CVODE, A STIFF/NONSTIFF ODE SOLVER IN C[*]

SCOTT D. COHEN[†] AND ALAN C. HINDMARSH[‡]

Abstract. CVODE is a package written in C for solving initial value problems for ordinary differential equations. It provides the capabilities of two older Fortran packages, VODE and VODPK. CVODE solves both stiff and nonstiff systems, using variable-coefficient Adams and BDF methods. In the stiff case, options for treating the Jacobian of the system include dense and band matrix solvers, and a preconditioned Krylov (iterative) solver. In the highly modular organization of CVODE, the core integrator module is independent of the linear system solvers, and all operations on N-vectors are isolated in a module of vector kernels. A set of parallel extenstions of CVODE, called PVODE, is being developed. CVODE is available from Netlib, and comes with an extensive user guide.

1. Introduction. CVODE is a general-purpose solver for the initial value problem (IVP) for ordinary differential equation (ODE) systems. We write such an IVP abstractly as

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

where $\dot{y}$ denotes the derivative dy/dt. CVODE is written in ANSI standard C, and is based on two older packages written in Fortran, VODE and VODPK.

The basic methods represented in CVODE are summarized as follows. Two linear multistep methods are available, namely

- **Adams** (Adams-Moulton) methods in variable-order, variable-step form, and
- **BDF** (Backward Differentiation Formula) methods, in variable-order, variable-step, fixed-leading-coefficient form.

Both linear multistep methods can be written in the form

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

Here the y_n are computed approximations to $y(t_n)$, $h_n = t_n - t_{n-1}$ is the step size, 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. The choices available in CVODE for the solution of the system (3) are

- **Functional** iteration, involving no linear systems, and
- **Newton** iteration, where linear systems must be solved.

[*] This research was supported by Lawrence Livermore National Laboratory under contract W-7405-ENG-48. This paper was presented at SciCADE95: Scientific Computing and Differential Equations, Stanford University, March 1995 – a conference held in honor of the 60th birthday of C. W. Gear.

[†] Stanford University, and Center for Computational Sciences & Engineering, L-316, Lawrence Livermore National Laboratory, Livermore, California 94551.

[‡] Center for Computational Sciences & Engineering, L-316, Lawrence Livermore National Laboratory, Livermore, California 94551.

The Newton iteration, which in some cases is a Modified Newton iteration, involves the solution of the equation

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

in which

$$M \approx I - \gamma J, \quad J = \partial f / \partial y \quad \text{and} \quad \gamma = h_n \beta_{n,0}. \tag{5}$$

For this the choices available in CVODE are:

- a direct method with dense treatment of J
- a direct method with banded treatment of J
- a direct method with approximate diagonal treatment of J
- a Krylov method, preconditioned GMRES (Generalized Minimal RESidual method) [5].

With scaling and preconditioning included in the GMRES method, we refer to this algorithm as SPGMR: Scaled Preconditioned GMRES. The combination of a BDF integrator and a preconditioned GMRES algorithm yields 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].

Most of the algorithmic features of CVODE have their origins in the work of Bill Gear. These include

- BDF and Adams methods in a single solver
- Nordsieck history array for BDF methods
- Modified Newton iteration, with Jacobian fixed over several time steps
- User options for difference-quotient or analytic Jacobian
- BDF local error estimates via predictor-corrector differences
- BDF step size and order selection via asymptotic local error behavior.

The two Fortran solvers that together represent these methods 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 only direct methods (full or banded) for the solution of linear systems. VODPK is a variant of VODE that uses a preconditioned GMRES method for the solution of linear systems.

2. CVODE Organization. The VODE and VODPK solvers have certain drawbacks, some of which are inherent in the Fortran language, that have motivated the writing of CVODE. First, there are two distinct solvers, one containing the direct linear system methods (VODE), and one containing the Krylov methods (VODPK). In addition, there are two versions of each solver, one in single precision, and one in double. The differences among the four solvers are limited primarily to declarations and the linear solvers, with a great deal of overlap otherwise. The main driver subroutine in each solver contains logic that is specific to the linear solvers as well as logic for the time integration. In VODPK, the implementation of the preconditioned GMRES method involves a mix of logic for the generic GMRES method and logic specific to the context of Newton iteration in a time integration.

In contrast, the design of CVODE was planned to avoid these drawbacks. The modules of the CVODE package are shown in Figure 1. The core integrator module, also referred to as CVode, deals strictly with the time integration issues, and is completely independent of the method used to solve the linear systems (4) (if any). An array of linear solver modules is part of the package, and the core integrator connects with one of those according to a user selection made in advance. There are four linear solvers in this array at present, described in Sections 4 and 5 below, but

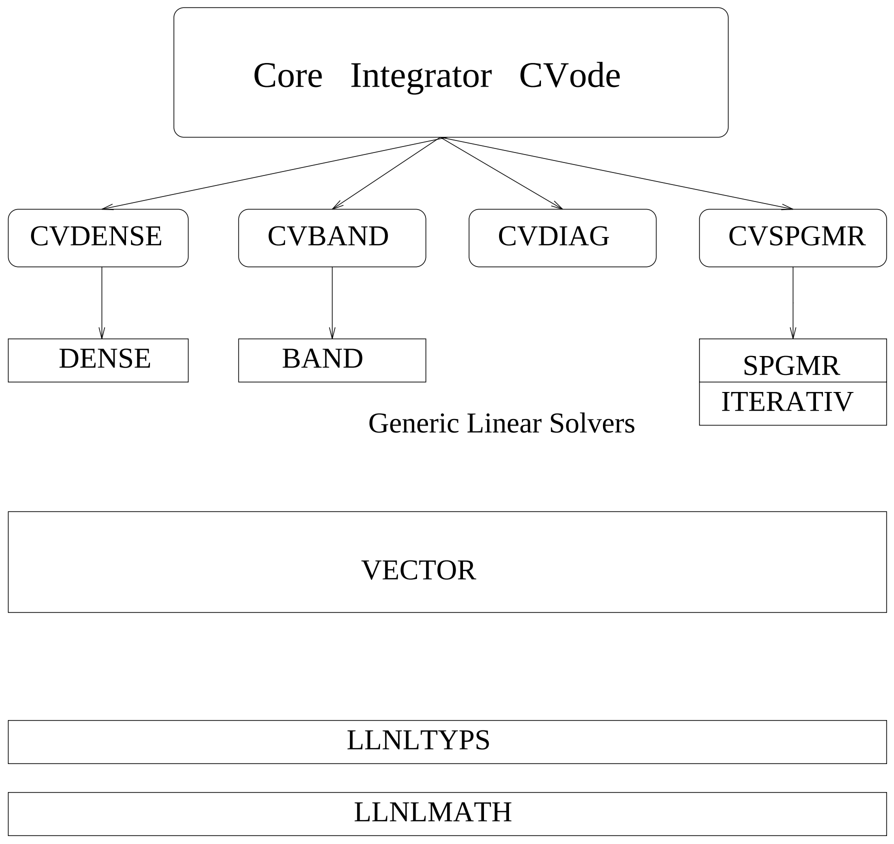

FIG. 1. *Overall block diagram of CVODE package.*

that number is likely to expand. The addition of new linear solvers will have no effect on the core integrator module.

The CVODE package includes a VECTOR module, described in Section 6 below, which contains kernels for vector operations. All operations on N-vectors are done by calls to this module.

Two small modules handle issues of arithmetic precision and low-level mathematical operations. The LLNLTYPS module contains declarations for the types `real` and `integer`, in a way that makes it very easy to change from double precision to single or vice-versa. The LLNLMATH module contains macros for operations such as MAX and MIN, power functions, and a (machine-independent) routine to compute the unit roundoff.

3. The CVODE User Interface. To use CVODE, a user must provide certain problem-defining routines and make calls to the solver, as in the case of Fortran solvers like VODE. We give here only a summary and a small example. Complete details, including three fully annotated examples, are given in the CVODE User Guide [4].

To start with, the user's calling program must make calls to either three or four functions in CVODE. This contrasts with VODE etc., because we have explicitly separated the actual integration calls from the loading of fixed inputs, the allocation and deallocation of memory, and the specification of the linear solver. These user calls are summarized as follows:

1. `CVodeMalloc(...)` receives problem and method specifications, and allocates memory. Its

arguments include problem definition quantities (N, f, t_0, y_0), method options, error tolerances, and optional input/output arrays. CVodeMalloc returns a pointer to the CVODE memory block (for CVODE state data and pointers).

2. A linear solver-specific routine specifies the linear solver used in the Newton iteration. At present, the routine called (if any) must be one of CVDense, CVBand, CVDiag, or CVSpgmr.
3. CVode(cvode_mem, tout, yout, t, itask) carries out the integration, returns a completion flag, and returns the solution vector in yout. If itask = NORMAL, it steps to tout, overshooting it and interpolating to get the solution at tout. If itask = ONE_STEP, it takes only one step towards tout and returns to the calling routine. CVode uses the memory allocated by CVodeMalloc to keep track of its state.
4. CVodeFree frees the memory allocated by CVodeMalloc.

The calling program must also create the dependent variable vector y and load it with the initial values. However, CVODE's VECTOR module includes a routine N_VNew for the allocation of an N-vector, and a routine N_VFree to free it.

The user-supplied routines consist of one to four functions, depending on the method options, as summarized below. The names are dummy names; arbitrary names can be passed.

1. f defines the function $f(t, y)$.
2. Jac supplies an approximate Jacobian in direct cases (CVDENSE, CVBAND). It is optional, and a default difference quotient Jac routine is available if none is supplied.
3. Precond and PSolve supply the preconditioner in Krylov case (CVSPGMR). These routines are optional, but there is no default. Precond sets Jacobian-related data, and does any matrix preprocessing needed. PSolve solves the preconditioner linear system.

In summary, then, a skeleton user program would typically look as follows. Here, we also illustrate the use of optional outputs, by accessing the number of steps taken as iopt[NST].

```
main() {

  y = N_VNew(...);

  mem = CVodeMalloc(N, f, t_0, y, ..., iopt, ...);

  CVDense or CVBand or CVDiag or CVSpgmr(...);

  for (tout= ...)
    flag = CVode(mem, tout, y, ...);
    printf("nst = %d", iopt[NST]);

  CVodeFree(...);

  N_VFree(y);
}

f(...) { ... }

Jac(...) { ... }  or

Precond(...) { ... } and PSolve(...) { ... }
```

Since, for various reasons, we have chosen not to use reverse communication in CVODE, the user must have a way of communicating data between the calling program and the user-supplied routines. This is accomplished with pointers that are passed into CVODE and back to the user routines, and can point to any structure of the user's design as appropriate for the application. There are three such pointers:

- `f_data`. The user passes this pointer to `CVodeMalloc`, and CVODE passes it to the user's `f` routine.
- `jac_data`. The user passes this pointer to `CVDense` or `CVBand`, and CVODE passes it to the user's `Jac` routine.
- `P_data`. The user passes this pointer to `CVSpgmr`, and CVODE passes it to the user's `Precond` and `PSolve` routines.

Of course, in the dense or band case, `jac_data` can be identical to `f_data`, and in the Krylov case, `P_data` can be identical to `f_data`.

A complete working example is shown below, in the Appendix. It solves a 3-species stiff chemical kinetics problem with dense difference-quotient treatment of the Jacobian. The output is given following the program.

4. The CVODE Linear Solvers. The linear solvers in the CVODE package form an expandable array of code modules that play a critical role in the success of CVODE on stiff systems. At present these modules are CVDENSE, CVBAND, CVDIAG, and CVSPGMR. Considerable thought has been given to the design of these modules, in both their internal and external aspects. Each linear solver module must interface with the user, with the core integrator module, with the generic linear system solver that supports it, and with the VECTOR and other lower level modules that support it.

As indicated in the previous section, the user specifies which linear solver is to be used by making a call to a routine `CVxxx`, the name being `CVDense`, `CVBand`, `CVDiag`, or `CVSpgmr`. This call also supplies inputs that are specific to that linear solver, such as associated user-supplied routines.

Aside from the `CVxxx` function, each linear solver (and any that are added in the future) must consist of the following four parts:

- Initialization function. This allocates memory for solver-specific data (e.g. the matrix M and pivot array). It also initializes solver-specific counters.
- Matrix setup function. This handles the computation of Jacobian-related data, if called for, in the context of the Newton iteration. It must perform any necessary preprocessing for the operation of the system solve function (e.g. LU factorization).
- System solve function. This carries out the solution of the linear system $Mx = b$, within the Newton iteration context. It uses solver-specific memory generated earlier, and may call a generic solver routine (e.g. triangular back-solve).
- Free function. This frees all solver-specific memory.

These four functions are called from within the core CVode module. In order that this module be independent of the particular linear solver selected, the call sequences of the four functions are fixed. The link from the core integrator module to the proper linear solver module is made by the `CVxxx` function, which sets pointers that reside in the CVODE memory block. The linear solver functions also receive a pointer to the CVODE memory block in order to share data about the state of the integration, and to connect with the solver-specific memory block.

These various connections are shown in Figure 2, in which the linear solver is CVDENSE for the sake of illustration. The CVODE memory block, to which a pointer `cvode_mem` is returned by `CVodeMalloc`, includes quantities like N, work arrays, and a pointer to the user's `f` function. It also includes pointers to the four parts of the linear solver module (`cv_linit` pointing to the initialization function, etc.) and a pointer `cv_lmem` to the solver-specific memory. These five pointers are set by (in this case) `CVDense`. The linear solver memory in this case includes the matrix M, a saved copy of the Jacobian J, a pivot array, and a pointer `d_jac` to the Jacobian routine. If the `Jac` argument of `CVDense` was not NULL, `d_jac` points to that function; otherwise it points to the difference quotient routine supplied in CVDENSE, called `CVDenseJacDQ`.

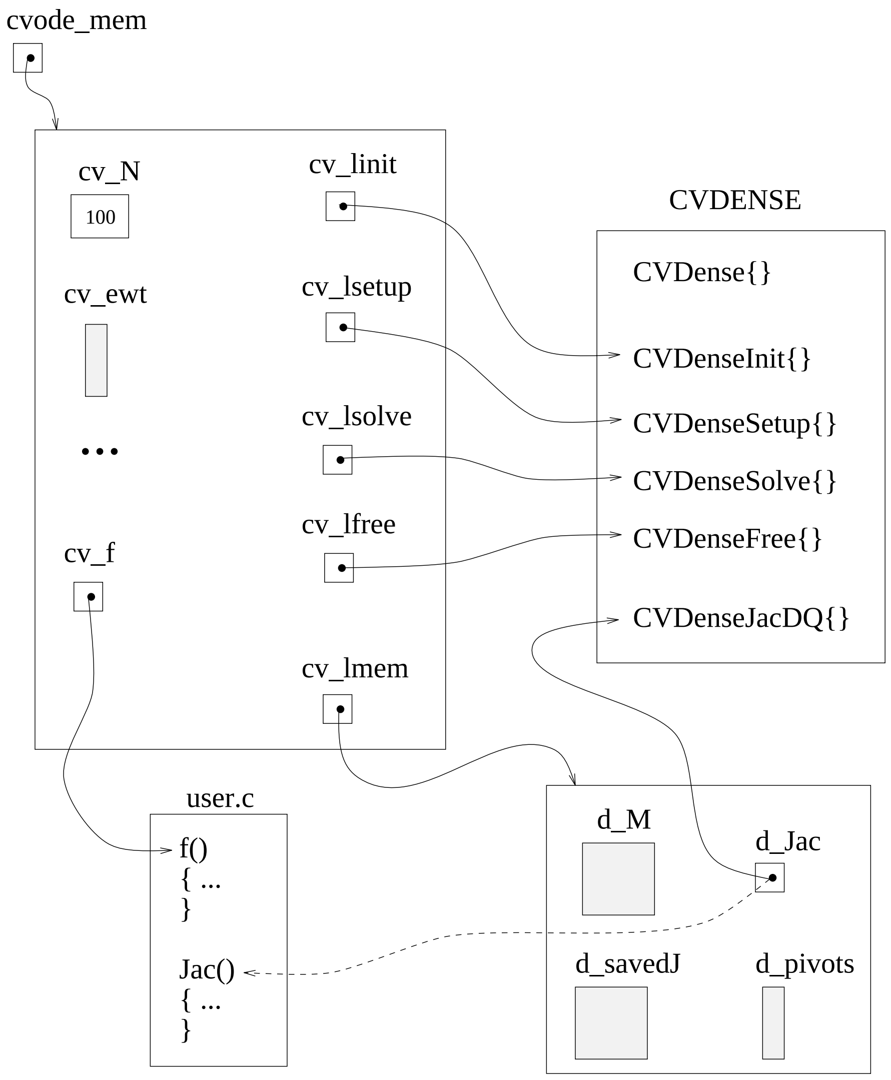

FIG. 2. *Link between CVODE Memory and a Linear Solver.*

5. Generic Solver Modules. An important criterion in the design of CVODE was that it make use of linear system solvers that are in generic form, i.e. code modules that are suitable as solvers in stand-alone form, with no reference to the ODE or Newton iteration context. The package includes (at present) three generic linear solver modules, shown in Figure 1. (The CVDIAG module requires no generic solver.) These modules can be described briefly as follows:

- DENSE solves dense linear systems by LU factorization and back-solving with partial pivoting. It is a rewrite in C of the LINPACK routine pair DGEFA/DGESL, augmented by functions for memory allocation and freeing, and some auxiliary operations (scaling, identity addition, etc.) The functions in DENSE operate on matrices of type `DenseMat` (defined in DENSE), and vectors of type `N_Vector` (defined in VECTOR), but in the sequential implementation they actually call functions that operate on ordinary arrays.

- BAND solves banded linear systems by LU factorization and back-solving with partial pivoting. It is a rewrite in C of the LINPACK routine pair DGBFA/DGBSL, augmented by functions for memory allocation and freeing, and some auxiliary operations. The functions in BAND operate on matrices of type `BandMat` (defined in BAND), and vectors of type `N_Vector`, but the sequential versions call functions that operate on ordinary arrays.
- SPGMR + ITERATIV solves arbitrary linear systems by a scaled preconditioned GMRES algorithm. The SPGMR module contains three user-callable functions — for memory allocation, linear system solution, and memory freeing. The ITERATIV module contains support functions — for modified and classical Gram-Schmidt procedures, and QR factorization and least-squares solution of a Hessenberg system.

The generic nature of these modules is evidenced by the fact that the SPGMR/ITERATIV module pair is being used successfully as the linear system solver in another application, a nonlinear steady-state plasma fluid simulation problem.

6. Basic Vector Kernels. The VECTOR module in the CVODE package is an isolated collection of kernels for all the vector operations on N-vectors performed within CVODE. This was done for two reasons. First, on any machine, it isolates the spots where machine-optimized versions of these operations can be incorporated. Secondly, on a parallel machine, it isolates the coding which must be altered to accommodate distributed vectors.

This module includes some operations in the Level-1 BLAS collection, but it includes others as well. The operations include memory allocation and freeing (with names `N_VNew, N_VFree`), vector arithmetic (with names `N_VLinearSum, N_VScale, N_VProd, N_VDiv, N_VConst`, etc.), and scalar measures (with names `N_VDotProd, N_VWrmsNorm, N_VMaxNorm, N_VMin`).

All of the kernels in the VECTOR module operate on vectors of type `N_Vector`, which is defined in the module. For the sequential version of CVODE, this type simply consists of the length and the initial address of a contiguous array. In order to insulate the user from the details of the `N_Vector` type, the module includes some macros for access to the vector data itself.

7. Parallel Extensions. A principal motivation for writing CVODE was to facilitate the development of ODE solvers on massively parallel processors (MPPs). This development is currently under way, in the form of (what will be) a set of parallel extensions to CVODE, called PVODE. PVODE is to run in the SPMD model (Single Program, Multiple Data), in which all vectors of length N are identically distributed across the processors, and all operations on vectors are done in parallel accordingly. As a minimum, what must be rewritten in the CVODE package is the VECTOR module, which is to have a different version specific to each MPP targeted.

Our initial effort includes only a subset of three nonlinear iteration method options, namely

- Functional iteration
- Newton iteration with CVDIAG, and
- Newton iteration with CVSPGMR.

Later we plan to include the Newton/CVBAND option by incorporating a parallel band solver.

The usage of PVODE involves the same calls as with the use of CVODE, but with an additional call at the beginning and another at the end. At the beginning, a call of the form

```
machEnv = PVInit<machine> ( ...  )
```

is required, with machine-dependent name and functionality, to set a block of information specific to machine environment, as needed by the VECTOR module. This block will include characteristics of the MPP, information about the vector distribution, and communication workspaces. The pointer `machEnv` is part of the `N_Vector` type, to allow access by the VECTOR kernels to this information. A final additional call will free the memory pointed to by `machEnv`.

At this time, versions of PVODE have been written for two machines, namely

- the Cray-T3D 256-processor MPP at LLNL, with its Shared Memory (SHMEM) programming model, and
- the IBM-SP2 128-node MPP at Argonne National Laboratory, using the MPI (Message Passing Interface) library MPICH.

In the T3D-SHMEM version, the VECTOR module uses efficient machine-specific communication routines from the SHMEM library. In contrast, the MPI version has no machine dependence, and assumes only that the MPI system (or at least a certain basic subset of it) has been implemented on the target machine. Since MPI is rapidly becoming a standard interface for message-passing on parallel machines, the MPI version of PVODE is highly portable to many MPP machines. (On any given machine, however, there may be difference in efficiency between a machine-specific and an MPI version.) Testing of these packages is still in progress.

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] S. D. Cohen and A. C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, September 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.

Appendix.

```
/* Example.  This chemical kinetics problem consists of 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 an internally
   generated (difference-quotient) 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, and
   run statistics (optional outputs) are printed at the end.            */

#include <stdio.h>
#include "llnltyps.h" /* definitions of types real and integer          */
#include "cvode.h"    /* prototypes for CVODE routines, and constants    */
#include "cvdense.h"  /* prototype for CVDense, constant DENSE_NJE       */
#include "vector.h"   /* definitions of type N_Vector and macro N_VIth,  */
#include "dense.h"    /* definitions of type DenseMat, macro DENSE_ELEM  */
#define NEQ   3            /* number of equations  */

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

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

  N_VIth(y,0) = 1.0; N_VIth(y,1) = 0.0; N_VIth(y,2) = 0.0;  /* Set initial y */

  reltol = 1.0e-4;                                    /* Set the tolerances */
  N_VIth(abstol,0) = 1e-8; N_VIth(abstol,1) = 1e-14; N_VIth(abstol,2) = 1e-6;

  /* Call CVodeMalloc to initialize CVODE */
  cvode_mem = CVodeMalloc(NEQ, f, 0.0, 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 dense linear solver with internal Jacobian. */
  CVDense(cvode_mem, NULL, NULL);
  /* In loop over output times, call CVode, print results, test for error */
  printf(" \n3-species kinetics problem\n\n");
  for (iout=1, tout=0.4; iout <= 12; iout++, tout *= 10.0) {
    flag = CVode(cvode_mem, tout, y, &t, NORMAL);
    printf("At t = %0.4e      y =%14.6e  %14.6e  %14.6e\n",
           t, N_VIth(y,0), N_VIth(y,1), N_VIth(y,2));
    if (flag != SUCCESS) { printf("CVode failed, flag=%d.\n", flag); break; }
  }

  N_VFree(y); N_VFree(abstol);      /* Free the y and abstol vectors */
  CVodeFree(cvode_mem);             /* Free the CVODE problem memory */
  printf("\nFinal Statistics.. \n\n");   /* Print some final statistics  */
  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]);
}

static void f(integer N, real t, N_Vector y, N_Vector ydot, void *f_data)
{
  real y1, y2, y3, yd1, yd3;
  y1 = N_VIth(y,0); y2 = N_VIth(y,1); y3 = N_VIth(y,2);

  yd1 = N_VIth(ydot,0) = -0.04*y1 + 1e4*y2*y3;
  yd3 = N_VIth(ydot,2) = 3e7*y2*y2;
        N_VIth(ydot,1) = -yd1 - yd3;
}
```

```
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    9.185043e-06    2.841892e-01
At t = 4.0000e+02      y =  4.505209e-01    3.222829e-06    5.494759e-01
At t = 4.0000e+03      y =  1.832118e-01    8.942573e-07    8.167873e-01
At t = 4.0000e+04      y =  3.898201e-02    1.621690e-07    9.610178e-01
At t = 4.0000e+05      y =  4.938094e-03    1.984922e-08    9.950619e-01
At t = 4.0000e+06      y =  5.172663e-04    2.070134e-09    9.994827e-01
At t = 4.0000e+07      y =  5.202999e-05    2.081305e-10    9.999480e-01
At t = 4.0000e+08      y =  5.213911e-06    2.085575e-11    9.999948e-01
At t = 4.0000e+09      y =  5.213453e-07    2.085382e-12    9.999995e-01
At t = 4.0000e+10      y =  5.653959e-08    2.261584e-13    9.999999e-01

Final Statistics..

nst = 529    nfe  = 774    nsetups = 102    nje = 11
nni = 738    ncfn = 0      netf = 19
```