIES.

   IJKth(vdata,i,j,k) references the element in the vdata array for
   species i at mesh point (j,k), where 1 <= i <= NUM_SPECIES,
   0 <= j <= MX-1, 0 <= k <= MZ-1. The vdata array is obtained via
   the macro call vdata = N_VDATA(v), where v is an N_Vector.
   For each mesh point (j,k), the elements for species i and i+1 are
   contiguous within vdata.

   IJth(a,i,j) references the (i,j)th entry of the small matrix real **a,
   where 1 <= i,j <= NUM_SPECIES. 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 IJKth(vdata,i,j,k) (vdata[i-1 + (j)*NUM_SPECIES + (k)*NSMX])
#define IJth(a,i,j)        (a[j-1][i-1])


/* Type : UserData
   contains preconditioner blocks, pivot arrays, and problem constants */

typedef struct {
```

```
  real **P[MX][MZ], **Jbd[MX][MZ];
  integer *pivot[MX][MZ];
  real q4, om, dx, dz, hdco, haco, vdco;
} *UserData;


/* Private Helper Functions */

static UserData AllocUserData(void);
static void InitUserData(UserData data);
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector y, real dx, real dz);
static void PrintOutput(long int iopt[], real ropt[], N_Vector y, real t);
static void PrintFinalStats(long int iopt[]);

/* Functions Called by the CVODE Solver */

static void f(integer N, real t, N_Vector y, N_Vector ydot, void *f_data);

static int Precond(integer N, real tn, N_Vector y, N_Vector fy, bool jok,
                   bool *jcurPtr, real gamma, N_Vector ewt, real h,
                   real uround, long int *nfePtr, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);

static int PSolve(integer N, real tn, N_Vector y, N_Vector fy, N_Vector vtemp,
                  real gamma, N_Vector ewt, real delta, long int *nfePtr,
                  N_Vector r, int lr, void *P_data, N_Vector z);


/***************************** Main Program *****************************/

main()
{
  real abstol, reltol, t, tout, ropt[OPT_SIZE];
  long int iopt[OPT_SIZE];
  N_Vector y;
  UserData data;
  void *cvode_mem;
  int iout, flag;

  /* Allocate memory, and set problem data, initial values, tolerances */

  y = N_VNew(NEQ, NULL);
  data = AllocUserData();
  InitUserData(data);
  SetInitialProfiles(y, data->dx, data->dz);
  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 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
     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 cvode_mem.  */

  cvode_mem = CVodeMalloc(NEQ, f, T0, y, BDF, NEWTON, SS, &reltol,
                          &abstol, data, NULL, FALSE, iopt, ropt, NULL);
  if (cvode_mem == NULL) { printf("CVodeMalloc failed."); return(1); }

  /* Call CVSpgmr 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, and the pointer to the user-defined block data.              */

  CVSpgmr(cvode_mem, LEFT, MODIFIED_GS, 0, 0.0, Precond, PSolve, data);

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

  printf(" \n2-species diurnal advection-diffusion problem\n\n");
  for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CVode(cvode_mem, tout, y, &t, NORMAL);
    PrintOutput(iopt, ropt, y, t);
    if (flag != SUCCESS) { printf("CVode failed, flag=%d.\n", flag); break; }
  }

  /* Free memory and print final statistics */

  N_VFree(y);
  FreeUserData(data);
  CVodeFree(cvode_mem);
  PrintFinalStats(iopt);
  return(0);
}


/*********************** Private Helper Functions ************************/

/* Allocate memory for data structure of type UserData */

static UserData AllocUserData(void)
{
  int jx, jz;
  UserData data;

  data = (UserData) malloc(sizeof *data);

  for (jx=0; jx < MX; jx++) {
    for (jz=0; jz < MZ; jz++) {
      (data->P)[jx][jz] = denalloc(NUM_SPECIES);
      (data->Jbd)[jx][jz] = denalloc(NUM_SPECIES);
      (data->pivot)[jx][jz] = denallocpiv(NUM_SPECIES);
    }
  }

  return(data);
}

/* Load problem constants in data */

static void InitUserData(UserData data)
{
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/(MX-1);
  data->dz = (ZMAX-ZMIN)/(MZ-1);
```

```
  data->hdco = KH/SQR(data->dx);
  data->haco = VEL/(2.0*data->dx);
  data->vdco = (1.0/SQR(data->dz))*KV0;
}

/* Free data memory */

static void FreeUserData(UserData data)
{
  int jx, jz;

  for (jx=0; jx < MX; jx++) {
    for (jz=0; jz < MZ; jz++) {
      denfree((data->P)[jx][jz]);
      denfree((data->Jbd)[jx][jz]);
      denfreepiv((data->pivot)[jx][jz]);
    }
  }

  free(data);
}

/* Set initial conditions in y */

static void SetInitialProfiles(N_Vector y, real dx, real dz)
{
  int jx, jz;
  real x, z, cx, cz;
  real *ydata;

  /* Set pointer to data array in vector y. */

  ydata = N_VDATA(y);

  /* Load initial profiles of c1 and c2 into y vector */

  for (jz=0; jz < MZ; jz++) {
    z = ZMIN + jz*dz;
    cz = SQR(0.1*(z - ZMID));
    cz = 1.0 - cz + 0.5*SQR(cz);
    for (jx=0; jx < MX; jx++) {
      x = XMIN + jx*dx;
      cx = SQR(0.1*(x - XMID));
      cx = 1.0 - cx + 0.5*SQR(cx);
      IJKth(ydata,1,jx,jz) = C1_SCALE*cx*cz;
      IJKth(ydata,2,jx,jz) = C2_SCALE*cx*cz;
    }
  }
}

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

static void PrintOutput(long int iopt[], real ropt[], N_Vector y, real t)
{
  real *ydata;

  ydata = N_VDATA(y);

  printf("t = %.2e   no. steps = %d   order = %d   stepsize = %.2e\n",
         t, iopt[NST], iopt[QU], ropt[HU]);
  printf("c1 (bot.left/middle/top rt.) = %12.3e  %12.3e  %12.3e\n",
         IJKth(ydata,1,0,0), IJKth(ydata,1,4,4), IJKth(ydata,1,9,9));
```

```
  printf("c2 (bot.left/middle/top rt.) = %12.3e  %12.3e  %12.3e\n\n",
         IJKth(ydata,2,0,0), IJKth(ydata,2,4,4), IJKth(ydata,2,9,9));
}

/* 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]);
}


/***************** Functions Called by the CVODE Solver ******************/

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

static void f(integer N, real t, N_Vector y, N_Vector ydot, void *f_data)
{
  real q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  real c1rt, c2rt, czdn, czup, hord1, hord2, horad1, horad2;
  real qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, zdn, zup;
  real q4coef, delz, verdco, hordco, horaco;
  real *ydata, *dydata;
  int jx, jz, idn, iup, ileft, iright;
  UserData data;

  data = (UserData) f_data;
  ydata = N_VDATA(y);
  dydata = N_VDATA(ydot);

  /* Set diurnal rate coefficients. */

  s = sin(data->om*t);
  if (s > 0.0) {
    q3 = exp(-A3/s);
    data->q4 = exp(-A4/s);
  } else {
    q3 = 0.0;
    data->q4 = 0.0;
  }

  /* Make local copies of problem variables, for efficiency. */

  q4coef = data->q4;
  delz = data->dz;
  verdco = data->vdco;
  hordco  = data->hdco;
  horaco  = data->haco;

  /* Loop over all grid points. */

  for (jz=0; jz < MZ; jz++) {

    /* Set vertical diffusion coefficients at jz +- 1/2 */
```

```
    zdn = ZMIN + (jz - .5)*delz;
    zup = zdn + delz;
    czdn = verdco*exp(0.2*zdn);
    czup = verdco*exp(0.2*zup);
    idn = (jz == 0) ? 1 : -1;
    iup = (jz == MZ-1) ? -1 : 1;
    for (jx=0; jx < MX; jx++) {

      /* Extract c1 and c2, and set kinetic rate terms. */

      c1 = IJKth(ydata,1,jx,jz);
      c2 = IJKth(ydata,2,jx,jz);
      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 = IJKth(ydata,1,jx,jz+idn);
      c2dn = IJKth(ydata,2,jx,jz+idn);
      c1up = IJKth(ydata,1,jx,jz+iup);
      c2up = IJKth(ydata,2,jx,jz+iup);
      vertd1 = czup*(c1up - c1) - czdn*(c1 - c1dn);
      vertd2 = czup*(c2up - c2) - czdn*(c2 - c2dn);

      /* Set horizontal diffusion and advection terms. */

      ileft = (jx == 0) ? 1 : -1;
      iright =(jx == MX-1) ? -1 : 1;
      c1lt = IJKth(ydata,1,jx+ileft,jz);
      c2lt = IJKth(ydata,2,jx+ileft,jz);
      c1rt = IJKth(ydata,1,jx+iright,jz);
      c2rt = IJKth(ydata,2,jx+iright,jz);
      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 ydot. */

      IJKth(dydata, 1, jx, jz) = vertd1 + hord1 + horad1 + rkin1;
      IJKth(dydata, 2, jx, jz) = vertd2 + hord2 + horad2 + rkin2;
    }
  }
}

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

static int Precond(integer N, real tn, N_Vector y, N_Vector fy, bool jok,
                   bool *jcurPtr, real gamma, N_Vector ewt, real h,
                   real uround, long int *nfePtr, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
  real c1, c2, czdn, czup, diag, zdn, zup, q4coef, delz, verdco, hordco;
  real **(*P)[MZ], **(*Jbd)[MZ];
  integer *(*pivot)[MZ], ier;
  int jx, jz;
  real *ydata, **a, **j;
  UserData data;
```

```
/* Make local copies of pointers in P_data, and of pointer to y's data */

data = (UserData) P_data;
P = data->P;
Jbd = data->Jbd;
pivot = data->pivot;
ydata = N_VDATA(y);

if (jok) {

/* jok = TRUE: Copy Jbd to P */

  for (jz=0; jz < MZ; jz++)
    for (jx=0; jx < MX; jx++)
      dencopy(Jbd[jx][jz], P[jx][jz], NUM_SPECIES);

*jcurPtr = FALSE;

}

else {
/* jok = FALSE: Generate Jbd from scratch and copy to P */

/* Make local copies of problem variables, for efficiency. */

q4coef = data->q4;
delz = data->dz;
verdco = data->vdco;
hordco  = data->hdco;

/* Compute 2x2 diagonal Jacobian blocks,
   (using q4 values computed on the last f call).  Load into P. */

  for (jz=0; jz < MZ; jz++) {
    zdn = ZMIN + (jz - .5)*delz;
    zup = zdn + delz;
    czdn = verdco*exp(0.2*zdn);
    czup = verdco*exp(0.2*zup);
    diag = -(czdn + czup + 2.0*hordco);
    for (jx=0; jx < MX; jx++) {
      c1 = IJKth(ydata,1,jx,jz);
      c2 = IJKth(ydata,2,jx,jz);
      j = Jbd[jx][jz];
      a = P[jx][jz];
      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, NUM_SPECIES);
    }
  }

*jcurPtr = TRUE;

}

/* Scale by -gamma */

  for (jz=0; jz < MZ; jz++)
    for (jx=0; jx < MX; jx++)
      denscale(-gamma, P[jx][jz], NUM_SPECIES);
```

```
  /* Add identity matrix and do LU decompositions on blocks in place. */

  for (jx=0; jx < MX; jx++) {
    for (jz=0; jz < MZ; jz++) {
      denaddI(P[jx][jz], NUM_SPECIES);
      ier = gefa(P[jx][jz], NUM_SPECIES, pivot[jx][jz]);
      if (ier != 0) return(1);
    }
  }

  return(0);
}


/* Preconditioner solve routine */

static int PSolve(integer N, real tn, N_Vector y, N_Vector fy, N_Vector vtemp,
                  real gamma, N_Vector ewt, real delta, long int *nfePtr,
                  N_Vector r, int lr, void *P_data, N_Vector z)
{
  real **(*P)[MZ];
  integer *(*pivot)[MZ];
  int jx, jz;
  real *zdata, *v;
  UserData data;

  /* Extract the P and pivot arrays from P_data. */

  data = (UserData) P_data;
  P = data->P;
  pivot = data->pivot;
  zdata = N_VDATA(z);

  N_VScale(1.0, r, z);

  /* Solve the block-diagonal system Px = r using LU factors stored
     in P and pivot data in pivot, and return the solution in z. */

  for (jx=0; jx < MX; jx++) {
    for (jz=0; jz < MZ; jz++) {
      v = &(IJKth(zdata, 1, jx, jz));
      gesl(P[jx][jz], NUM_SPECIES, pivot[jx][jz], v);
    }
  }

  return(0);
}
```

Among the initial `#include` lines in this case are lines to include `iterativ.h`, `cvspgmr.h`, and `llnlmath.h`. The first two contain constants and function prototypes associated with preconditioned Krylov methods, including the values of the `pretype` argument to `CVSpgmr`. The inclusion of `llnlmath.h` is done to access the `SQR` macro for the square of a `real` number.

The `main` function calls `CVodeMalloc` specifying `BDF` and `NEWTON`, with scalar tolerances. It calls `CVSpgmr` specifying the CVSPGMR linear solver with left preconditioning, modified Gram-Schmidt orthogonalization, default values (indicated by zero arguments) for the optional inputs `maxl` and `delt`, and non-`NULL` preconditioner setup and solve functions, `Precond` and `PSolve`. The `data` pointer passed to CVSpgmr is passed to Precond and PSolve whenever these are called. See Section 10.2.4.1 for details on the CVSpgmr function. The preconditioner is the block-diagonal part of the true Newton matrix. Then for a sequence of tout values, CVode is called in the NORMAL mode, sampled output is printed, and the return value is tested for error conditions. Finally, CVodeFree is called and final statistics are printed. The latter include various counters, such as the total number of steps (nst), f evaluations (nfe), nonlinear iterations (nni), and linear iterations (nli), among others. Also printed are the lengths of the problem-dependent real and integer workspaces used by the main integrator CVode, denoted lenrw and leniw, and those used by CVSPGMR, denoted llrw and lliw. These optional outputs are described in Sections 10.1.7 and 10.2.4.4.

Mathematically, the dependent variable has three dimensions: species number, x mesh point, and y mesh point. But in CVODE, a vector of type N_Vector works with a one-dimensional contiguous array of data components. The macro IJKth isolates the translation from three dimensions to one. Its use results in clearer code and makes it easy to change the underlying layout of the three-dimensional data. Here the problem size is 200, so we use the N_VDATA macro for efficient N_Vector access. The N_VDATA macro gives a pointer to the first component of an N_Vector which we pass to the IJKth macro to do an N_Vector access. The N_VDATA macro is documented in Section 10.3.1.2.

The preconditioner is generated and factored in the Precond routine and backsolved in the PSolve routine. It is a block-diagonal matrix with 2×2 blocks that include the interaction Jacobian elements and the diagonal contribution of the diffusion Jacobian elements. The block-diagonal part of the Jacobian itself, J_{bd}, is saved in separate storage each time it is generated, on calls to Precond with jok == FALSE. On calls with jok == TRUE, signifying that saved Jacobian data can be reused, the preconditioner $P = I - \gamma J_{bd}$ is formed from the saved matrix J_{bd} and factored. (A call to Precond with jok == TRUE can only occur after a call with jok == FALSE.) In each case, we set the value of jcur, i.e. *jcurPtr, to TRUE when J_{bd} is re-evaluated, and FALSE otherwise, to inform CVSPGMR of the status of Jacobian data.

We need to take a brief detour to explain one last important point about the cvkx.c user program. As documented in dense.h and band.h, the generic DENSE and BAND solvers each contain two sets of routines: one for large, potentially distributed matrices, and one for small matrices which will always reside on a single processor. The "large" dense and band routines work with the types DenseMat and BandMat, respectively. The "small"dense and band routines work with real ** as the underlying dense and band matrix types. The types DenseMat and BandMat, as well as their supporting functions, may be re-implemented in order to use this generic DENSE solver in a distributed-memory environment. The CVDENSE and CVBAND linear solvers use the types DenseMat and BandMat for the $N \times N$ dense and band Jacobian and Newton matrices, and call the large matrix routines. The small dense and band routines are available for CVODE user programs, and are documented in Sections 10.3.2.4 and 10.3.3.4, respectively. In cvkx.c, each of the blocks of the preconditioner is a 2×2 dense matrix. Thus the small dense matrix routines denalloc, dencopy, denscale, gefa, gesl, denaddI, denfree, and denfreepiv are used. The macro IJth defined near the top of the file is used to access individual elements in each preconditioner block, numbered from 1.

The output generated by cvkx.c is shown below. Note that the number of preconditioner evaluations, npe, is much smaller than the number of preconditioner setups, nsetups, as a result of the Jacobian re-use scheme.

```
2-species diurnal advection-diffusion problem

t = 7.20e+03   no. steps = 219   order = 5   stepsize = 1.59e+02
c1 (bot.left/middle/top rt.) =    1.047e+04     2.964e+04     1.119e+04
c2 (bot.left/middle/top rt.) =    2.527e+11     7.154e+11     2.700e+11

t = 1.44e+04   no. steps = 251   order = 5   stepsize = 3.77e+02
c1 (bot.left/middle/top rt.) =    6.659e+06     5.316e+06     7.301e+06
c2 (bot.left/middle/top rt.) =    2.582e+11     2.057e+11     2.833e+11

t = 2.16e+04   no. steps = 277   order = 5   stepsize = 2.75e+02
c1 (bot.left/middle/top rt.) =    2.665e+07     1.036e+07     2.931e+07
c2 (bot.left/middle/top rt.) =    2.993e+11     1.028e+11     3.313e+11

t = 2.88e+04   no. steps = 324   order = 3   stepsize = 1.16e+02
c1 (bot.left/middle/top rt.) =    8.702e+06     1.292e+07     9.650e+06
c2 (bot.left/middle/top rt.) =    3.380e+11     5.029e+11     3.751e+11

t = 3.60e+04   no. steps = 367   order = 4   stepsize = 6.16e+01
c1 (bot.left/middle/top rt.) =    1.404e+04     2.029e+04     1.561e+04
c2 (bot.left/middle/top rt.) =    3.387e+11     4.894e+11     3.765e+11

t = 4.32e+04   no. steps = 422   order = 4   stepsize = 5.42e+02
c1 (bot.left/middle/top rt.) =   -1.487e-06    -1.085e-05    -1.756e-06
c2 (bot.left/middle/top rt.) =    3.382e+11     1.355e+11     3.804e+11

t = 5.04e+04   no. steps = 438   order = 4   stepsize = 2.55e+02
c1 (bot.left/middle/top rt.) =   -4.109e-11    -2.908e-10    -4.857e-11
c2 (bot.left/middle/top rt.) =    3.358e+11     4.930e+11     3.864e+11

t = 5.76e+04   no. steps = 452   order = 5   stepsize = 3.93e+02
c1 (bot.left/middle/top rt.) =    5.077e-11     3.991e-10     5.971e-11
c2 (bot.left/middle/top rt.) =    3.320e+11     9.650e+11     3.909e+11

t = 6.48e+04   no. steps = 463   order = 5   stepsize = 6.62e+02
c1 (bot.left/middle/top rt.) =    3.436e-12     2.760e-11     4.030e-12
c2 (bot.left/middle/top rt.) =    3.313e+11     8.922e+11     3.963e+11

t = 7.20e+04   no. steps = 474   order = 5   stepsize = 6.62e+02
c1 (bot.left/middle/top rt.) =   -1.191e-12    -1.056e-11    -1.367e-12
c2 (bot.left/middle/top rt.) =    3.330e+11     6.186e+11     4.039e+11

t = 7.92e+04   no. steps = 485   order = 5   stepsize = 6.62e+02
c1 (bot.left/middle/top rt.) =   -5.511e-14    -4.677e-13    -6.380e-14
c2 (bot.left/middle/top rt.) =    3.334e+11     6.669e+11     4.120e+11

t = 8.64e+04   no. steps = 496   order = 5   stepsize = 6.62e+02
c1 (bot.left/middle/top rt.) =   -1.789e-15    -1.548e-14    -2.068e-15
c2 (bot.left/middle/top rt.) =    3.352e+11     9.107e+11     4.163e+11


Final Statistics..

lenrw   =  2000    leniw =     0
llrw    =  2046    lliw  =     0
nst     =   496    nfe   =  1269
nni     =   640    nli   =   626
nsetups =    90    netf  =    30
npe     =     9    nps   =  1204
ncfn    =     0    ncfl  =     0
```

6 CVODE Linear Solvers

As previously explained, Newton iteration requires the solution of linear systems of the form (4). There are four CVODE 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 CVODE linear solver, CVSPGMR, is an iterative solver. The SPGMR in the name indicates that it uses a scaled preconditioned GMRES method.

To specify a CVODE 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 CVODE memory pointer returned by `CVodeMalloc`. A call to one of these functions links the main CVODE integrator to a CVODE 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 CVODE 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 Sections 10.3.2-10.3.4.

6.1 Direct Methods

6.1.1 CVDENSE

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

```
#include "cvdense.h"
```

After the call to `CVodeMalloc`, the user must call the following routine to select the CVDENSE solver:

```
void CVDense(void *cvode_mem, CVDenseJacFn djac, void *jac_data);
```

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`. 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 type of the dense Jacobian routine `CVDenseJacFn` is defined by

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

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 the `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) =` $J_{m,n}$. 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] =` $J_{m,n}$. 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 Sections 10.3.2.1 and 10.3.2.2.

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.

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 $2N^2$ real words and N integer words.

See the reference guide Section 10.2.1 for documentation for the function `CVDense` and the type `CVDenseJacFn`, and for a listing of these CVDENSE optional outputs.

6.1.2 CVBAND

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

```
#include "cvband.h"
```

After the call to CVodeMalloc, the user must call the following routine to select the CVBAND solver:

```
void CVBand(void *cvode_mem, integer mupper, integer mlower, CVBandJacFn bjac,
            void *jac_data);
```

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

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. 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 type of the band Jacobian routine CVBandJacFn is defined by

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

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) = $J_{m,n}$. Alternatively, BAND_COL(A,j) returns a pointer to the diagonal element of the jth column of A, and if we assign this address to real *col_j, then the ith element of the jth 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 jth 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 Sections 10.3.3.1 and 10.3.3.2.

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.

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) $N * (2$ mupper $+ 3$ mlower $+ 2)$ real words and N integer words.

See the reference guide Section 10.2.2 for documentation for the function CVBand and the type CVBandJacFn, and for a listing of these CVBAND optional outputs.

6.1.3 CVDIAG

In using the CVDIAG solver with CVODE, 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 following routine to select the CVDIAG solver.

```
void CVDiag(void *cvode_mem);
```

The CVDIAG solver is the simplest of all the current CVODE linear solvers. The CVDiag routine receives only the CVODE memory pointer returned by CVodeMalloc. CVDIAG 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 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 $3N$ real words and no integer words. The number of approximate diagonal Jacobians formed is equal to iopt[NSETUPS].

See the reference guide Section 10.2.3 for documentation of the function CVDiag and the CVDIAG optional outputs.

6.2 Iterative Methods

6.2.1 CVSPGMR

The CVSPGMR solver uses a scaled preconditioned GMRES iterative method to solve the linear system (4). 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 CVODE, 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 following routine to select the CVSPGMR solver.

```
void CVSpgmr(void *cvode_mem, int pretype, int gstype, int maxl, real delt,
             CVSpgmrPrecondFn precond, CVSpgmrPSolveFn psolve, void *P_data);
```

The call to `CVSpgmr` 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 `CVSpgmr` 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.

As in the dense and band cases, `CVSpgmr` 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. This function has type CVSpgmrPSolveFn as defined by

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

If P_1 is the left preconditioner and P_2 is the right preconditioner, then the user's psolve function must solve P_1z = r if lr is LEFT or P_2z = r if lr is RIGHT.

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. This routine must be of type CVSpgmrPrecondFn as defined by

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

This routine is not called in advance of every psolve call, 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 psolve. 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 γ = gamma).

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 ℓ_{max}, the actual sizes of these workspaces are $N*(\ell_{max}+5)+\ell_{max}*(\ell_{max}+4)+1$ real words and no integer words.

See the reference guide Section 10.2.4 for documentation of the function CVSpgmr, the types CVSpgmrPrecondFn and CVSpgmrPSolveFn of the user-supplied preconditioner functions, and the optional outputs from CVSPGMR.

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 CVSpgmr 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 Section 10.3.4.

7 Types real and integer

7.1 Description

The `llnltyps.h` file contains the definitions of the types `real` and `integer`. CVODE uses the type `real` for all floating point data and the type `integer` for all problem size-related data such as the actual problem size, the bandwidths in the CVBAND solver, and the integers stored in the length N pivot arrays in both the CVDENSE and CVBAND solvers. These types make it easy to have CVODE solve problems of virtually any size using single or double precision arithmetic. The type `real` can be `double` or `float` and the type `integer` can be `int` or `long int`. The default settings are `double` and `int`.

7.2 Changing Type real

The user can change the precision of CVODE arithmetic from double to single by changing the typedef

```
typedef double real;      to      typedef float real;
```

and by changing the macro definitions

```
#define FLOAT  0          to          #define FLOAT  1
#define DOUBLE 1                      #define DOUBLE 0
```

in `llnltyps.h`. These macro definitions are used to enable `llnltyps.h` to branch on the setting of `real` at compile time.

Changing from double precision to single precision arithmetic also requires minor changes in the implementation file `llnlmath.c` for the LLNLMATH module which CVODE uses. 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 (if the user wants CVODE to perform only single precision arithmetic).

Within CVODE, 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 `real`. 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 `real` is `double` and it expands to `1.0F` if `real` is `float`. CVODE uses the `RCONST` macro for all its floating point constants.

A user program which uses the type `real` 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 `real` but not `RCONST`.) Users can, however, use the type `double` or `float` in their code (assuming the typedef for `real` matches this choice). Thus, a previously existing piece of ANSI C code can use CVODE without modifying the code to use `real`.

7.3 Changing Type integer

CVODE uses the type `integer` 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. The size `int` may be too small on a machine for a very large problem. In this case, the user should change the typedef

```
typedef int integer;       to       typedef long int integer;
```

and the macro definitions

```
#define INT      1         to        #define INT      0
#define LONG_INT 0                   #define LONG_INT 1
```

in `llnltyps.h`. In terms of the problem size N, and the bandwidths `mlower` and `mupper` in the case of the band module CVBAND, the largest integer that must be accommodated by the `integer` type is $N+$ `mlower` $+$ `mupper` 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 `integer` (assuming the typedef for `integer` matches this choice).

8 Demonstration Programs

As an aid to users of the CVODE package, two demonstration programs are provided as part of the package. These are more complex in their usage of CVODE than the sample programs given above, and include some self-checking for accuracy. They exercise all of the major method options of CVODE, and so provide a validation check for recipients of the package.

8.1 Direct Demonstration Program, cvdemd.c

This program solves two different problems, and includes self-checking features in both cases.

The first problem is the Van der Pol oscillator problem, a second-order ODE converted to two first-order ODEs, with a parameter that makes it mildly stiff. It is solved with eight different method choices: Adams and BDF methods, each in combination with functional iteration, a dense user-supplied Jacobian, a dense difference-quotient Jacobian, and a diagonal Jacobian approximation. Global errors are checked by stopping at two of the zeros of the exact solution.

The second problem is the semi-discrete form of a simple advection PDE in two dimensions, giving a linear ODE system of size 25. It has an analytic solution, and global errors are measured at each output time. It is also solved with eight different method choices: Adams and BDF methods, each in combination with functional iteration, a banded user-supplied Jacobian, a banded difference-quotient Jacobian, and a diagonal Jacobian approximation.

8.2 Krylov Demonstration Program, cvdemk.c

This program solves a 6-species predator-prey population model, based on a system of PDEs with diffusion and nonlinear species interaction rates in two dimensions. The PDEs are semi-discretized on a 6×6 mesh, giving an ODE system of size 216. This is solved with the BDF method and Newton iteration using the CVSPGMR linear solver. The preconditioner used is the product of two matrices—one based on a fixed number of Gauss-Seidel iterations using the diffusion coefficients only, and the other being a block-diagonal matrix based on the Jacobian of the interaction rates only. The latter uses a block-grouping scheme in which the 36 spatial mesh points are grouped into 4 groups, and only one 6×6 Jacobian block is computed in each group. Four different cases are run. The preconditioner is applied to both the left and right sides, and in each case, both modified and classical Gram-Schmidt options are run. The program prints performance statistics at each output point, and the full solution vector at selected output points.

The preconditioner setup routine `Precond` in this program is intended to be usable as a prototype for the general case of a block-diagonal preconditioner. It generates the blocks of the preconditioner by way of difference quotients, and calls to a separate routine that evaluates a block of the interaction rate vector at a single spatial point. It makes use of block-grouping information that is also completely general. A separate routine is used to set that information, based on a simple partitioning of the rectangular grid.

9 Overall Structure

The CVODE 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 CVODE package is shown in Figure 2. The basic elements of the structure are a module for the basic integration algorithm and a set of modules for the solution of linear systems that arise in the case of a stiff system. The central integration module CVode, implemented in the files `cvode.h` and `cvode.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.

At present, the package includes the following four CVODE 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 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 in the direct case, the references [2]-[3], together with the example and demonstration programs included with CVODE, offer considerable assistance in building preconditioners.

Each CVODE linear solver module consists of four routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) 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 CVODE module to each of the four 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 CVODE

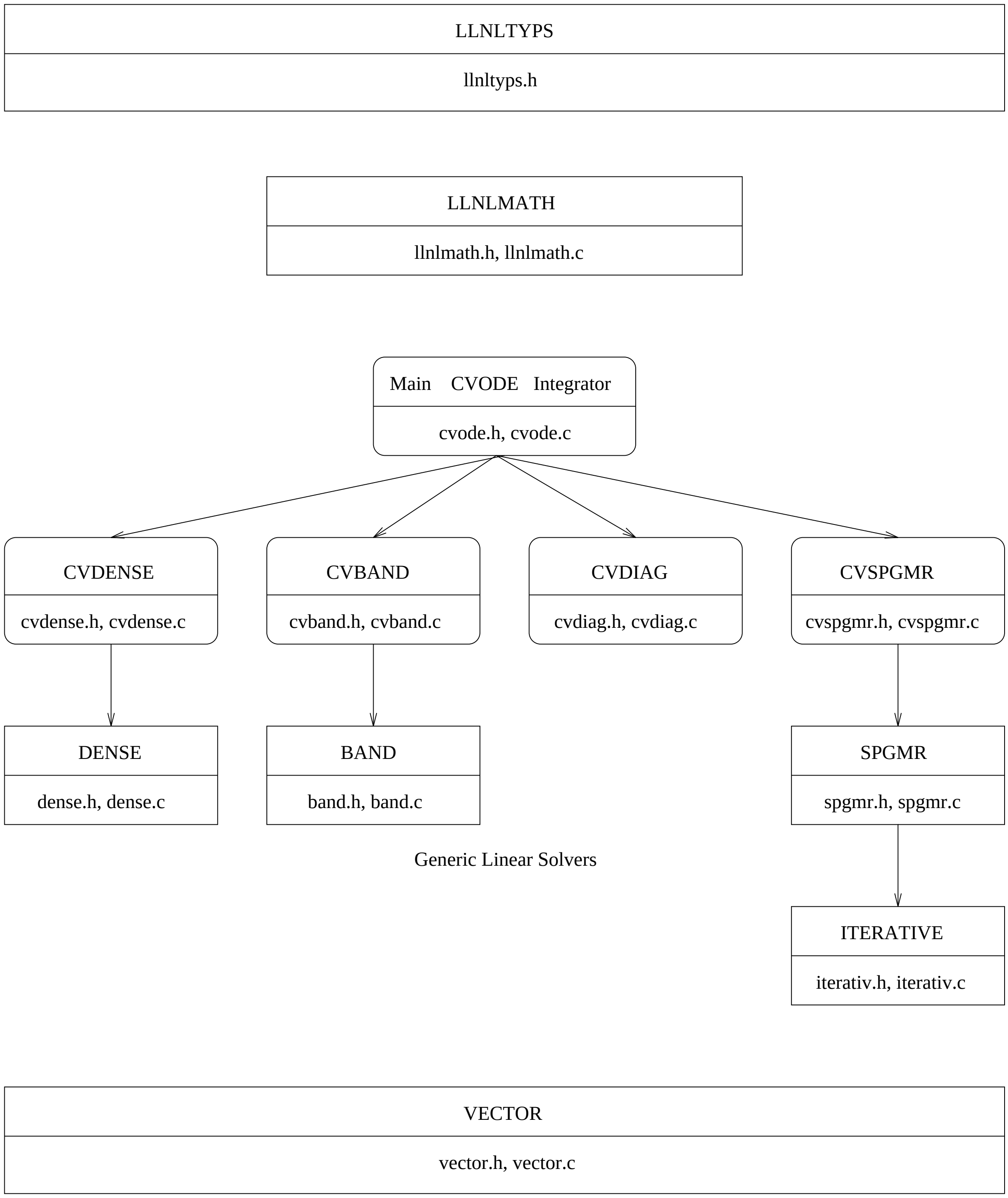

Figure 2: Overall structure diagram of the CVODE package. Modules specific to CVODE are distinguished by rounded boxes, while generic solver and auxiliary modules are in unrounded boxes.

context, whereas the generic solver is independent of the context. While the generic solvers here were generated with CVODE 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 CVODE package elsewhere.

Variation in the types of scalar variables is handled by way of a separate header file `llnltyps.h` which contains type definitions for `real` and `integer`. The default settings for `real` and `integer` are `double` and `int`, respectively. With this arrangement, the precision can be changed from double to single with minor changes to `llnltyps.h` and `llnlmath.c`.

The vectors y, f, etc. of length N in CVODE are objects of type `N_Vector`. This type declaration is contained in a separate module in the CVODE package, along with a collection of vector kernels which are used by both the central integrator module and the linear solver modules. These kernels perform operations such as linear sums, dot products, and weighted norms. This arrangement was made in anticipation of a multiprocessor extension of CVODE, in which the vector data would be distributed, with the `N_Vector` type containing a description of the data distribution. Changes in the `N_Vector` type would necessitate rewriting the vector kernels, and some parts of the dense and band linear solvers (both the CVODE-specific and generic parts); parts of the example programs which perform `N_Vector` accesses might also have to be revised.

All state information used by CVODE 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 CVODE 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 CVODE memory structure. The reentrancy of CVODE 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 CVODE package. Header and source files are described as a single unit.

In Table 2 below, we list the vector kernels in the VECTOR module within the CVODE package. There are 16 such kernels, excluding the debugging tool `N_VPrint`. All kernel names begin with "N_" to indicate that the kernels are operations on length N vectors (of type `N_Vector`), where N is the problem size. The table also shows, for each kernel, which of the code modules uses the kernel. The CVode column shows kernel usage within the main CVODE integrator module, while the remaining four columns show kernel usage within each of the four CVODE linear solvers. There is one subtlety in the CVSPGMR column hidden by the table. The dot product kernel `N_VDotProd`, for example, is not called within the implementation file `cvspgmr.c` for the CVSPGMR solver, yet we have marked it as "used" by CVSPGMR. This is because `N_VDotProd` is called within the implementation file `spgmr.c` for the generic SPGMR solver upon which the CVSPGMR solver is implemented. This issue does not arise for the other three CVODE 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. At this point, we should emphasize that the CVODE user does not need to know anything about the usage of vector kernels by the CVODE code modules in order to use CVODE. The information is presented as implementation details for the interested reader, and in anticipation of parallel extensions to CVODE.

File(s)	Description and/or Contents
llnltyps.h	types real, integer, and bool; constants FALSE, TRUE
llnlmath.h, .c	Math package for real precision arithmetic MIN, MAX, ABS, SQR, UnitRoundoff, RPowerI, RPowerR, RSqrt
vector.h, .c	Generic vector package Type N_Vector, N_VIth, N_VDATA N_Vector kernels including N_VNew, N_VFree
cvode.h, .c	Central CVODE integrator module CVodeMalloc, CVode, CVodeFree, CVodeDky
cvdense.h, .c	CVODE dense linear solver CVDENSE CVDense, CVDenseDQJac functions
cvband.h, .c	CVODE band linear solver CVBAND CVBand, CVBandDQJac functions
cvdiag.h, .c	CVODE diagonal linear solver CVDIAG CVDiag function
cvspgmr.h, .c	CVODE GMRES linear solver CVSPGMR CVSpgmr function
dense.h, .c	Generic DENSE linear solvers: large, small dense matrices Large: Type DenseMat, DENSE_ELEM, DENSE_COL Small: denalloc, gefa, gesl, denaddI, denfree
band.h, .c	Generic BAND linear solvers: large, small band matrices Large: Type BandMat, BAND_ELEM, BAND_COL, BAND_COL_ELEM Small: bandalloc, gbfa, gbsl, bandfree
iterativ.h, .c	Items common to generic iterative linear solvers ModifiedGS, ClassicalGS, QRfact, QRsol routines
spgmr.h, .c	Generic preconditioned GMRES linear solver

Table 1: List of files in CVODE package

Kernel	CVode	CVDENSE	CVBAND	CVDIAG	CVSPGMR
N_VNew	X			X	X
N_VFree	X			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	
N_VPrint					

Table 2: List of vector kernels and usage by CVODE code modules

10 Reference Guide

10.1 Main CVODE Integrator

The following seven sections contain documentation and declarations from the header file `cvode.h`. In the first, the options listed are the basic solution method parameters that are input to `CVodeMalloc` and `CVode`. Next is the specification of the right-hand side function `f`. The remaining sections describe the call lists for `CVodeMalloc`, `CVode`, `CVodeDky`, and `CVodeFree`, and the list of optional inputs and outputs.

10.1.1 Options

```
/******************************************************************
 *                                                                *
 * Enumerations for input parameters to CVodeMalloc and CVode.    *
 *----------------------------------------------------------------*
 * Symbolic constants for the lmm, iter, and itol input           *
 * parameters to CVodeMalloc, as well as the input parameter      *
 * itask to CVode, are given below.                               *
 *                                                                *
 * lmm  : The user of the CVODE package specifies whether to use  *
 *        the ADAMS or BDF (backward differentiation formula)     *
 *        linear multistep method. The BDF method is recommended  *
 *        for stiff problems, and the ADAMS method is recommended *
 *        for nonstiff problems.                                  *
 *                                                                *
 * iter : At each internal time step, a nonlinear equation must   *
 *        be solved. The user can specify either FUNCTIONAL       *
 *        iteration, which does not require linear algebra, or a  *
 *        NEWTON iteration, which requires the solution of linear *
 *        systems. In the NEWTON case, the user also specifies a  *
 *        CVODE linear solver. NEWTON is recommended in case of   *
 *        stiff problems.                                         *
 *                                                                *
 * itol : This parameter specifies the relative and absolute      *
 *        tolerance types to be used. The SS tolerance type means *
 *        a scalar relative and absolute tolerance, while the SV  *
 *        tolerance type means a scalar relative tolerance and a  *
 *        vector absolute tolerance (a potentially different      *
 *        absolute tolerance for each vector component).          *
 *                                                                *
 * itask : The itask input parameter to CVode indicates the job   *
 *         of the solver for the next user step. The NORMAL       *
 *         itask is to have the solver take internal steps until  *
 *         it has reached or just passed the user specified tout  *
 *         parameter. The solver then interpolates in order to    *
 *         return an approximate value of y(tout). The ONE_STEP   *
 *         option tells the solver to just take one internal step *
 *         and return the solution at the point reached by that   *
 *         step.                                                  *
 *                                                                *
 ******************************************************************/

enum { ADAMS, BDF };           /* lmm */

enum { FUNCTIONAL, NEWTON };   /* iter */
```

```
enum { SS, SV };              /* itol */

enum { NORMAL, ONE_STEP };    /* itask */
```

10.1.2 The Right Hand Side Function, Type RhsFn

```
/******************************************************************
 *                                                                *
 * Type : RhsFn                                                   *
 *----------------------------------------------------------------*
 * The f function which defines the right hand side of the ODE    *
 * system y'=f(t,y) must have type RhsFn.                         *
 * f takes as input the problem size N, the independent variable  *
 * value t, and the dependent variable vector y.  It stores 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 CVODE.)       *
 * 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 f does not have a return value.                        *
 *                                                                *
 ******************************************************************/

typedef void (*RhsFn)(integer N, real t, N_Vector y, N_Vector ydot,
                      void *f_data);
```

10.1.3 CVodeMalloc

```
/******************************************************************
 *                                                                *
 * Function : CVodeMalloc                                         *
 *----------------------------------------------------------------*
 * CVodeMalloc allocates and initializes memory for a problem to  *
 * to be solved by CVODE.                                         *
 *                                                                *
 * N       is the number of equations in the ODE system.          *
 *                                                                *
 * f       is the right hand side function in y' = f(t,y).        *
 *                                                                *
 * t0      is the initial value of t.                             *
 *                                                                *
 * y0      is the initial condition vector y(t0).                 *
 *                                                                *
 * lmm     is the type of linear multistep method to be used.     *
 *            The legal values are ADAMS and BDF (see previous    *
 *            description).                                       *
 *                                                                *
 * iter    is the type of iteration used to solve the nonlinear   *
 *            system that arises during each internal time step.  *
 *            The legal values are FUNCTIONAL and NEWTON.         *
 *                                                                *
 * itol    is the type of tolerances to be used.                  *
 *            The legal values are:                               *
 *               SS (scalar relative and absolute  tolerances),   *
```

```
 *              SV (scalar relative tolerance and vector        *
 *                  absolute tolerance).                        *
 *                                                              *
 * reltol  is a pointer to the relative tolerance scalar.       *
 *                                                              *
 * abstol  is a pointer to the absolute tolerance scalar or     *
 *            an N_Vector tolerance.                            *
 *                                                              *
 * f_data  is a pointer to user data that will be passed to the *
 *             user's f function every time f is called.        *
 *                                                              *
 * errfp   is the file pointer for an error file where all CVODE *
 *            warning and error messages will be written. This  *
 *            parameter can be stdout (standard output), stderr *
 *            (standard error), a file pointer (corresponding to *
 *            a user error file opened for writing) returned by *
 *            fopen, or NULL. If the user passes NULL, then all *
 *            messages will be written to standard output.      *
 *                                                              *
 * optIn   is a flag indicating whether there are any optional  *
 *            inputs from the user in the arrays iOpt and rOpt. *
 *            Pass FALSE to indicate no optional inputs and TRUE *
 *            to indicate that optional inputs are present.     *
 *                                                              *
 * iopt    is the user-allocated array (of size OPT_SIZE given  *
 *            later) that will hold optional integer inputs and *
 *            outputs.  The user can pass NULL if he/she does not *
 *            wish to use optional integer inputs or outputs.   *
 *            If optIn is TRUE, the user should preset to 0 those *
 *            locations for which default values are to be used. *
 *                                                              *
 * ropt    is the user-allocated array (of size OPT_SIZE given  *
 *            later) that will hold optional real inputs and    *
 *            outputs.  The user can pass NULL if he/she does not *
 *            wish to use optional real inputs or outputs.      *
 *            If optIn is TRUE, the user should preset to 0.0 the *
 *            locations for which default values are to be used. *
 *                                                              *
 * machEnv is a pointer to machine environment-specific         *
 *            information.                                      *
 *                                                              *
 * Note: The tolerance values may be changed in between calls to *
 *       CVode for the same problem. These values refer to      *
 *       (*reltol) and either (*abstol), for a scalar absolute  *
 *       tolerance, or the components of abstol, for a vector   *
 *       absolute tolerance.                                    *
 *                                                              *
 * If successful, CVodeMalloc returns a pointer to initialized  *
 * problem memory. This pointer should be passed to CVode. If   *
 * an initialization error occurs, CVodeMalloc prints an error  *
 * message to the file specified by errfp and returns NULL.     *
 *                                                              *
 ****************************************************************/


void *CVodeMalloc(integer N, RhsFn f, real t0, N_Vector y0, int lmm, int iter,
                  int itol, real *reltol, void *abstol, void *f_data,
                  FILE *errfp, bool optIn, long int iopt[], real ropt[],
                  void *machEnv);
```

10.1.4 CVode

```
/******************************************************************
 *                                                                *
 * Function : CVode                                               *
 *----------------------------------------------------------------*
 * CVode integrates the ODE over an interval in t.                *
 * If itask is NORMAL, then the solver integrates from its        *
 * current internal t value to a point at or beyond tout, then    *
 * interpolates to t = tout and returns y(tout) in the user-      *
 * allocated vector yout. If itask is ONE_STEP, then the solver   *
 * takes one internal time step and returns in yout the value of  *
 * y at the new internal time. In this case, tout is used only    *
 * during the first call to CVode to determine the direction of   *
 * integration and the rough scale of the problem. In either      *
 * case, the time reached by the solver is placed in (*t). The    *
 * user is responsible for allocating the memory for this value.  *
 *                                                                *
 * cvode_mem is the pointer to CVODE memory returned by           *
 *              CVodeMalloc.                                      *
 *                                                                *
 * tout  is the next time at which a computed solution is desired *
 *                                                                *
 * yout  is the computed solution vector. In NORMAL mode with no  *
 *          errors, yout=y(tout).                                 *
 *                                                                *
 * t     is a pointer to a real location. CVode sets (*t) to the  *
 *          time reached by the solver and returns yout=y(*t).    *
 *                                                                *
 * itask is either NORMAL or ONE_STEP mode. These two modes have  *
 *          described above.                                      *
 *                                                                *
 * The return values for CVode are defined later in this file.    *
 * Here is a brief description of each return value:              *
 *                                                                *
 * SUCCESS       : CVode succeeded.                               *
 *                                                                *
 * 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 < 0 during        *
 *                 internal time-stepping. The ILL_INPUT flag     *
 *                 will also be returned if the linear solver     *
 *                 routine CV--- (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 init 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| = hmin.                      *
 *                                                                *
```

```
 * CONV_FAILURE  : Convergence test failures occurred too many    *
 *                 times (= MXNCF = 10) during one internal time  *
 *                 step or occurred with |h| = hmin.              *
 *                                                                *
 * 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.                          *
 *                                                                *
 ******************************************************************/


int CVode(void *cvode_mem, real tout, N_Vector yout, real *t, int itask);


/* CVode return values */

enum { SUCCESS=0, CVODE_NO_MEM=-1, ILL_INPUT=-2, TOO_MUCH_WORK=-3,
       TOO_MUCH_ACC=-4, ERR_FAILURE=-5, CONV_FAILURE=-6,
       SETUP_FAILURE=-7, SOLVE_FAILURE=-8 };
```

10.1.5 CVodeDky

```
/******************************************************************
 *                                                                *
 * Function : CVodeDky                                            *
 *----------------------------------------------------------------*
 * CVodeDky computes the kth derivative of the y function at      *
 * time t, where tn-hu <= t <= tn, tn denotes the current         *
 * internal time reached, and hu is the last internal step size   *
 * successfully used by the solver. The user may request          *
 * k=0, 1, ..., qu, where qu is the current order. The            *
 * derivative vector is returned in dky. This vector must be      *
 * allocated by the caller. It is only legal to call this         *
 * function after a successful return from CVode.                 *
 *                                                                *
 * cvode_mem is the pointer to CVODE memory returned by           *
 *             CVodeMalloc.                                       *
 *                                                                *
 * t   is the time at which the kth derivative of y is evaluated. *
 *        The legal range for t is [tn-hu,tn] as described above. *
 *                                                                *
 * k   is the order of the derivative of y to be computed. The    *
 *        legal range for k is [0,qu] as described above.         *
 *                                                                *
 * dky is the output derivative vector [(D_k)y](t).               *
 *                                                                *
 * The return values for CVodeDky are defined later in this file. *
 * Here is a brief description of each return value:             *
 *                                                                *
 * OKAY : CVodeDky succeeded.                                     *
 *                                                                *
 * BAD_K : k is not in the range 0, 1, ..., qu.                   *
 *                                                                *
 * BAD_T : t is not in the interval [tn-hu,tn].                   *
 *                                                                *
 * BAD_DKY : The dky argument was NULL.                           *
 *                                                                *
```

```
 * DKY_NO_MEM : The cvode_mem argument was NULL.              *
 *                                                            *
 **************************************************************/


int CVodeDky(void *cvode_mem, real t, int k, N_Vector dky);


/* CVodeDky return values */

enum { OKAY=0, BAD_K=-1, BAD_T=-2, BAD_DKY=-3, DKY_NO_MEM=-4 };
```

10.1.6 CVodeFree

```
/**************************************************************
 *                                                            *
 * Function : CVodeFree                                       *
 *------------------------------------------------------------*
 * CVodeFree frees the problem memory cvode_mem allocated by  *
 * CVodeMalloc.  Its only argument is the pointer cvode_mem   *
 * returned by CVodeMalloc.                                   *
 *                                                            *
 **************************************************************/

void CVodeFree(void *cvode_mem);
```

10.1.7 Optional Inputs and Outputs

```
/**************************************************************
 *                                                            *
 * Optional Inputs and Outputs                                *
 *------------------------------------------------------------*
 * 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 declaration should look like:                *
 *                                                            *
 * long int iopt[OPT_SIZE];                                   *
 * real     ropt[OPT_SIZE];                                   *
 *                                                            *
 * The enumerations below the OPT_SIZE definition             *
 * are indices into the iopt and ropt arrays. Here is a brief *
 * description of the contents of these positions:            *
 *                                                            *
 * iopt[MAXORD] : maximum lmm order to be used by the solver. *
 *                Optional input. (Default = 12 for ADAMS, 5 for *
 *                BDF).                                       *
 *                                                            *
 * iopt[MXSTEP] : maximum number of internal steps to be taken by *
 *                the solver in its attempt to reach tout.    *
 *                Optional input. (Default = 500).            *
 *                                                            *
 * iopt[MXHNIL] : maximum number of warning messages issued   *
 *                by the solver that t+h==t on the next internal *
 *                step. Optional input. (Default = 10).       *
 *                                                            *
```

```
 * iopt[NST]     : cumulative number of internal steps taken by    *
 *                 the solver (total so far).  Optional output.    *
 *                                                                 *
 * iopt[NFE]     : number of calls to the user's f function.       *
 *                 Optional output.                                *
 *                                                                 *
 * iopt[NSETUPS] : number of calls made to the linear solver's     *
 *                  setup routine. Optional output.                *
 *                                                                 *
 * iopt[NNI]      : number of NEWTON iterations performed.         *
 *                  Optional output.                               *
 *                                                                 *
 * iopt[NCFN]     : number of nonlinear convergence failures       *
 *                  that have occurred. Optional output.           *
 *                                                                 *
 * iopt[NETF]     : number of local error test failures that       *
 *                  have occurred. Optional output.                *
 *                                                                 *
 * iopt[QU]       : order used during the last internal step.      *
 *                  Optional output.                               *
 *                                                                 *
 * iopt[QCUR]     : order to be used on the next internal step.    *
 *                  Optional output.                               *
 *                                                                 *
 * iopt[LENRW]    : size of required CVODE internal real work      *
 *                  space, in real words.  Optional output.        *
 *                                                                 *
 * iopt[LENIW]    : size of required CVODE internal integer work   *
 *                  space, in integer words.  Optional output.     *
 *                                                                 *
 * ropt[H0]       : initial step size. Optional input.             *
 *                                                                 *
 * ropt[HMAX]     : maximum absolute value of step size allowed.   *
 *                  Optional input. (Default is infinity).         *
 *                                                                 *
 * ropt[HMIN]     : minimum absolute value of step size allowed.   *
 *                  Optional input. (Default is 0.0).              *
 *                                                                 *
 * ropt[HU]       : step size for the last internal step.          *
 *                  Optional output.                               *
 *                                                                 *
 * ropt[HCUR]     : step size to be attempted on the next internal *
 *                  step. Optional output.                         *
 *                                                                 *
 * ropt[TCUR]     : current internal time reached by the solver.   *
 *                  Optional output.                               *
 *                                                                 *
 * ropt[TOLSF]    : a suggested factor by which the user's         *
 *                  tolerances should be scaled when too much      *
 *                  accuracy has been requested for some internal  *
 *                  step. Optional output.                         *
 *                                                                 *
 *******************************************************************/

/* iopt, ropt array sizes */

#define OPT_SIZE 40
```

10.2 CVODE Linear Solvers

Each of the linear system solvers in the CVODE package (of which there are currently four) is described in the next sections.

10.2.1 CVDENSE

The following sections contain documentation and declarations from the header file `cvdense.h`.

10.2.1.1 CVDense

```
/******************************************************************
 *                                                                *
 * Function : CVDense                                             *
 *----------------------------------------------------------------*
 * A call to the CVDense function links the main CVODE integrator *
 * with the CVDENSE linear solver.                                *
 *                                                                *
 * cvode_mem is the pointer to CVODE memory returned by           *
 *              CVodeMalloc.                                      *
 *                                                                *
 * djac is the dense Jacobian approximation routine to be used.   *
 *         A user-supplied djac routine must be of type           *
 *         CVDenseJacFn. Pass NULL for djac to use the default    *
 *         difference quotient routine CVDenseDQJac supplied      *
 *         with this solver.                                      *
 *                                                                *
 * jac_data is a pointer to user data which is passed to the      *
 *         djac routine every time it is called.                  *
 *                                                                *
 ******************************************************************/

void CVDense(void *cvode_mem, CVDenseJacFn djac, void *jac_data);
```

10.2.1.2 Type CVDenseJacFn

```
/******************************************************************
 *                                                                *
 * Type : CVDenseJacFn                                            *
 *----------------------------------------------------------------*
 * A dense Jacobian approximation function Jac must have the      *
 * prototype given below. Its parameters are:                     *
 *                                                                *
 * N is the length of all vector arguments.                       *
 *                                                                *
 * J is the dense matrix (of type DenseMat) that will be loaded   *
 * by a CVDenseJacFn with an approximation to the Jacobian matrix *
 * J = (df_i/dy_j) at the point (t,y).                            *
 * J is preset to zero, so only the nonzero elements need to be   *
 * loaded. Two efficient ways to load J are:                      *
 *                                                                *
 * (1) (with macros - no explicit data structure references)      *
```

```
 *     for (j=0; j < N; j++) {                                     *
 *       col_j = DENSE_COL(J,j);                                   *
 *       for (i=0; i < N; i++) {                                   *
 *         generate J_ij = the (i,j)th Jacobian element            *
 *         col_j[i] = J_ij;                                        *
 *       }                                                         *
 *     }                                                           *
 *                                                                 *
 * (2) (without macros - explicit data structure references)      *
 *     for (j=0; j < N; j++) {                                     *
 *       col_j = (J->data)[j];                                     *
 *       for (i=0; i < N; i++) {                                   *
 *         generate J_ij = the (i,j)th Jacobian element            *
 *         col_j[i] = J_ij;                                        *
 *       }                                                         *
 *     }                                                           *
 *                                                                 *
 * The DENSE_ELEM(A,i,j) macro is appropriate for use in small    *
 * problems in which efficiency of access is NOT a major concern. *
 *                                                                 *
 * f is the right hand side function for the ODE problem.         *
 *                                                                 *
 * f_data is a pointer to user data to be passed to f, the same   *
 *        as the F_data parameter passed to CVodeMalloc.          *
 *                                                                 *
 * 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).                                        *
 *                                                                 *
 * ewt is the error weight vector.                                 *
 *                                                                 *
 * h is a tentative step size in t.                                *
 *                                                                 *
 * uround is the machine unit roundoff.                            *
 *                                                                 *
 * jac_data is a pointer to user data - the same as the jac_data  *
 *          parameter passed to CVDense.                           *
 *                                                                 *
 * nfePtr is a pointer to the memory location containing the      *
 * CVODE problem data nfe = number of calls to f. The Jacobian    *
 * routine should update this counter by adding on the number     *
 * of f calls made in order to approximate the Jacobian, if any.  *
 * For example, if the routine calls f a total of N times, then   *
 * the update is *nfePtr += N.                                     *
 *                                                                 *
 * vtemp1, vtemp2, and vtemp3 are pointers to memory allocated    *
 * for vectors of length N which can be used by a CVDenseJacFn    *
 * as temporary storage or work space.                             *
 *                                                                 *
 ******************************************************************/

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

10.2.1.3 Statistics

```
/******************************************************************
 *                                                                *
 * CVDENSE solver statistics indices                              *
 *----------------------------------------------------------------*
 * The following enumeration gives a symbolic name to each        *
 * CVDENSE statistic. The symbolic names are used as indices into *
 * the iopt and ropt arrays passed to CVodeMalloc.                *
 * The CVDENSE statistics are:                                    *
 *                                                                *
 * iopt[DENSE_NJE] : number of Jacobian evaluations, i.e. of      *
 *                   calls made to the dense Jacobian routine     *
 *                   (default or user-supplied).                  *
 *                                                                *
 * iopt[DENSE_LRW] : size (in real words) of real workspace       *
 *                   matrices and vectors used by this solver.    *
 *                                                                *
 * iopt[DENSE_LIW] : size (in integer words) of integer           *
 *                   workspace vectors used by this solver.       *
 *                                                                *
 ******************************************************************/
```

10.2.2 CVBAND

The following sections contain documentation and declarations from the header file `cvband.h`.

10.2.2.1 CVBand

```
/******************************************************************
 *                                                                *
 * Function : CVBand                                              *
 *----------------------------------------------------------------*
 * A call to the CVBand function links the main CVODE integrator  *
 * with the CVBAND linear solver.                                 *
 *                                                                *
 * cvode_mem is the pointer to CVODE memory returned by           *
 *              CVodeMalloc.                                      *
 *                                                                *
 * mupper is the upper bandwidth of the band Jacobian             *
 *          approximation.                                        *
 *                                                                *
 * mlower is the lower bandwidth of the band Jacobian             *
 *          approximation.                                        *
 *                                                                *
 *                                                                *
 * bjac is the band Jacobian approximation routine to be used.    *
 *          A user-supplied bjac routine must be of type          *
 *          CVBandJacFn. Pass NULL for bjac to use the default    *
 *          difference quotient routine CVBandDQJac supplied      *
 *          with this solver.                                     *
 *                                                                *
 * jac_data is a pointer to user data which is passed to the      *
 *          bjac routine every time it is called.                 *
 *                                                                *
 ******************************************************************/
```

```
void CVBand(void *cvode_mem, integer mupper, integer mlower, CVBandJacFn bjac,
            void *jac_data);
```

10.2.2.2 Type CVBandJacFn

```
/******************************************************************
 *                                                                *
 * Type : CVBandJacFn                                             *
 *----------------------------------------------------------------*
 * A band Jacobian approximation function Jac must have the       *
 * prototype given below. Its parameters are:                     *
 *                                                                *
 * N is the length of all vector arguments.                       *
 *                                                                *
 * mupper is the upper half-bandwidth of the approximate banded   *
 * Jacobian. This parameter is the same as the mupper parameter   *
 * passed by the user to the CVBand function.                     *
 *                                                                *
 * mlower is the lower half-bandwidth of the approximate banded   *
 * Jacobian. This parameter is the same as the mlower parameter   *
 * passed by the user to the CVBand function.                     *
 *                                                                *
 * J is the band matrix (of type BandMat) that will be loaded     *
 * by a CVBandJacFn with an approximation to the Jacobian matrix  *
 * J = (df_i/dy_j) at the point (t,y).                            *
 * J is preset to zero, so only the nonzero elements need to be   *
 * loaded. Three efficient ways to load J are:                    *
 *                                                                *
 * (1) (with macros - no explicit data structure references)      *
 *    for (j=0; j < N; j++) {                                     *
 *       col_j = BAND_COL(J,j);                                   *
 *       for (i=j-mupper; i <= j+mlower; i++) {                   *
 *         generate J_ij = the (i,j)th Jacobian element           *
 *         BAND_COL_ELEM(col_j,i,j) = J_ij;                       *
 *       }                                                        *
 *     }                                                          *
 *                                                                *
 * (2) (with BAND_COL macro, but without BAND_COL_ELEM macro)     *
 *    for (j=0; j < N; j++) {                                     *
 *       col_j = BAND_COL(J,j);                                   *
 *       for (k=-mupper; k <= mlower; k++) {                      *
 *         generate J_ij = the (i,j)th Jacobian element, i=j+k    *
 *         col_j[k] = J_ij;                                       *
 *       }                                                        *
 *     }                                                          *
 *                                                                *
 * (3) (without macros - explicit data structure references)      *
 *     offset = J->smu;                                           *
 *     for (j=0; j < N; j++) {                                    *
 *       col_j = ((J->data)[j])+offset;                           *
 *       for (k=-mupper; k <= mlower; k++) {                      *
 *         generate J_ij = the (i,j)th Jacobian element, i=j+k    *
 *         col_j[k] = J_ij;                                       *
 *       }                                                        *
 *     }                                                          *
 * Caution: J->smu is generally NOT the same as mupper.           *
 *                                                                *
 * The BAND_ELEM(A,i,j) macro is appropriate for use in small     *
```

```
 * problems in which efficiency of access is NOT a major concern. *
 *                                                                *
 * f is the right hand side function for the ODE problem.         *
 *                                                                *
 * f_data is a pointer to user data to be passed to f, the same   *
 *        as the F_data parameter passed to CVodeMalloc.          *
 *                                                                *
 * 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).                                       *
 *                                                                *
 * ewt is the error weight vector.                                *
 *                                                                *
 * h is a tentative step size in t.                               *
 *                                                                *
 * uround is the machine unit roundoff.                           *
 *                                                                *
 * jac_data is a pointer to user data - the same as the jac_data  *
 *          parameter passed to CVBand.                           *
 *                                                                *
 * nfePtr is a pointer to the memory location containing the      *
 * CVODE problem data nfe = number of calls to f. The Jacobian    *
 * routine should update this counter by adding on the number     *
 * of f calls made in order to approximate the Jacobian, if any.  *
 * For example, if the routine calls f a total of N times, then   *
 * the update is *nfePtr += N.                                    *
 *                                                                *
 * vtemp1, vtemp2, and vtemp3 are pointers to memory allocated    *
 * for vectors of length N which can be used by a CVBandJacFn     *
 * as temporary storage or work space.                            *
 *                                                                *
 *****************************************************************/

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

10.2.2.3 Statistics

```
/******************************************************************
 *                                                                *
 * CVBAND solver statistics indices                               *
 *----------------------------------------------------------------*
 * The following enumeration gives a symbolic name to each        *
 * CVBAND statistic. The symbolic names are used as indices into  *
 * the iopt and ropt arrays passed to CVodeMalloc.                *
 * The CVBAND statistics are:                                     *
 *                                                                *
 * iopt[BAND_NJE] : number of Jacobian evaluations, i.e. of       *
 *                  calls made to the band Jacobian routine       *
 *                  (default or user-supplied).                   *
 *                                                                *
 * iopt[BAND_LRW] : size (in real words) of real workspace        *
 *                  matrices and vectors used by this solver.     *
```

```
 *                                                              *
 * iopt[BAND_LIW] : size (in integer words) of integer          *
 *                  workspace vectors used by this solver.      *
 *                                                              *
 ****************************************************************/
```

10.2.3 CVDIAG

The following sections contain documentation and declarations from the header file `cvdiag.h`.

10.2.3.1 CVDiag

```
/****************************************************************
 *                                                              *
 * Function : CVDiag                                            *
 *--------------------------------------------------------------*
 * A call to the CVDiag function links the main CVODE integrator *
 * with the CVDIAG linear solver.                               *
 *                                                              *
 * cvode_mem is the pointer to CVODE memory returned by         *
 *              CVodeMalloc.                                    *
 *                                                              *
 ****************************************************************/

void CVDiag(void *cvode_mem);
```

10.2.3.2 Statistics

```
/****************************************************************
 *                                                              *
 * CVDIAG solver statistics indices                             *
 *--------------------------------------------------------------*
 * The following enumeration gives a symbolic name to each      *
 * CVDIAG statistic. The symbolic names are used as indices into *
 * the iopt and ropt arrays passed to CVodeMalloc.              *
 * The CVDIAG statistics are:                                   *
 *                                                              *
 * iopt[DIAG_LRW] : size (in real words) of real workspace      *
 *                  vectors used by this solver.                *
 *                                                              *
 * iopt[DIAG_LIW] : size (in integer words) of integer          *
 *                  workspace vectors used by this solver.      *
 *                                                              *
 * The number of diagonal approximate Jacobians formed is equal  *
 * to the number of CVDiagSetup calls. This number is available  *
 * in cv_iopt[NSETUPS].                                         *
 *                                                              *
 ****************************************************************/
```

10.2.4 CVSPGMR

The following sections contain documentation and declarations from the header file `cvspgmr.h`.

10.2.4.1 CVSpgmr

```
/******************************************************************
 *                                                                *
 * Function : CVSpgmr                                             *
 *----------------------------------------------------------------*
 * A call to the CVSpgmr function links the main CVODE integrator *
 * with the CVSPGMR linear solver.                                *
 *                                                                *
 * cvode_mem is the pointer to CVODE memory returned by           *
 *             CVodeMalloc.                                       *
 *                                                                *
 * pretype   is the type of user preconditioning to be done.      *
 *             This must be one of the four enumeration constants *
 *             NONE, LEFT, RIGHT, or BOTH defined in iterativ.h.  *
 *             These correspond to no preconditioning,            *
 *             left preconditioning only, right preconditioning   *
 *             only, and both left and right preconditioning,     *
 *             respectively.                                      *
 *                                                                *
 * gstype    is the type of Gram-Schmidt orthogonalization to be  *
 *           used. This must be one of the two enumeration        *
 *           constants MODIFIED_GS or CLASSICAL_GS defined in     *
 *           iterativ.h. These correspond to using modified       *
 *           Gram-Schmidt and classical Gram-Schmidt,             *
 *           respectively.                                        *
 *                                                                *
 * maxl      is the maximum Krylov dimension. This is an          *
 *             optional input to the CVSPGMR solver. Pass 0 to    *
 *             use the default value MIN(N, CVSPGMR_MAXL=5).      *
 *                                                                *
 * delt      is the factor by which the tolerance on the          *
 *             nonlinear iteration is multiplied to get a         *
 *             tolerance on the linear iteration. This is an      *
 *             optional input to the CVSPGMR solver. Pass 0 to    *
 *             use the default value CVSPGMR_DELT = 0.05.         *
 *                                                                *
 * precond   is the user's preconditioner routine. It is used to  *
 *             evaluate and preprocess any Jacobian-related data  *
 *             needed by the psolve routine.  See the             *
 *             documentation for the type CVSpgmrPrecondFn for    *
 *             full details.  Pass NULL if no such setup of       *
 *             Jacobian data is required.  A precond routine is   *
 *             NOT required for any of the four possible values   *
 *             of pretype.                                        *
 *                                                                *
 * psolve    is the user's preconditioner solve routine. It is    *
 *             used to solve Pz=r, where P is a preconditioner    *
 *             matrix.  See the documentation for the type        *
 *             CVSpgmrPSolveFn for full details.  The only case   *
 *             in which psolve is allowed to be NULL is when      *
 *             pretype is NONE.  A valid psolve function must be  *
 *             supplied when any preconditioning is to be done.   *
 *                                                                *
 * P_data    is a pointer to user preconditioner data. This       *
```

```
 *             pointer is passed to precond and psolve every time *
 *             these routines are called.                         *
 *                                                                *
 ******************************************************************/

void CVSpgmr(void *cvode_mem, int pretype, int gstype, int maxl, real delt,
             CVSpgmrPrecondFn precond, CVSpgmrPSolveFn psolve, void *P_data);
```

10.2.4.2 Type CVSpgmrPrecondFn

```
/******************************************************************
 *                                                                *
 * Type : CVSpgmrPrecondFn                                        *
 *----------------------------------------------------------------*
 * The user-supplied preconditioner setup function Precond and    *
 * the user-supplied preconditioner solve function PSolve         *
 * together must define left and right preconditoner matrices     *
 * P1 and P2 (either of which may be trivial), such that the      *
 * product P1*P2 is an approximation to the Newton matrix         *
 * M = I - gamma*J.  Here J is the system Jacobian J = df/dy,     *
 * and gamma is a scalar proportional to the integration step     *
 * size h.  The solution of systems P z = r, with P = P1 or P2,   *
 * is to be carried out by the PSolve function, and Precond is    *
 * to do any necessary setup operations.                          *
 *                                                                *
 * The user-supplied preconditioner setup function Precond        *
 * is to evaluate and preprocess any Jacobian-related data        *
 * needed by the preconditioner solve function PSolve.            *
 * This might include forming a crude approximate Jacobian,       *
 * and performing an LU factorization on the resulting            *
 * approximation to M.  This function will not be called in       *
 * advance of every call to PSolve, but instead will be called    *
 * only as often as necessary to achieve convergence within the   *
 * Newton iteration in CVODE.  If the PSolve function needs no    *
 * preparation, the Precond function can be NULL.                 *
 *                                                                *
 * For greater efficiency, the Precond function may save          *
 * Jacobian-related data and reuse it, rather than generating it  *
 * from scratch.  In this case, it should use the input flag jok  *
 * to decide whether to recompute the data, and set the output    *
 * flag *jcurPtr accordingly.                                     *
 *                                                                *
 * The error weight vector ewt, step size h, and unit roundoff    *
 * uround are provided to the Precond function for possible use   *
 * in approximating Jacobian data, e.g. by difference quotients.  *
 *                                                                *
 * A function Precond must have the prototype given below.        *
 * 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, *
 *            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, as follows:              *
 *           jok == FALSE means recompute Jacobian-related data  *
 *                  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 Precond 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 by Precond as follows:                      *
 *         Set *jcurPtr = TRUE if Jacobian data was recomputed.  *
 *         Set *jcurPtr = 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    *
 *           CVODE problem data nfe = number of calls to f.      *
 *           The Precond routine should update this counter by   *
 *           adding on the number of f calls made in order to    *
 *           approximate the Jacobian, 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. *
 *                                                               *
 *                                                               *
 * Returned value:                                               *
 * The value to be returned by the Precond function is a flag    *
 * indicating whether it was successful.  This value should be   *
 *   0   if successful,                                          *
 *   > 0 for a recoverable error (step will be retried),         *
 *   < 0 for an unrecoverable error (integration is halted).     *
 *                                                               *
 *****************************************************************/

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

10.2.4.3 Type CVSpgmrPSolveFn

```
/******************************************************************
 *                                                                *
 * Type : CVSpgmrPSolveFn                                         *
 *----------------------------------------------------------------*
 * The user-supplied preconditioner solve function PSolve         *
 * is to solve a linear system P z = r in which the matrix P is   *
 * one of the preconditioner matrices P1 or P2, depending on the  *
 * type of preconditioning chosen.                                *
 *                                                                *
 * A function PSolve must have the prototype given below.         *
 * 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 by PSolve 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 for use by PSolve if it uses      *
 *          an iterative method in its solution.  In that case,   *
 *          the residual vector Res = r - P z of the system       *
 *          should be made less than delta in weighted L2 norm,   *
 *          i.e., sqrt [ Sum (Res[i]*ewt[i])^2 ] < delta .        *
 *                                                                *
 * nfePtr is a pointer to the memory location containing the      *
 *          CVODE problem data nfe = number of calls to f. The    *
 *          PSolve 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 PSolve is to use    *
 *          the left preconditioner P1 or right preconditioner    *
 *          P2: lr = 1 means use P1, and lr = 2 means use P2.     *
 *                                                                *
 * P_data is a pointer to user data - the same as the P_data      *
 *          parameter passed to CVSpgmr.                          *
 *                                                                *
 * z      is the output vector computed by PSolve.                *
 *                                                                *
 * Returned value:                                                *
 * The value to be returned by the PSolve function is a flag      *
 * indicating whether it was successful.  This value should be    *
 *   0 if successful,                                             *
 *   positive for a recoverable error (step will be retried),     *
 *   negative for an unrecoverable error (integration is halted). *
 *                                                                *
 ******************************************************************/
```

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

10.2.4.4 Statistics

```
/******************************************************************
 *                                                                *
 * CVSPGMR solver statistics indices                              *
 *----------------------------------------------------------------*
 * The following enumeration gives a symbolic name to each        *
 * CVSPGMR statistic. The symbolic names are used as indices into *
 * the iopt and ropt arrays passed to CVodeMalloc.                *
 * The CVSPGMR statistics are:                                    *
 *                                                                *
 * iopt[SPGMR_NPE]  : number of preconditioner evaluations,       *
 *                    i.e. of calls made to user's precond        *
 *                    function with jok == FALSE.                 *
 *                                                                *
 * iopt[SPGMR_NLI]  : number of linear iterations.                *
 *                                                                *
 * iopt[SPGMR_NPS]  : number of calls made to user's psolve       *
 *                    function.                                   *
 *                                                                *
 * iopt[SPGMR_NCFL] : number of linear convergence failures.      *
 *                                                                *
 * iopt[SPGMR_LRW]  : size (in real words) of real workspace      *
 *                    vectors and small matrices used by this     *
 *                    solver.                                     *
 *                                                                *
 * iopt[SPGMR_LIW]  : size (in integer words) of integer          *
 *                    workspace vectors used by this solver.      *
 *                                                                *
 ******************************************************************/
```

10.3 Generic Packages

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

- The VECTOR package, which includes the `N_Vector` type and a collection of kernels that perform operations on `N_Vector` vectors.
- 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 CVODE. The functions for small dense and band 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.

10.3.1 VECTOR

The following sections contain documentation and declarations from the header file `vector.h`.

```
/******************************************************************
 *                                                                *
 * File         : vector.h                                        *
 * Programmers  : Scott D. Cohen and Alan C. Hindmarsh @ LLNL     *
 * Last Modified : 1 September 1994                               *
 *----------------------------------------------------------------*
 *                                                                *
 * This is the header file for a generic VECTOR package. It       *
 * exports the type N_Vector.                                     *
 *                                                                *
 * Part I of this file contains declarations which are specific   *
 * to the particular machine environment in which this version    *
 * of the vector package is to be used. This includes the         *
 * typedef for the type N_Vector, as well as accessor macros      *
 * that allow the user to use efficiently the type N_Vector       *
 * without making explicit references to its underlying           *
 * representation. The underlying type of N_Vector will always    *
 * be some pointer type.                                          *
 *                                                                *
 * Part II of this file contains the prototypes for the vector    *
 * kernels which operate on the type N_Vector. These prototypes   *
 * are fixed for all implementations of the vector package. The   *
 * definitions of the types real and integer are in the header    *
 * file llnltyps.h and these may be changed according to the      *
 * user's needs. The llnltyps.h file also contains the            *
 * definition for the type bool (short for boolean) that is the   *
 * return type for the routine N_VInvTest.                        *
 *                                                                *
 * Important Note: N_Vector arguments to arithmetic kernels       *
 * need not be distinct. Thus, for example, the call              *
 *                                                                *
 *         N_VLinearSum(a,x,b,y,y);    y <- ax+by                 *
 *                                                                *
 * is legal.                                                      *
 *                                                                *
 * This version of vector.h is for the ordinary sequential        *
 * machine environment. In the documentation given below, N is    *
 * the length of all N_Vector parameters and x[i] denotes the     *
 * ith component of the N_Vector x, where 0 <= i <= N-1.          *
 *                                                                *
 ******************************************************************/
```

10.3.1.1 Type N_Vector

```
/*************************************************************
 *                                                           *
 * Type: N_Vector                                            *
 *-----------------------------------------------------------*
 * The type N_Vector is an abstract vector type. The fields of *
 * its concrete representation should not be accessed        *
 * directly, but rather through the macros given below.      *
 *                                                           *
 * A user may assume that the N components of an N_Vector    *
 * are stored contiguously. A pointer to the first component  *
 * can be obtained via the macro N_VDATA.                    *
 *                                                           *
 *************************************************************/

typedef struct {
  integer length;
  real   *data;
} *N_Vector;
```

10.3.1.2 N_Vector Accessor Macros

```
/*************************************************************
 *                                                           *
 * Macros: N_VMAKE, N_VDISPOSE, N_VDATA, N_VLENGTH, N_VIth   *
 *-----------------------------------------------------------*
 * In the descriptions below, the following user             *
 * declarations are assumed:                                 *
 *                                                           *
 * N_Vector v; real *v_data, r; integer v_len, i;            *
 *                                                           *
 * (1) N_VMAKE, N_VDISPOSE                                   *
 *                                                           *
 *     These companion routines are used to create and       *
 *     destroy an N_Vector with a component array v_data     *
 *     allocated by the user.                                *
 *                                                           *
 *     The call N_VMAKE(v, v_data, v_len) makes v an         *
 *     N_Vector with component array v_data and length v_len. *
 *     N_VMAKE 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 N_VDISPOSE(v) frees all memory associated    *
 *     with v except the its component array. This memory was *
 *     allocated by the user and, therefore, should be       *
 *     deallocated by the user.                              *
 *                                                           *
 * (2) N_VDATA, N_VLENGTH                                    *
 *                                                           *
 *     These routines give individual access to the parts of *
 *     an N_Vector.                                          *
 *                                                           *
 *     The assignment v_data=N_VDATA(v) sets v_data to be    *
 *     a pointer to the first component of v. The assignment *
 *     N_VDATA(v)=v_data sets the component array of v to    *
 *     be v_data by storing the pointer v_data.              *
```

```
 *                                                            *
 *     The assignment v_len=N_VLENGTH(v) sets v_len to be     *
 *     the length of v. The call N_VLENGTH(v)=len_v sets      *
 *     the length of v to be len_v.                           *
 *                                                            *
 * (3) N_VIth                                                 *
 *                                                            *
 *     In the following description, the components of an    *
 *     N_Vector are numbered 0..N-1, where N is the length of *
 *     v.                                                     *
 *                                                            *
 *     The assignment r=N_VIth(v,i) sets r to be the value of *
 *     the ith component of v. The assignment N_VIth(v,i)=r   *
 *     sets the value of the ith component of v to be r.      *
 *                                                            *
 * Notes..                                                    *
 *                                                            *
 * Users who use the macros (1) 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=N_VDATA(v) and then access v_data[i] within the     *
 * loop than it is to use N_VDATA(v,i) within the loop.       *
 *                                                            *
 * N_VMAKE and N_VDISPOSE are similar to N_VNew and N_VFree.  *
 * 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 N_VMAKE and N_VDISPOSE, the *
 * component memory is allocated and freed by the user of     *
 * this package.                                              *
 *                                                            *
 **************************************************************/

#define N_VMAKE(v, v_data, v_len) v = (N_Vector) malloc(sizeof(*v)); \
                                  v->data   = v_data; \
                                  v->length = v_len

#define N_VDISPOSE(v) free(v)

#define N_VDATA(v) (v->data)

#define N_VLENGTH(v) (v->length)

#define N_VIth(v,i) ((v->data)[i])
```

10.3.1.3 N_Vector Kernels

10.3.1.3.1 Memory Allocation and Deallocation

```
/**************************************************************
 *                                                            *
 * Memory Allocation and Deallocation: N_VNew, N_VFree        *
 *                                                            *
 **************************************************************/
```

```
/**************************************************************
 *                                                            *
 * Function : N_VNew                                          *
 * Usage    : x = N_VNew(N, machEnv);                         *
 *------------------------------------------------------------*
 *                                                            *
 * Returns a new N_Vector of length N. The parameter machEnv  *
 * is a pointer to machine environment-specific information.  *
 * It is ignored in the sequential machine environment and the *
 * user in this environment should simply pass NULL for this  *
 * argument. If there is not enough memory for a new N_Vector, *
 * then N_VNew returns NULL.                                  *
 *                                                            *
 **************************************************************/

N_Vector N_VNew(integer n, void *machEnv);


/**************************************************************
 *                                                            *
 * Function : N_VFree                                         *
 * Usage    : N_VFree(x);                                     *
 *------------------------------------------------------------*
 *                                                            *
 * Frees the N_Vector x. It is illegal to use x after the call *
 * N_VFree(x).                                                *
 *                                                            *
 **************************************************************/

void N_VFree(N_Vector x);
```

10.3.1.3.2 Arithmetic

```
/**************************************************************
 *                                                            *
 * N_Vector Arithmetic: N_VLinearSum, N_VConst, N_VProd,      *
 *                      N_VDiv, N_VScale, N_VAbs, N_VInv,     *
 *                      N_VAddConst                           *
 *                                                            *
 **************************************************************/


/**************************************************************
 *                                                            *
 * Function  : N_VLinearSum                                   *
 * Operation : z = a x + b y                                  *
 *                                                            *
 **************************************************************/

void N_VLinearSum(real a, N_Vector x, real b, N_Vector y, N_Vector z);


/**************************************************************
 *                                                            *
 * Function  : N_VConst                                       *
 * Operation : z[i] = c for i=0, 1, ..., N-1                  *
 *                                                            *
 **************************************************************/
```

```
void N_VConst(real c, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VProd                                         *
 * Operation : z[i] = x[i] * y[i] for i=0, 1, ..., N-1         *
 *                                                             *
 ***************************************************************/

void N_VProd(N_Vector x, N_Vector y, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VDiv                                          *
 * Operation : z[i] = x[i] / y[i] for i=0, 1, ..., N-1         *
 *                                                             *
 ***************************************************************/

void N_VDiv(N_Vector x, N_Vector y, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VScale                                        *
 * Operation : z = c x                                         *
 *                                                             *
 ***************************************************************/

void N_VScale(real c, N_Vector x, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VAbs                                          *
 * Operation : z[i] = |x[i]|,   for i=0, 1, ..., N-1           *
 *                                                             *
 ***************************************************************/

void N_VAbs(N_Vector x, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VInv                                          *
 * Operation : z[i] = 1.0 / x[i] for i = 0, 1, ..., N-1        *
 *-------------------------------------------------------------*
 *                                                             *
 * This routine does not check for division by 0. It should be *
 * called only with an N_Vector x which is guaranteed to have  *
 * all non-zero components.                                    *
 *                                                             *
 ***************************************************************/

void N_VInv(N_Vector x, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VAddConst                                     *
```

```
 * Operation : z[i] = x[i] + b   for i = 0, 1, ..., N-1        *
 *                                                             *
 ***************************************************************/

void N_VAddConst(N_Vector x, real b, N_Vector z);
```

10.3.1.3.3 Measures

```
/***************************************************************
 *                                                             *
 * N_Vector Measures: N_VDotProd, N_VMaxNorm, VWrmsNorm,       *
 *                    N_VMin                                   *
 *                                                             *
 ***************************************************************/


/***************************************************************
 *                                                             *
 * Function : N_VDotProd                                       *
 * Usage    : dotprod = N_VDotProd(x, y);                      *
 *-------------------------------------------------------------*
 *                                                             *
 * Returns the value of the ordinary dot product of x and y:   *
 *                                                             *
 * -> sum (i=0 to N-1) {x[i] * y[i]}                           *
 *                                                             *
 * Returns 0.0 if N <= 0.                                      *
 *                                                             *
 ***************************************************************/

real N_VDotProd(N_Vector x, N_Vector y);


/***************************************************************
 *                                                             *
 * Function : N_VMaxNorm                                       *
 * Usage    : maxnorm = N_VMaxNorm(x);                         *
 *-------------------------------------------------------------*
 *                                                             *
 * Returns the maximum norm of x:                              *
 *                                                             *
 * -> max (i=0 to N-1) |x[i]|                                  *
 *                                                             *
 * Returns 0.0 if N <= 0.                                      *
 *                                                             *
 ***************************************************************/

real N_VMaxNorm(N_Vector x);


/***************************************************************
 *                                                             *
 * Function : N_VWrmsNorm                                      *
 * Usage    : wrmsnorm = N_VWrmsNorm(x, w);                    *
 *-------------------------------------------------------------*
 *                                                             *
 * Returns the weighted root mean square norm of x with        *
 * weight vector w:                                            *
 *                                                             *
 * -> sqrt [(sum (i=0 to N-1) {(x[i] * w[i])^2}) / N]          *
```

```
 *                                                             *
 * Returns 0.0 if N <= 0.                                      *
 *                                                             *
 ***************************************************************/

real N_VWrmsNorm(N_Vector x, N_Vector w);


/***************************************************************
 *                                                             *
 * Function : N_VMin                                           *
 * Usage    : min = N_VMin(x);                                 *
 *-------------------------------------------------------------*
 *                                                             *
 * Returns min x[i] if N > 0 and returns 0.0 if N <= 0.        *
 *          i                                                  *
 *                                                             *
 ***************************************************************/

real N_VMin(N_Vector x);
```

10.3.1.3.4 Miscellaneous

```
/***************************************************************
 *                                                             *
 * Miscellaneous : N_VCompare, N_VInvTest                      *
 *                                                             *
 ***************************************************************/


/***************************************************************
 *                                                             *
 * Function  : N_VCompare                                      *
 * Operation : z[i] = 1.0 if |x[i]| >= c   i = 0, 1, ..., N-1  *
 *                    0.0 otherwise                            *
 *                                                             *
 ***************************************************************/

void N_VCompare(real c, N_Vector x, N_Vector z);


/***************************************************************
 *                                                             *
 * Function  : N_VInvTest                                      *
 * Operation : z[i] = 1.0 / x[i] with a test for x[i]==0.0     *
 *             before inverting x[i].                          *
 *-------------------------------------------------------------*
 *                                                             *
 * This routine returns TRUE if all components of x are        *
 * non-zero (successful inversion) and returns FALSE           *
 * otherwise.                                                  *
 *                                                             *
 ***************************************************************/

bool N_VInvTest(N_Vector x, N_Vector z);
```

10.3.1.3.5 Debugging Tools

```
/**************************************************************
 *                                                            *
 * Debugging Tools : N_VPrint                                 *
 *                                                            *
 **************************************************************/

/**************************************************************
 *                                                            *
 * Function : N_VPrint                                        *
 * Usage    : N_VPrint(x);                                    *
 *------------------------------------------------------------*
 *                                                            *
 * Prints the N_Vector x to stdout. Each component of x is    *
 * printed on a separate line using the %g specification. This *
 * routine is provided as an aid in debugging code which uses  *
 * this vector package.                                       *
 *                                                            *
 **************************************************************/

void N_VPrint(N_Vector x);
```

10.3.2 DENSE

The following sections contain documentation and declarations from the header file `dense.h`.

```
/******************************************************************
 *                                                                *
 * File          : dense.h                                        *
 * Programmers   : Scott D. Cohen and Alan C. Hindmarsh @ LLNL    *
 * Last Modified : 1 September 1994                               *
 *----------------------------------------------------------------*
 * This is the header file for a generic DENSE linear solver      *
 * package. There are two sets of dense solver routines listed in *
 * this file: one set uses type DenseMat defined below and the    *
 * other set uses the type real ** for dense matrix arguments.    *
 * The two sets of dense solver routines make it easy to work     *
 * with two types of dense matrices:                              *
 *                                                                *
 * (1) The DenseMat type is intended for use with large dense     *
 *     matrices whose elements/columns may be stored in           *
 *     non-contiguous memory locations or even distributed into   *
 *     different processor memories. This type may be modified to *
 *     include such distribution information. If this is done,    *
 *     then all the routines that use DenseMat must be modified   *
 *     to reflect the new data structure.                         *
 *                                                                *
 * (2) The set of routines that use real ** (and NOT the DenseMat *
 *     type) is intended for use with small matrices which can    *
 *     easily be allocated within a contiguous block of memory    *
 *     on a single processor.                                     *
 *                                                                *
 * Routines that work with the type DenseMat begin with "Dense".  *
 * The DenseAllocMat function allocates a dense matrix for use in *
 * the other DenseMat routines listed in this file. Matrix        *
 * storage details are given in the documentation for the type    *
 * DenseMat. The DenseAllocPiv function allocates memory for      *
 * pivot information. The storage allocated by DenseAllocMat and  *
 * DenseAllocPiv is deallocated by the routines DenseFreeMat and  *
```

```
 * DenseFreePiv, respectively. The DenseFactor and DenseBacksolve *
 * routines perform the actual solution of a dense linear system. *
 * Note that the DenseBacksolve routine has a parameter b of type *
 * N_Vector. The current implementation makes use of a machine    *
 * environment specific macro (N_VDATA) which may not exist for   *
 * other implementations of the type N_Vector. Thus, the          *
 * implementation of DenseBacksolve may need to change if the     *
 * type N_Vector is changed.                                      *
 *                                                                *
 * Routines that work with real ** begin with "den" (except for   *
 * the factor and solve routines which are called gefa and gesl,  *
 * respectively). The underlying matrix storage is described in   *
 * the documentation for denalloc.                                *
 *                                                                *
 ******************************************************************/
```

10.3.2.1 Type DenseMat

```
/******************************************************************
 *                                                                *
 * Type: DenseMat                                                 *
 *----------------------------------------------------------------*
 * The type DenseMat is defined to be a pointer to a structure    *
 * with a size and a data field. The size field indicates the     *
 * number of columns (== 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 <= i,j <= size-1) is given by the expression (A->data)[j][i] *
 * or by the expression (A->data)[0][j*n+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 jth   *
 * column of elements can be obtained via the DENSE_COL macro.    *
 * Users should use these macros whenever possible.               *
 *                                                                *
 ******************************************************************/

typedef struct {
  integer size;
  real  **data;
} *DenseMat;
```

10.3.2.2 DenseMat Accessor Macros

```
/******************************************************************
 *                                                                *
 * Macro : DENSE_ELEM                                             *
 * Usage : DENSE_ELEM(A,i,j) = a_ij;  OR                          *
 *         a_ij = DENSE_ELEM(A,i,j);                              *
 *----------------------------------------------------------------*
 * DENSE_ELEM(A,i,j) references the (i,j)th element of the N by N *
 * DenseMat A, 0 <= i,j <= N-1.                                   *
 *                                                                *
```

```
 ****************************************************************/

#define DENSE_ELEM(A,i,j) ((A->data)[j][i])


/****************************************************************
 *                                                              *
 * Macro : DENSE_COL                                            *
 * Usage : col_j = DENSE_COL(A,j);                              *
 *--------------------------------------------------------------*
 * DENSE_COL(A,j) references the jth column of the N by N       *
 * DenseMat A, 0 <= j <= N-1. The type of the expression        *
 * DENSE_COL(A,j) is real *. 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].          *
 *                                                              *
 ****************************************************************/

#define DENSE_COL(A,j) ((A->data)[j])
```

10.3.2.3 DenseMat Functions

The following functions for `DenseMat` matrices are available in the DENSE package. For full details, see the 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

10.3.2.4 Small Dense Matrix Functions

```
/******************************************************************
 *                                                                *
 * Function : denalloc                                            *
 * Usage    : real **a;                                           *
 *            a = denalloc(n);                                    *
 *            if (a == NULL) ... memory request failed            *
 *----------------------------------------------------------------*
 * 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 real **. If we allocate a dense matrix real **a by *
 * a = denalloc(n), then a[j][i] references the (i,j)th element   *
 * of the matrix a, 0 <= i,j <= n-1, and a[j] is a pointer to the *
 * first element in the jth column of a. The location a[0]        *
 * contains a pointer to n^2 contiguous locations which contain   *
 * the elements of a.                                             *
 *                                                                *
 ******************************************************************/

real **denalloc(integer n);


/******************************************************************
 *                                                                *
 * Function : denallocpiv                                         *
 * Usage    : integer *pivot;                                     *
 *            pivot = denallocpiv(n);                             *
 *            if (pivot == NULL) ... memory request failed        *
 *----------------------------------------------------------------*
 * 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.     *
 *                                                                *
 ******************************************************************/

integer *denallocpiv(integer n);


/******************************************************************
 *                                                                *
 * Function : gefa                                                *
 * Usage    : integer ier;                                        *
 *            ier = gefa(a,n,p);                                  *
 *            if (ier > 0) ... zero element encountered during    *
 *                             the factorization                  *
 *----------------------------------------------------------------*
 * 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, ..., 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.                                          *
 *                                                                *
 ******************************************************************/

integer gefa(real **a, integer n, integer *p);


/******************************************************************
 *                                                                *
 * Function : gesl                                                *
 * Usage    : real *b;                                            *
 *            ier = gefa(a,n,p);                                  *
 *            if (ier == 0) gesl(a,n,p,b);                        *
 *----------------------------------------------------------------*
 * 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.                                   *
 *                                                                *
 ******************************************************************/

void gesl(real **a, integer n, integer *p, real *b);


/******************************************************************
 *                                                                *
 * Function : denzero                                             *
 * Usage    : denzero(a,n);                                       *
 *----------------------------------------------------------------*
 * denzero(a,n) sets all the elements of the n by n dense matrix  *
 * a to be 0.0.                                                   *
 *                                                                *
 ******************************************************************/

void denzero(real **a, integer n);


/******************************************************************
 *                                                                *
 * Function : dencopy                                             *
 * Usage    : dencopy(a,b,n);                                     *
 *----------------------------------------------------------------*
 * dencopy(a,b,n) copies the n by n dense matrix a into the       *
 * n by n dense matrix b.                                         *
 *                                                                *
 ******************************************************************/

void dencopy(real **a, real **b, integer n);


/******************************************************************
 *                                                                *
 * Function : denscale                                            *
 * Usage    : denscale(c,a,n);                                    *
 *----------------------------------------------------------------*
```

```
 * denscale(c,a,n) scales every element in the n by n dense       *
 * matrix a by c.                                                 *
 *                                                                *
 ******************************************************************/

void denscale(real c, real **a, integer n);


/******************************************************************
 *                                                                *
 * Function : denaddI                                             *
 * Usage    : denaddI(a,n);                                       *
 *----------------------------------------------------------------*
 * denaddI(a,n) increments the n by n dense matrix a by the       *
 * identity matrix.                                               *
 *                                                                *
 ******************************************************************/

void denaddI(real **a, integer n);


/******************************************************************
 *                                                                *
 * Function : denfreepiv                                          *
 * Usage    : denfreepiv(p);                                      *
 *----------------------------------------------------------------*
 * denfreepiv(p) frees the pivot array p allocated by             *
 * denallocpiv.                                                   *
 *                                                                *
 ******************************************************************/

void denfreepiv(integer *p);


/******************************************************************
 *                                                                *
 * Function : denfree                                             *
 * Usage    : denfree(a);                                         *
 *----------------------------------------------------------------*
 * denfree(a) frees the dense matrix a allocated by denalloc.     *
 *                                                                *
 ******************************************************************/

void denfree(real **a);


/******************************************************************
 *                                                                *
 * Function : denprint                                            *
 * Usage    : denprint(a,n);                                      *
 *----------------------------------------------------------------*
 * 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.                                          *
 *                                                                *
 ******************************************************************/

void denprint(real **a, integer n);
```

10.3.3 BAND

The following sections contain documentation and declarations from the header file `band.h`.

```
/******************************************************************
 *                                                                *
 * File          : band.h                                         *
 * Programmers   : Scott D. Cohen and Alan C. Hindmarsh @ LLNL    *
 * Last Modified : 1 September 1994                               *
 *----------------------------------------------------------------*
 * This is the header file for a generic BAND linear solver       *
 * package. There are two sets of band solver routines listed in  *
 * this file: one set uses type BandMat defined below and the     *
 * other set uses the type real ** for band matrix arguments.     *
 * The two sets of band solver routines make it easy to work      *
 * with two types of band matrices:                               *
 *                                                                *
 * (1) The BandMat type is intended for use with large            *
 *     band matrices whose elements/columns may be stored in      *
 *     non-contiguous memory locations or even distributed into   *
 *     different processor memories. This type may be modified to *
 *     include such distribution information. If this is done,    *
 *     then all the routines that use BandMat must be modified to *
 *     reflect the new data structure.                            *
 *                                                                *
 * (2) The set of routines that use real ** (and NOT the BandMat  *
 *     type) is intended for use with small matrices which can    *
 *     easily be allocated within a contiguous block of memory    *
 *     on a single processor.                                     *
 *                                                                *
 * Routines that work with the type BandMat begin with "Band".    *
 * The BandAllocMat function allocates a band matrix for use in   *
 * the other matrix routines listed in this file. Matrix storage  *
 * details are given in the documentation for the type BandMat.   *
 * The BandAllocPiv function allocates memory for pivot           *
 * information. The storage allocated by BandAllocMat and         *
 * BandAllocPiv is deallocated by the routines BandFreeMat and    *
 * BandFreePiv, respectively. The BandFactor and BandBacksolve    *
 * routines perform the actual solution of a band linear system.  *
 * Note that the BandBacksolve routine has a parameter b of type  *
 * N_Vector. The current implementation makes use of a machine    *
 * environment specific macro (N_VDATA) which may not exist for   *
 * other implementations of the type N_Vector. Thus, the          *
 * implementation of BandBacksolve may need to change if the      *
 * type N_Vector is changed.                                      *
 *                                                                *
 * Routines that work with real ** begin with "band" (except for  *
 * the factor and solve routines which are called gbfa and gbsl,  *
 * respectively). The underlying matrix storage is described in   *
 * the documentation for bandalloc.                               *
 *                                                                *
 ******************************************************************/
```

10.3.3.1 Type BandMat

See Figure 3 following the BandMat documentation for a diagram of the BandMat type.

```
/******************************************************************
 *                                                                *
 * Type: BandMat                                                  *
 *----------------------------------------------------------------*
 * The type BandMat is the type of a large (possibly distributed) *
 * band matrix. It is defined to be a pointer to a structure      *
 * with the following fields:                                     *
 *                                                                *
 * size is the number of columns (== number of rows)              *
 *                                                                *
 * mu   is the upper bandwidth, 0 <= mu <= size-1                 *
 *                                                                *
 * ml   is the lower bandwidth, 0 <= ml <= size-1                 *
 *                                                                *
 * smu  is the storage upper bandwidth, mu <= smu <= size-1.      *
 *         The BandFactor routine writes the LU factors           *
 *         into the storage for A. The upper triangular factor U, *
 *         however, may have an upper bandwidth as big as         *
 *         MIN(size-1,mu+ml) because of partial pivoting. The smu *
 *         field holds the upper 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 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 jth column. This pointer may be treated as   *
 *            an array indexed from smu-mu (to access the         *
 *            uppermost element within the band in the jth        *
 *            column) to smu+ml (to access the lowest element     *
 *            within the band in the jth 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 <= i <= 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 jth *
 * column of elements can be obtained via the BAND_COL macro. The *
 * BAND_COL_ELEM macro selects an element from a column which has *
 * already been isolated via BAND_COL. BAND_COL_ELEM allows the   *
 * user to avoid the translation from the matrix location (i,j)   *
 * to the index in the array returned by BAND_COL at which the    *
 * (i,j)th element is stored. See the documentation for BAND_COL  *
 * and BAND_COL_ELEM for usage details. Users should use these    *
 * macros whenever possible.                                      *
 *                                                                *
```

```
 ****************************************************************/

typedef struct {
  integer size;
  integer mu, ml, smu;
  real **data;
} *BandMat;
```

10.3.3.2 BandMat Accessor Macros

```
/****************************************************************
 *                                                              *
 * Macro : BAND_ELEM                                            *
 * Usage : BAND_ELEM(A,i,j) = a_ij;  OR                         *
 *         a_ij = BAND_ELEM(A,i,j);                             *
 *--------------------------------------------------------------*
 * BAND_ELEM(A,i,j) references the (i,j)th element of the       *
 * N by N band matrix A, where 0 <= i,j <= N-1. The location    *
 * (i,j) should further satisfy j-(A->mu) <= i <= j+(A->ml).    *
 *                                                              *
 ****************************************************************/

#define BAND_ELEM(A,i,j) ((A->data)[j][i-j+(A->smu)])


/****************************************************************
 *                                                              *
 * Macro : BAND_COL                                             *
 * Usage : col_j = BAND_COL(A,j);                               *
 *--------------------------------------------------------------*
 * BAND_COL(A,j) references the diagonal element of the jth     *
 * column of the N by N band matrix A, 0 <= j <= N-1. The type of *
 * the expression BAND_COL(A,j) is real *. The pointer returned  *
 * by the call BAND_COL(A,j) can be treated as an array which is *
 * indexed from -(A->mu) to (A->ml).                            *
 *                                                              *
 ****************************************************************/

#define BAND_COL(A,j) (((A->data)[j])+(A->smu))


/****************************************************************
 *                                                              *
 * Macro : BAND_COL_ELEM                                        *
 * Usage : col_j = BAND_COL(A,j);                               *
 *         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 as shown above. The    *
 * index (i,j) should satisfy j-(A->mu) <= i <= j+(A->ml).      *
 *                                                              *
 ****************************************************************/

#define BAND_COL_ELEM(col_j,i,j) (col_j[i-j])
```

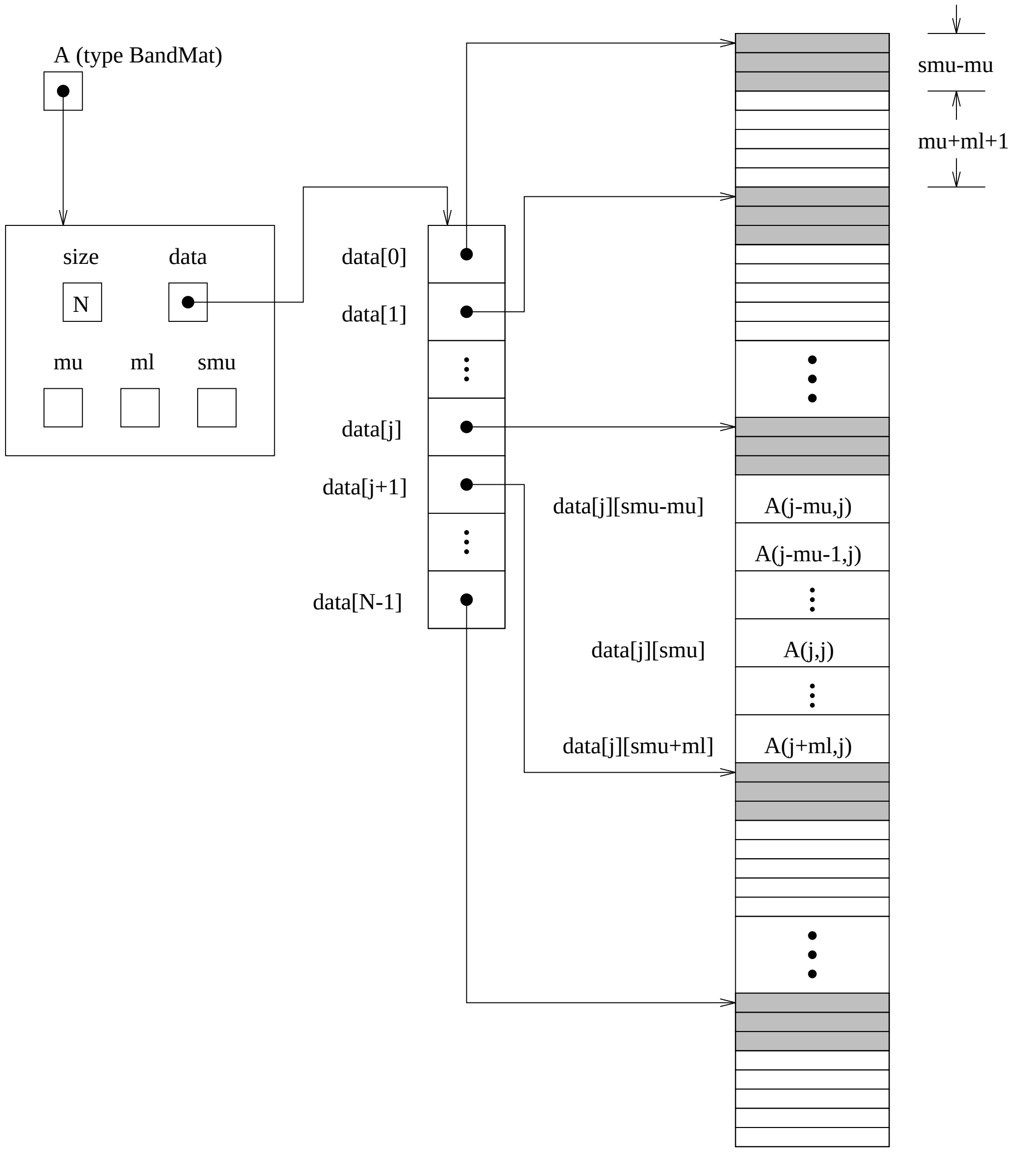

Figure 3: Diagram of the storage for a band matrix of type BandMat. Here A is an N $\times$ N band matrix of type BandMat with upper and lower bandwidths mu and ml, respectively. The rows and columns of A are numbered from 0 to N-1 and the (i,j)th element of A is denoted A(i,j). The greyed out areas of the underlying component storage are used by the BandFactor and BandBacksolve routines.

10.3.3.3 BandMat Functions

The following functions for BandMat matrices are available in the BAND package. For full details, see the 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

10.3.3.4 Small Band Matrix Functions

```
/******************************************************************
 *                                                                *
 * Function : bandalloc                                           *
 * Usage    : real **a;                                           *
 *            a = bandalloc(n, smu, ml);                          *
 *            if (a == NULL) ... memory request failed            *
 *----------------------------------------------------------------*
 * bandalloc(n, smu, ml) allocates storage for an n by n band     *
 * matrix A with storage upper bandwidth smu and lower bandwidth  *
 * ml. It returns a pointer to the newly allocated storage if     *
 * successful. If the memory request cannot be satisfied, then    *
 * bandalloc returns NULL. If, mathematically, A has upper and    *
 * lower bandwidths mu and ml, respectively, then the value       *
 * passed to bandalloc for smu may need to be greater than mu.    *
 * The gbfa routine writes the LU factors into the storage (named *
 * "a" in the above usage documentation) for A (thus destroying   *
 * the original elements of A). The upper triangular factor U,    *
 * however, may have a larger upper bandwidth than the upper      *
 * bandwidth mu of A. Thus some "extra" storage for A must be     *
 * allocated if A is to be factored by gbfa. Pass smu as follows: *
 *                                                                *
 * (1) Pass smu = mu if A will not be factored.                   *
 *                                                                *
 * (2) Pass smu = MIN(n-1,mu+ml) if A will be factored.           *
 *                                                                *
 * The underlying type of the band matrix returned is real **. If *
```

```
 * we allocate a band matrix A in real **a by                     *
 * a = bandalloc(n,smu,ml), then a[0] is a pointer to             *
 * n * (smu + ml + 1) contiguous storage locations and a[j] is a  *
 * pointer to the uppermost element in the storage for the jth    *
 * column. The expression a[j][i-j+smu] references the (i,j)th    *
 * element of A, where 0 <= i,j <= n-1 and j-mu <= i <= j+ml.     *
 * (The elements a[j][0], a[j][1], ..., a[j][smu-mu-1] are used   *
 * by gbfa and gbsl.)                                             *
 *                                                                *
 ******************************************************************/

real **bandalloc(integer n, integer smu, integer ml);


/******************************************************************
 *                                                                *
 * Function : bandallocpiv                                        *
 * Usage    : integer *pivot;                                     *
 *            pivot = bandallocpiv(n);                            *
 *            if (pivot == NULL) ... memory request failed        *
 *----------------------------------------------------------------*
 * bandallocpiv(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.     *
 *                                                                *
 ******************************************************************/

integer *bandallocpiv(integer n);


/******************************************************************
 *                                                                *
 * Function : gbfa                                                *
 * Usage    : integer ier;                                        *
 *            ier = gbfa(a,n,mu,ml,smu,p);                        *
 *            if (ier > 0) ... zero element encountered during    *
 *                             the factorization                  *
 *----------------------------------------------------------------*
 * gbfa(a,n,mu,ml,smu,p) factors the n by n band matrix A (upper  *
 * and lower bandwidths mu and ml, storage upper bandwidth smu)   *
 * stored in "a". It overwrites the elements of A with the LU     *
 * factors and it keeps track of the pivot rows chosen in the     *
 * pivot array p.                                                 *
 *                                                                *
 * A successful LU factorization leaves a and 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, ..., 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.        *
 *                                                                *
 * gbfa 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.                                          *
 *                                                                *
```

```
 * IMPORTANT NOTE: Suppose A is a band matrix with upper          *
 * bandwidth mu and lower bandwidth ml, then the upper triangular *
 * factor U can have upper bandwidth as big as MIN(n-1,mu+ml)     *
 * because of partial pivoting. The lower triangular factor L has *
 * lower bandwidth ml. Thus, if A is to be factored and           *
 * backsolved using gbfa and gbsl, then it should be allocated    *
 * as a = bandalloc(n,smu,ml), where smu = MIN(n-1,mu+ml). The    *
 * call to gbfa is ier = gbfa(a,n,mu,ml,smu,p). The corresponding *
 * call to gbsl is gbsl(a,n,smu,ml,p,b). The user does not need   *
 * to zero the "extra" storage allocated for the purpose of       *
 * factorization. This is handled by the gbfa routine. If A is    *
 * not going to be factored and backsolved, then it can be        *
 * allocated as a = bandalloc(n,smu,ml). In either case, all      *
 * routines in this section use the parameter name smu for a      *
 * parameter which must be the "storage upper bandwidth" which    *
 * was passed to bandalloc.                                       *
 *                                                                *
 ******************************************************************/

integer gbfa(real **a, integer n, integer mu, integer ml, integer smu,
             integer *p);


/******************************************************************
 *                                                                *
 * Function : gbsl                                                *
 * Usage    : real *b;                                            *
 *            ier = gbfa(a,n,mu,ml,smu,p);                        *
 *            if (ier == 0) gbsl(a,n,smu,ml,p,b);                 *
 *----------------------------------------------------------------*
 * gbsl(a,n,smu,ml,p,b) solves the n by n linear system           *
 * Ax = b, where A is band matrix stored in "a" with storage      *
 * upper bandwidth smu and lower bandwidth ml. It assumes that A  *
 * has been LU factored and the pivot array p has been set by a   *
 * successful call gbfa(a,n,mu,ml,smu,p). The solution x is       *
 * written into the b array.                                      *
 *                                                                *
 ******************************************************************/

void gbsl(real **a, integer n, integer smu, integer ml, integer *p, real *b);


/******************************************************************
 *                                                                *
 * Function : bandzero                                            *
 * Usage    : bandzero(a,n,mu,ml,smu);                            *
 *----------------------------------------------------------------*
 * a(i,j) <- 0.0,   0 <= i,j <= n-1, j-mu <= i <= j+ml.           *
 *                                                                *
 ******************************************************************/

void bandzero(real **a, integer n, integer mu, integer ml, integer smu);


/******************************************************************
 *                                                                *
 * Function : bandcopy                                            *
 * Usage    : bandcopy(a,b,n,a_smu,b_smu,copymu,copyml);          *
 *----------------------------------------------------------------*
 * b(i,j) <- a(i,j), 0 <= i,j <= n-1, j-copymu <= i <= j+copyml.  *
 *                                                                *
 ******************************************************************/
```

```
void bandcopy(real **a, real **b, integer n, integer a_smu, integer b_smu,
              integer copymu, integer copyml);


/******************************************************************
 *                                                                *
 * Function : bandscale                                           *
 * Usage    : bandscale(c,a,n,mu,ml);                             *
 *----------------------------------------------------------------*
 * a(i,j) <- c*a(i,j),   0 <= i,j <= n-1, j-mu <= i <= j+ml.      *
 *                                                                *
 ******************************************************************/

void bandscale(real c, real **a, integer n, integer mu, integer ml,
               integer smu);


/******************************************************************
 *                                                                *
 * Function : bandaddI                                            *
 * Usage    : bandaddI(a,n,smu);                                  *
 *----------------------------------------------------------------*
 * a(j,j) <- a(j,j)+1.0,   0 <= j <= n-1.                         *
 *                                                                *
 ******************************************************************/

void bandaddI(real **a, integer n, integer smu);


/******************************************************************
 *                                                                *
 * Function : bandfreepiv                                         *
 * Usage    : bandfreepiv(p);                                     *
 *----------------------------------------------------------------*
 * bandfreepiv(p) frees the pivot array p allocated by            *
 * bandallocpiv.                                                  *
 *                                                                *
 ******************************************************************/

void bandfreepiv(integer *p);


/******************************************************************
 *                                                                *
 * Function : bandfree                                            *
 * Usage    : bandfree(a);                                        *
 *----------------------------------------------------------------*
 * bandfree(a) frees the band matrix a allocated by bandalloc.    *
 *                                                                *
 ******************************************************************/

void bandfree(real **a);


/******************************************************************
 *                                                                *
 * Function : bandprint                                           *
 * Usage    : bandprint(a,n,mu,ml,smu);                           *
 *----------------------------------------------------------------*
 * bandprint(a,n,mu,ml,smu) prints the n by n band matrix stored  *
 * in a (with upper bandwidth mu and lower bandwidth ml) 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.                           *
 *                                                                *
 ******************************************************************/

void bandprint(real **a, integer n, integer mu, integer ml, integer smu);
```

10.3.4 SPGMR

The SPGMR package, in the files `spgmr.h`, `spgmr.c`, includes an implementation of the scaled precondition 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`.

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`, `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] George 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] Krishnan Radhakrishnan and Alan C. Hindmarsh, *Discription and Use of LSODE, the Livermore Solver for Ordinary Differential Equations*, NASA Reference Publication 1327, 1993, and LLNL Report UCRL-ID-113855, March 1994.

[5] 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

The authors have tried to provide as useful an index as possible. Page numbers in boldface are in Section 10, the Reference Guide. These entries point the reader toward the most complete source of information in this user guide for a given item. If an item with a single bold page number citation does not appear on the specified page, please check the previous page.