SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers

Daniel R. Reynolds

reynolds@smu.edu

Department of Mathematics
Southern Methodist University

Argonne Training Program on Extreme-Scale Computing
August 2015

SUite of Nonlinear and DIfferential/ALgebraic equation Solvers

- Suite of time integrators and nonlinear solvers
  - ODE/DAE time integrators w/ forward & adjoint sensitivity capabilities
  - Newton and fixed-point nonlinear solvers
  - Written in ANSI C with Fortran interfaces
  - Designed to be easily incorporated into existing codes
  - Modular implementation with swappable components:
    - Linear solvers – direct dense/band/sparse, iterative
    - Preconditioners – diagonal and block diagonal provided
    - Vector structures (core data structure for all solvers) – supplied with serial, threaded and MPI parallel
    - Users may supply custom versions of any of the above
  - Freely available (BSD license)

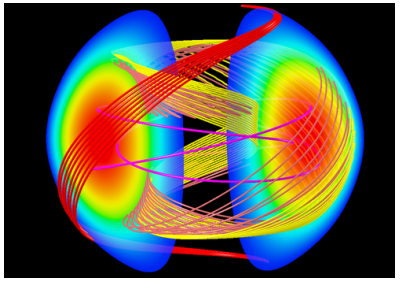
sundials

https://computation.llnl.gov/casc/sundials/main.html

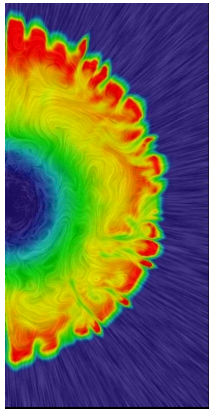
FASTMATH Argonne NATIONAL LABORATORY BERKELEY LAB Sandia National Laboratories Berkeley UNIVERSITY OF CALIFORNIA Rensselaer SMU UBC

SUNDIALS is used worldwide in applications from research and industry

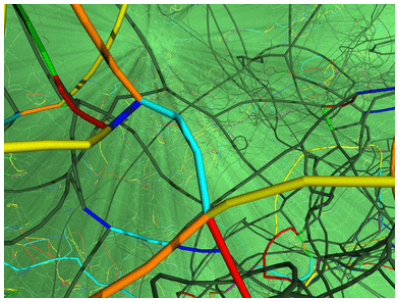
- Power grid modeling (RTE France, ISU)
- Simulation of clutches and power train parts (LuK GmbH & Co.)
- Electrical and heat generation within battery cells (CD-adapco)
- 3D parallel fusion (SMU, U. York, LLNL)
- Implicit hydrodynamics in core collapse supernovae (Stony Brook)
- Dislocation dynamics (LLNL)
- Sensitivity analysis of chemically reacting flows (Sandia)
- Large-scale subsurface flows (CO Mines, LLNL)
- Optimization in simulation of energy-producing algae (NREL)
- Micromagnetic simulations (U. Southampton)



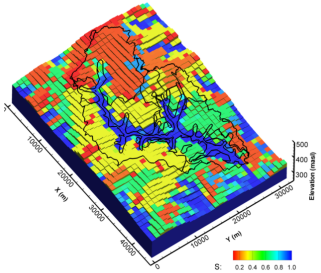
Magnetic reconnection



Core collapse supernovae



Dislocation dynamics



Subsurface flow

Over 3500 downloads each year

CVODE Solves IVPs: $\dot{y} = f(t, y), \quad y(t_0) = y_0$

- Variable order and step size Linear Multistep Methods ($y_n \approx y(t_n)$)

$$\sum_{j=0}^{K_1} \alpha_{n,j} y_{n-j} + \Delta t_n \sum_{j=0}^{K_2} \beta_{n,j} f(t_{n-j}, y_{n-j}) = 0 \tag{1}$$

- Adams-Moulton (nonstiff): $K_1 = 1, \quad K_2 = k, \quad k = 1, \ldots, 12$
- Fixed-leading coefficient BDF (stiff): $K_1 = k, \quad K_2 = 0, \quad k = 1, \ldots, 5.$
- Optional stability limit detection algorithm for orders > 2
- Nonlinear solvers execute a predictor-corrector scheme:

Explicit predictor provides $y_{n(0)}$:

$$y_{n(0)} = \sum_{j=1}^{q} \alpha_j^p y_{n-j} + \Delta t \beta_1^p f(t_{n-1}, y_{n-1})$$

Implicit corrector solves (1):

- $y_{n(0)}$ is the initial iterate
- nonstiff: fixed-point iteration
- stiff: Newton-based iteration

Convergence and errors are measured against user-specified tolerances

- User-defined tolerances:
  - Absolute tolerance on each solution component, $ATOL^i$
  - Relative tolerance for all solution components, $RTOL$

  define a weight vector,

$$ewt^i = \frac{1}{RTOL|y^i| - ATOL^i}$$

- Errors are then measured with a *weighted root-mean-square norm*

$$\|y\|_{WRMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(ewt^i\ y^i)}$$

- Choose time steps Δt_n to bound an estimate of the local temporal error:

$$\|y_n - y_{n(0)}\|_{WRMS} < \frac{1}{6}$$

Choose time steps to minimize local error & maximize efficiency

- Time steps selection criteria:
  1. Estimate the temporal error: $E(\Delta t) = C(y_n - y_{n(0)})$
     - Accept step if $\|E(\Delta t)\|_{WRMS} < 1$
     - Reject step otherwise
  2. Estimate error at the next step, $\Delta t'$, as (q is current method order)

$$E(\Delta t') \approx \left(\frac{\Delta t'}{\Delta t}\right)^{q+1} E(\Delta t)$$

  3. Choose next step so that $\|E(\Delta t')\|_{WRMS} < 1$
- Method order selection criteria:
  - Estimate error and prospective steps $\Delta t'$ for orders $\{q-1, q, q+1\}$
  - Choose order resulting in the largest time step that meets the error condition

ARKode Solves IVPs: $M\dot{y} = f_E(t,y) + f_I(t,y), \quad y(t_0) = y_0$

- Split system into stiff f_I and nonstiff f_E components
- M may be the identity or any nonsingular mass matrix (e.g. FEM)
- Variable step size *additive Runge-Kutta methods* – combine explicit (ERK) and diagonally-implicit (DIRK) RK methods to enable ImEx solver (disable either for pure explicit/implicit). Let $t_{n,j} \equiv t_{n-1} + c_j \Delta t_n$:

$$Mz_i = My_{n-1} + \Delta t_n \sum_{j=0}^{i-1} A^E_{i,j} f_E(t_{n,j}, z_j) + \Delta t_n \sum_{j=0}^{i} A^I_{i,j} f_I(t_{n,j}, z_j),$$

$$My_n = My_{n-1} + \Delta t_n \sum_{j=0}^{s} b_i \left[f_E(t_{n,j}, z_j) + f_I(t_{n,j}, z_j) \right],$$

$$M\tilde{y}_n = My_{n-1} + \Delta t_n \sum_{j=0}^{s} \tilde{b}_i \left[f_E(t_{n,j}, z_j) + f_I(t_{n,j}, z_j) \right].$$

- Solve for stage solutions $z_i = 1, \ldots, s$ sequentially (via Newton, fixed-point, linear solver, or just vector updates)

ARKode: the newest member of SUNDIALS

- Time-evolved solution y_n; embedded solution $\tilde{y}_n$
- Error estimate $E(\Delta t_n) = \|y_n - \tilde{y}_n\|_{WRMS}$
- Fixed order within each solve:
  - ARK $\mathcal{O}(\Delta t^3) \to \mathcal{O}(\Delta t^5)$
  - DIRK $\mathcal{O}(\Delta t^2) \to \mathcal{O}(\Delta t^5)$
  - ERK $\mathcal{O}(\Delta t^2) \to \mathcal{O}(\Delta t^6)$
  - user-supplied
- Multistage embedded methods (as opposed to multistep):
  - High order without solution history (enables spatial adaptivity)
  - Sharp estimates of solution error even for stiff problems
  - *But, DIRK/ARK require multiple implicit solves per step*
- User interface modeled on CVODE $\Rightarrow$ simple transition between solvers

Documentation/examples – http://faculty.smu.edu/reynolds/arkode

Initial value problems (IVPs) come in the form of ODEs and DAEs

- The most general form of an IVP is given by

$$f(t, y, \dot{y}) = 0, \quad y(t_0) = y_0$$

- If $\frac{\partial f}{\partial \dot{y}}$ is invertible, we typically solve for $\dot{y}$ to obtain an ordinary differential equation (ODE), but this is not always the best approach.
- Else, the IVP is a differential algebraic equation (DAE)
- A DAE has *differentiation index* i if i is the minimal number of analytical differentiations needed to extract an explicit ODE system.

IDA Solves DAEs: $f(t, y, \dot{y}) = 0, \quad y(t_0) = y_0$

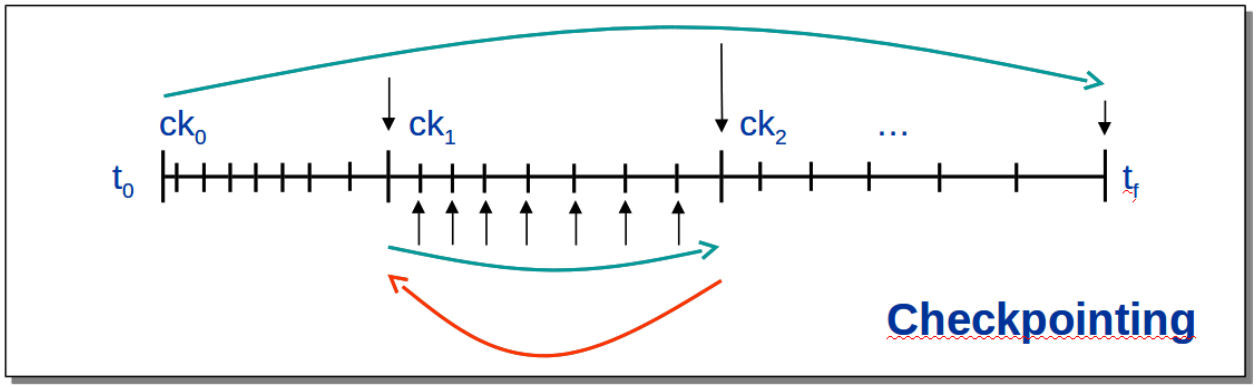
- Variable order and step size BDF (no Adams)
- C rewrite of DASPK [Brown, Hindmarsh, Petzold]
- Uses: implicit ODEs, index-1 DAEs, and Hessenberg index-2 DAEs
- Optional routine solves for consistent initial conditions y_0 and $\dot{y}_0$:
  - Semi-explicit index-1 DAEs
  - differential components known, algebraic unknown, OR
  - all of $\dot{y}_0$ specified, but y_0 unknown
- Nonlinear systems solved by Newton-based method
- Optional constraints: $y^i > 0$, $y^i < 0$, $y^i \geq 0$, $y^i \leq 0$.

Sensitivity Analysis – CVODES and IDAS

- Sensitivity Analysis (SA) is the study of how variations in the output of a model (numerical or otherwise) can be apportioned, qualitatively or quantitatively, to different sources of variation in inputs.
- Applications:
  - Model evaluation (most and/or least influential parameters)
  - Model reduction
  - Data assimilation
  - Uncertainty quantification
  - Optimization (parameter estimation, design optimization, optimal control, ...)
- Approaches:
  - Forward sensitivity analysis (FSA) – augments the state system with sensitivity equations.
  - Adjoint sensitivity analysis (ASA) – solves a backward in time adjoint problem (user supplies the adjoint problem).

Adjoint Sensitivity Analysis Implementation

- Solution of the forward problem is required for the adjoint problem: need predictable and compact storage of solution values for the solving the adjoint system



- Simulations are reproducible from each checkpoint
- Cubic Hermite or variable-degree polynomial interpolation between ck_i
- Store solution and first derivative at each checkpoint
- Force Jacobian evaluation at checkpoints to reduce storage
- Computational cost: 2 forward and 1 backward integrations

KINSOL Solves Nonlinear Systems: $F(u) = 0$

- C rewrite of Fortran NKSOL (Brown and Saad)
- Newton Solvers: update iterate via $u^{k+1} = u^k + s^k,\ k = 0, 1, \ldots,$
  - *Inexact*: approx. solves $J(u^k)s^k = -F(u^k)$, where $J(u) = \frac{\partial F(u)}{\partial u}$
  - *Modified*: directly solves $J(u^{k-l})s^k = -F(u^k)$, where $l \geq 0$
- Optional constraints: $u_i > 0$, $u_i < 0$, $u_i \geq 0$ or $u_i \leq 0$
- Can separately scale equations $F_i(u)$ and/or unknowns u_i
- Backtracking line search option to aid robustness
- Dynamic linear tolerance selection for use with iterative linear solvers

$$\left\| F(u^k) + J(u^k)s^k \right\| \leq \eta^k \left\| F(u^k) \right\|$$

New additions: accelerated fixed point and Picard solvers

Fixed-point iterations solve the fixed-point problem, $u = G(u)$ via the recursion

$$u^{k+1} = G(u^k), \quad k = 0, 1, \ldots$$

- Basic fixed point solvers define $\quad G(u) \equiv u - F(u)$
- Picard splits F into linear & nonlinear parts, $F(u) \equiv Lu - N(u)$, defines

$$G(u) \equiv L^{-1}N(u) = u - L^{-1}F(u) \quad \Rightarrow \quad u^{k+1} = u^k - L^{-1}F(u^k)$$

  [i.e. like Newton, but with $L \approx J(u^k)$]
- If G is a contraction, i.e. for all x, y, there exists $\gamma < 1$ such that

$$\|G(x) - G(y)\| \leq \gamma \|x - y\|,$$

  the fixed point iteration converges *globally*, but at a *linear* rate
- As of SUNDIALS v.2.6.0, KINSOL includes Picard and *accelerated* fixed point solvers; ARKode includes the accelerated fixed point solver.

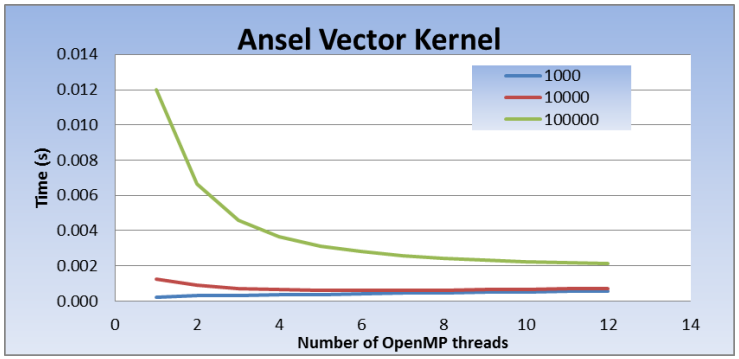
The SUNDIALS vector module is generic

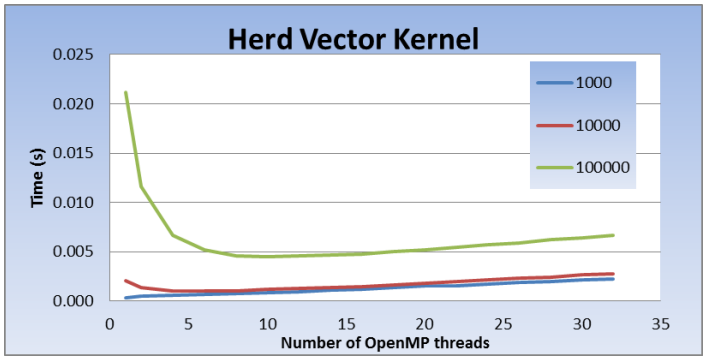
- Data vector structures can be user-supplied for problem-specific versions
- Essentially follows an object-oriented base/derived class approach (but in C), with most† solvers defined on the base class
- The generic `NVECTOR` module ("abstract base class") defines:
  - A `content` structure (`void *`)
  - An `ops` structure, containing function pointers to vector operations
- Each `NVECTOR` implementation ("derived class") defines:
  - Content structure – specifies the actual vector data and any information needed to make new vectors (problem or grid data)
  - Implemented vector operations
  - Routines to clone vectors
- All parallel communication resides in reduction ops: $u \cdot v$, $\|u\|$, etc.

†Only the direct linear solvers require specific derived vector modules

SUNDIALS provides serial, threaded and parallel NVECTOR implementations

- Vectors are laid out as a 1D array of doubles (or floats)
- Appropriate lengths (local, global) are specified at construction
- Operations are fast due to unit stride
- All operations are provided for each implementation: Serial, Threaded (OpenMP), Threaded (Pthreads), and Distributed-memory parallel (MPI)
- Can serve as templates for creating a user-supplied vector – these need only implement a minimal subset of vector ops for desired solver(s)
- Threaded kernels require long vectors (>10k) for appreciable benefit:





SUNDIALS provides many linear solver options

- Iterative Krylov (all allow scaling & preconditioning)
  - All solvers have GMRES, BiCGStab & TFQMR; KINSOL also has FGMRES; ARKode also has PCG & FGMRES
  - Only require matrix-vector products, Jv; may be user supplied, or estimated via $Jv \approx \frac{1}{\epsilon}\left[F(u+\epsilon v) - F(u)\right]$
  - Require preconditioning for scalability (see next slide)
- Dense direct (dense or banded)
  - Require serial/threaded vector environments; banded requires some predefined structure to the problem
  - J can be user supplied or estimated with finite differencing
- Sparse direct interfaces to external solver libraries
  - Currently requires serial or threaded vector environments
  - J must be supplied in *compressed sparse column* format
  - Allows KLU (serial) & SuperLU_MT (threaded); considering PARDISO (threaded) & SuperLU_DIST (parallel) for future releases

Krylov methods require preconditioning for scalability to large problems

- Competing preconditioner goals:
  - P should approximate the Newton matrix
  - P should be efficient to construct and solve
- For example with CVODE, a typical $P = I - \gamma \tilde{J}$, where $\tilde{J} \approx J$
- User can save and reuse P, as directed by the solver
- Banded and block-banded preconditioners are supplied for use with the included vector structures
- Setup/Solve hooks enable user-supplied preconditioning:
  - Setup: evaluate and preprocess P (infrequently)
  - Solve: solve systems $Px = b$ (frequently)
  - Can use *Hypre* or *SuperLU_DIST* or *PETSc* or ...

CVODE, ARKode and IDA are equipped with rootfinding capabilities

- Finds roots of user-defined functions, $g_i(t, y)$ or $g_i(t, y, \dot{y}),\ i = 1, \ldots, n_r$
- Useful when problem definition may change based on changes in solution
- Find roots by monitoring sign changes (odd multiplicity only)
- Checks each time step for sign change in $g_i,\ i = 1, \ldots, n_r$
- If a sign changes in g_l:
  - Applies a modified secant method with a tight tolerance to find t^* such that $g_l(t^*, y) = 0$
  - Computes $g_l(t^* + \delta, y)$ for a small δ in the direction of integration
  - Integration stops if any $g_l(t^* + \delta, y) = 0$
- Ensures that values of g_l are nonzero at some past value of t, beyond which a search for roots is valid

SUNDIALS provides Fortran interfaces

- Fortran interfaces provided for CVODE, ARKode, IDA, and KINSOL (not CVODES or IDAS)
- Cross-language calls go in both directions:
  - (a) Fortran program calls solver creation, setup, solve, and output interface routines
  - (b) Solver routines call users problem-defining RHS/residual, matrix-vector product, and preconditioning routines
- For portability across compilers, all routines in (b) have fixed names
- Examples are provided for each solver

SUNDIALS code usage is similar across the suite

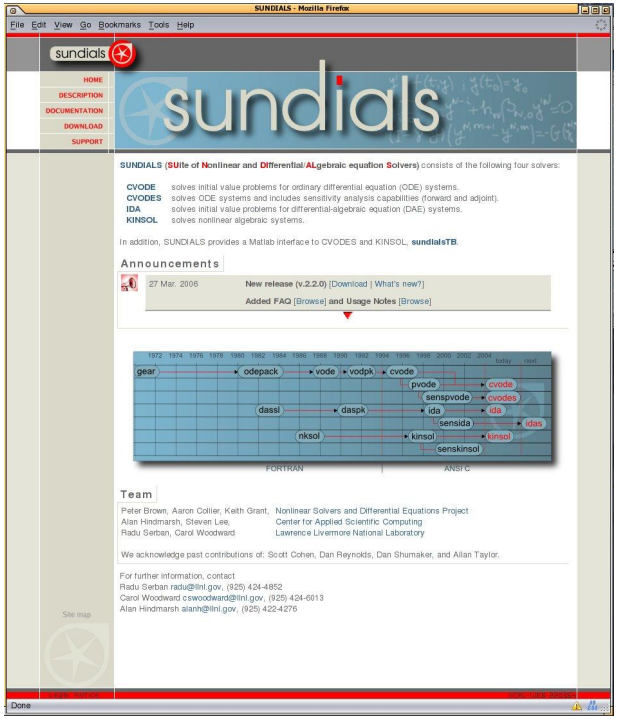
Core components to any user program (example follows – time permitting):

1. `#include` header files for solver(s) and vector implementation
2. Create data structure for solution vector
3. Set up solver:
   a. Create solver object (allocates solver-specific memory structure)
   b. Initialize solver (sets solution vector, problem-defining function pointers, default solver parameters)
   c. Call "Set" routines to customize solver behavior/parameters
4. Set up linear solver [if needed for Newton/Picard]:
   a. Create linear solver object
   b. Call "Set" routines to customize linear solver behavior/parameters
5. Call solver (once or repeatedly)
6. Destroy solver & vectors to free memory

Availability

Open source BSD license
http://computation.llnl.gov/casc/sundials

Publications
http://computation.llnl.gov/casc/sundials/documentation/documentation.html



Web site:

- Individual codes download
- SUNDIALS suite download
- User manuals
- User group email list

The SUNDIALS Team:
Carol S. Woodward, Daniel R. Reynolds, Alan C. Hindmarsh, & Lawrence E. Banks.
We acknowledge significant past contributions of Radu Serban.

Example food web problem for KINSOL [kinFoodWeb_kry_p.c]

Vesta: `/projects/FASTMath/ATPESC-2015/examples/sundials/kinsol/parallel`

A food web population model, with predator-prey interaction and diffusion on $\Omega = [0,1]^2$. PDE system with dependent variables $c_1, c_2, \ldots, c_{n_s}$:

$$d_i \nabla^2 c_i = -c_i\, b_i(x,y) \sum_{j=1}^{n_s} a_{i,j} c_j, \qquad (x,y) \in \Omega$$

$$\nabla c_i \cdot n = 0, \qquad (x,y) \in \partial\Omega, \quad i = 1, \ldots, n_s.$$

Here, $n_s = 2\, n_p$, with the first n_p being prey and the last n_p being predators. The coefficients $a_{i,j}$, b_i, and d_i are:

$$a_{i,j} = \begin{cases} -\mathsf{AA}, & i == j, \\ -\mathsf{GG}, & i \le n_p,\ j > n_p, \\ \mathsf{EE}, & i > n_p,\ j \le n_p, \end{cases}$$

$$b_i = \begin{cases} \mathsf{BB}(1 + \alpha\, xy), & i \le n_p, \\ -\mathsf{BB}(1 + \alpha\, xy), & i > n_p, \end{cases}$$

$$d_i = \begin{cases} \mathsf{DPREY}, & i \le n_p, \\ \mathsf{DPRED}, & i > n_p. \end{cases}$$

1. Include solver header files and define problem-specific constants

```
/* header files */
#include <kinsol/kinsol.h>
#include <kinsol/kinsol_spgmr.h>
#include <nvector/nvector_parallel.h>
#include <sundials/sundials_dense.h>
#include <sundials/sundials_types.h>
#include <sundials/sundials_math.h>
#include <mpi.h>

/* problem-defining constants */
#define NUM_SPECIES   6
#define NPEX          2
#define NPEY          2
#define MXSUB         10
#define MYSUB         10

#define MX      (NPEX*MXSUB)
#define MY      (NPEY*MYSUB)

#define NEQ    (NUM_SPECIES*MX*MY)
```

2. Define problem-specific data structure

```
/* Type: UserData contains preconditioner blocks,
   pivot arrays, and problem parameters */
typedef struct {
  realtype **P[MXSUB][MYSUB];
  long int *pivot[MXSUB][MYSUB];
  realtype **acoef, *bcoef;
  N_Vector rates;
  realtype *cox, *coy;
  realtype ax, ay, dx, dy;
  realtype uround, sqruround;
  int mx, my, ns, np;
  realtype cext[NUM_SPECIES * (MXSUB+2)*(MYSUB+2)];
  int my_pe, isubx, isuby, nsmxsub, nsmxsub2;
  MPI_Comm comm;
} *UserData;
```

3. Declare functions to define problem and perform preconditioning

```
/* User-Provided Functions Called by the KINSol Solver */

/*    nonlinear residual function, computes F(c) */
static int funcprpr(N_Vector cc, N_Vector fval,
                    void *user_data);

/*    block-diagonal preconditioner setup routine */
static int Precondbd(N_Vector cc, N_Vector cscale,
                     N_Vector fval, N_Vector fscale,
                     void *user_data, N_Vector vtemp1,
                     N_Vector vtemp2);

/*    preconditioner solve routine */
static int PSolvebd(N_Vector cc, N_Vector cscale,
                    N_Vector fval, N_Vector fscale,
                    N_Vector vv, void *user_data,
                    N_Vector vtemp);
```

4. Declare helper functions for use in main and by provided F & P

```
/* Private Helper Functions */
static UserData AllocUserData(void);
static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector cc, N_Vector sc);
static void PrintHeader(int globalstrategy, int maxl, int maxlrst,
                        realtype fnormtol, realtype scsteptol);
static void PrintOutput(int my_pe, MPI_Comm comm, N_Vector cc);
static void PrintFinalStats(void *kmem);
static void WebRate(realtype xx, realtype yy, realtype *cxy,
                    realtype *ratesxy, void *user_data);
static realtype DotProd(int size, realtype *x1, realtype *x2);
static void BSend(MPI_Comm comm, int my_pe, int isubx, int isuby,
                  int dsizex, int dsizey, realtype *cdata);
static void BRecvPost(MPI_Comm comm, MPI_Request request[],
                      int my_pe, int isubx, int isuby, int dsizex,
                      int dsizey, realtype *cext, realtype *buffer);
static void BRecvWait(MPI_Request request[], int isubx, int isuby,
                      int dsizex, realtype *cext, realtype *buffer);
static void ccomm(realtype *cdata, UserData data);
static void fcalcprpr(N_Vector cc, N_Vector fval,void *user_data);
static int check_flag(void *flagvalue, char *funcname,
                      int opt, int id);
```

5. Begin main: initialize MPI and user data structure

```
int main(int argc, char *argv[]) {

  /* initialize MPI, set communicator */
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;

  /* get total number of pe's and my rank */
  int my_pe, npes, npelast=NPEX*NPEY-1;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

  /* set local vec length */
  long int local_N = NUM_SPECIES*MXSUB*MYSUB;

  /* allocate/initialize user data structure */
  UserData data = AllocUserData();
  InitUserData(my_pe, comm, data);
```

6. Allocate solution/scaling/constraint vectors, create solver object

```
/* Set globalization strategy */
int globalstrategy = KIN_NONE;

/* Allocate and initialize vectors */
N_Vector cc = N_VNew_Parallel(comm, local_N, NEQ);
N_Vector sc = N_VNew_Parallel(comm, local_N, NEQ);
data->rates = N_VNew_Parallel(comm, local_N, NEQ);
N_Vector constraints = N_VNew_Parallel(comm, local_N, NEQ);
SetInitialProfiles(cc, sc);
N_VConst(ZERO, constraints);

/* Set tolerances */
realtype fnormtol=RCONST(1.e-7), scsteptol=RCONST(1.e-13);

/* Call KINCreate to allocate KINSOL problem memory;
   a pointer which is returned & stored in kmem */
void* kmem = KINCreate();
```

7. Initialize solver object, set optional data and parameters

```
/* Call KINInit to initialize KINSOL, sending cc as a
   template vector */
int flag = KINInit(kmem, funcprpr, cc);

/* Call KINSet* routines to set solver parameters;
   the return flag may be checked to assert success */
flag = KINSetNumMaxIters(kmem, 250);
flag = KINSetUserData(kmem, data);
flag = KINSetConstraints(kmem, constraints);
flag = KINSetFuncNormTol(kmem, fnormtol);
flag = KINSetScaledStepTol(kmem, scsteptol);

/* We no longer need the constraints vector since
   KINSetConstraints creates a private copy */
N_VDestroy_Parallel(constraints);
```

8. Create GMRES linear solver, set parameters and P routines

```
/* Set up GMRES as the linear solver, with Krylov
   subspace of dimension at most 20, and a maximum
   of 2 restarts */
int maxl = 20, maxlrst = 2;
flag = KINSpgmr(kmem, maxl);
flag = KINSpilsSetMaxRestarts(kmem, maxlrst);

/* Set Precondbd and PSolvebd as the preconditioner
   setup and solve routines */
flag = KINSpilsSetPreconditioner(kmem,
                                 Precondbd,
                                 PSolvebd);
```

9. Call solver, output statistics, and clean up

```
  /* Call KINSol and print output profile */
  flag = KINSol(kmem,     /* solver memory */
        cc,               /* in: guess; out: sol'n */
        globalstrategy,   /* nonlinear strategy */
        sc,               /* scaling vector for unknowns */
        sc);              /* scaling vector for fcn vals */

  /* Print final statistics */
  if (my_pe == 0) PrintFinalStats(kmem);

  /* Free memory, finalize MPI and quit */
  N_VDestroy_Parallel(cc);
  N_VDestroy_Parallel(sc);
  KINFree(&kmem);
  FreeUserData(data);
  MPI_Finalize();
  return(0);
}
```