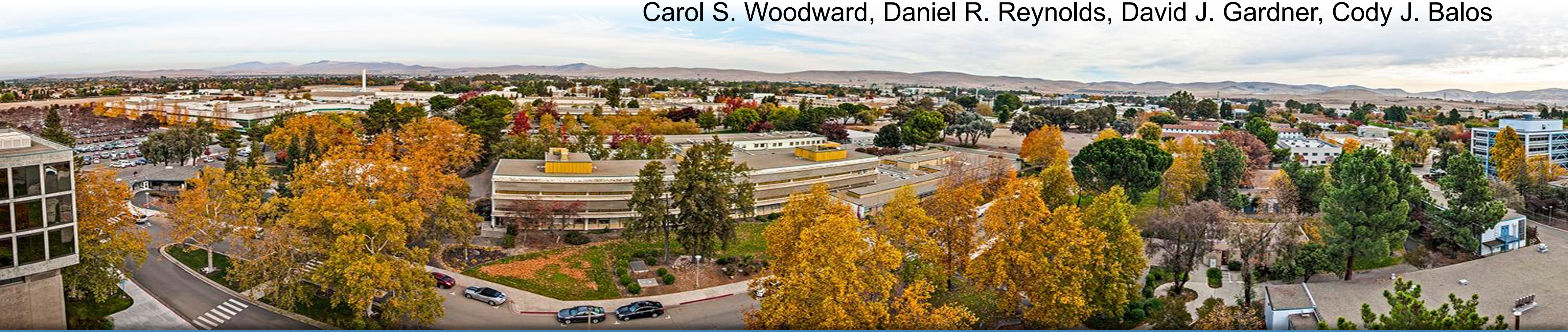
Introduction to the Capabilities and Use of the SUNDIALS Suite of Nonlinear and Differential/Algebraic Equation Solvers

ECP Annual Meeting, Houston, TX

Jan. 15, 2019

Carol S. Woodward, Daniel R. Reynolds, David J. Gardner, Cody J. Balos



LLNL-PRES-765149
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

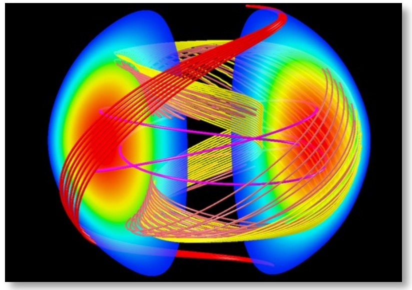
Tutorial Outline

- **Overview of SUNDIALS (Carol Woodward)**
- How to download and install SUNDIALS (Cody Balos)
- How to use the time integrators (Daniel Reynolds)
- Which nonlinear and linear solvers are available and how to use them (David Gardner)

ODEs and DAEs Arise in Numerous Application Areas

- **Ordinary Differential Equations** (ODEs) $\dot{y} = f(t,y), \quad y(t_0) = y_0$

  – PDEs: Method of lines discretization *f* contains discrete spatial operations
  – Chemical reactions: *f* includes terms for each reaction

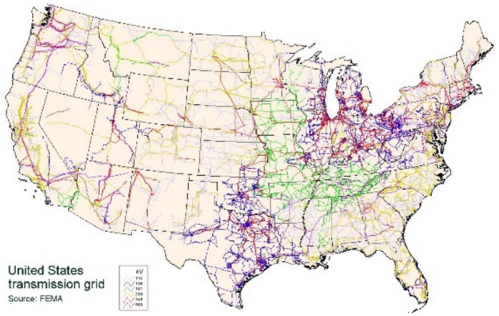


Magnetic reconnection

- **Differential Algebraic Equations** (DAEs)

$$F(t,y,\dot{y}) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0$$

  – PDEs: Method of lines discretization with algebraic constraints
  – Power transmission models: *F* includes differential equations for power generators and network-based algebraic system constraining power flow
  – Electronic circuit models
  – If $\partial F/\partial y$ is invertible, we can solve for $\dot{y}$ to obtain an ODE, but this is not always the best approach, else the system is a DAE.



US Transmission grid (Wikimedia Commons)

sundials SUite of Nonlinear and DIfferential-ALgebraic Solvers

- SUNDIALS is a software library consisting of ODE and DAE integrators and nonlinear solvers
  - 6 packages: CVODE(S), IDA(S), ARKode, and KINSOL
- Written in C with interfaces to Fortran
- Designed to be incorporated into existing codes
- Data use is fully encapsulated into vectors which can be user-supplied
- Nonlinear and linear solvers are fully encapsulated from the integrators and can be user-supplied
- All parallelism is encapsulated in vectors modules, solver modules, and user-supplied functions
- Freely available; released under the BSD 3-Clause license (>25,000 downloads in 2018)
- Active user community supported by sundials-users email list
- Detailed user manuals are included with each package

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

CVODE(S) and IDA(S) employ variable order and step BDF methods for integration

- CVODE solves ODEs *($\dot{y}$ = f(t, y))*
- IDA solves $F(t, y, \dot{y}) = 0$
  - Targets: implicit ODEs, index-1 DAEs, and Hessenberg index-2 DAEs
  - Optional routine solves for consistent values of y_0 and $\dot{y}_0$ for some cases
- Variable order and variable step size Linear Multistep Methods

$$\sum_{j=0}^{K_1} \alpha_{n,j} y_{n-j} + \Delta t_n \sum_{j=0}^{K_2} \beta_{n,j} \dot{y}_{n-j} = 0$$

- Both packages include stiff BDF method up to 5th order (K_1 = 1,…,5 and K_2 = 0)
- CVODE includes nonstiff Adams-Moulton methods up to 12th order (K_1 = 1, K_2 = 1,…,12)
- Both packages include rootfinding for detecting sign change in solution-dependent functions
- CVODES and IDAS include both forward and adjoint (user supplies the adjoint operator) sensitivity analysis

ARKode is the newest package in SUNDIALS

- ARKode solves ODEs $M\dot{y} = f_I(t,y) + f_E(t,y), \quad y(t_0) = y_0$
  - M may be the identity or any nonsingular mass matrix (e.g., FEM)
- Multistage embedded methods (as opposed to multistep):
  - High order without solution history (enables spatial adaptivity)
  - Sharp estimates of solution error even for stiff problems
  - Implicit and additive multistage methods require multiple implicit solves per step
- Supplied with three steppers now (but easy to add others)
  - ERKStep: explicit Runge-Kutta methods for $\dot{y} = f(t,y), \quad y(t_0) = y_0$
  - ARKStep: explicit, implicit, or IMEX methods for $M\dot{y} = f_I(t,y) + f_E(t,y), \quad y(t_0) = y_0$
    - Split system into stiff, f_I, and nonstiff, f_E, components
  - MRIStep: two-rate explicit-explicit multirate methods for $\dot{y} = f_f(t,y) + f_s(t,y), \quad y(t_0) = y_0$
    - Split the system into fast and slow components
    - More methods to come very soon

ARKode includes explicit, implicit, and additive Runge-Kutta methods

- Variable step size additive Runge-Kutta (RK) Methods – combine explicit (ERK) and diagonally implicit (DIRK) methods to enable IMEX solver
- Solve for each stage solution, z_i, sequentially then compute the time-evolved solution, y_n

$$z_i = y_{n-1} + \Delta t_n \sum_{j=1}^{i-1} a^E_{i,j} f_E(t^E_{n,j}, z_j) + \Delta t_n \sum_{j=1}^{i} a^I_{i,j} f_I(t^I_{n,j}, z_j) \quad i = 1, \ldots, s,$$

$$y_n = y_{n-1} + \Delta t_n \sum_{i=1}^{s} b^E_i f_E(t^E_{n,i}, z_i) + b^I_i f_I(t^I_{n,i}, z_i),$$

- Choose time steps based on error estimates
- ARKode provides methods of the following orders:
  - ARK: $O(\Delta t^3) - O(\Delta t^5)$
  - DIRK: $O(\Delta t^2) - O(\Delta t^5)$
  - ERK: $O(\Delta t^2) - O(\Delta t^8)$
  - Users can supply their own Butcher tables

Time steps are chosen to minimize local truncation error and maximize efficiency

- Time step selection
  - Based on the method, estimate the time step error
  - Accept step if $||E(\Delta t)||_{WRMS} < 1$; Reject it otherwise

$$\|y\|_{\mathrm{wrms}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(w_i\ y_i)^2} \qquad w_i = \frac{1}{RTOL|y_i| + ATOL_i}$$

  - Choose next step, Δt’, so that $||E(\Delta t')||_{\ WRMS} < 1$
- CVODE and IDA also adapt order
  - Choose next order resulting in largest step meeting error condition

- Relative tolerance (RTOL) controls local error relative to the size of the solution
  - RTOL = 10^{-4} means that errors are controlled to 0.01%
- Absolute tolerances (ATOL) control error when a solution component may be small
  - Ex: solution starting at a nonzero value but decaying to noise level, ATOL should be set to noise level

KINSOL solves systems of nonlinear algebraic equations, F(u) = 0

- Newton solvers: update iterate via $u^{k+1} = u^k + s^k, k = 0, ..., 1$
  - Compute the update by solving: $J(u^k)s^k = -F(u^k) \quad J(u) = \dfrac{\partial F(u)}{\partial u}$
  - An inexact Newton method approximately solves this equation
- Dynamic linear tolerance selection for use with iterative linear solvers

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

- Can separately scale equations and unknowns
- Backtracking and line search options for robustness
- Fixed point and Picard iterations with optional Anderson acceleration are also available

$$u^{k+1} = G(u^k), k = 0, 1, ...$$

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

SUNDIALS uses Control Inversion to interoperate with other solvers and applications

Use case:

- Implicit integration
- Iterative linear solver
- Finite element (FEM) application

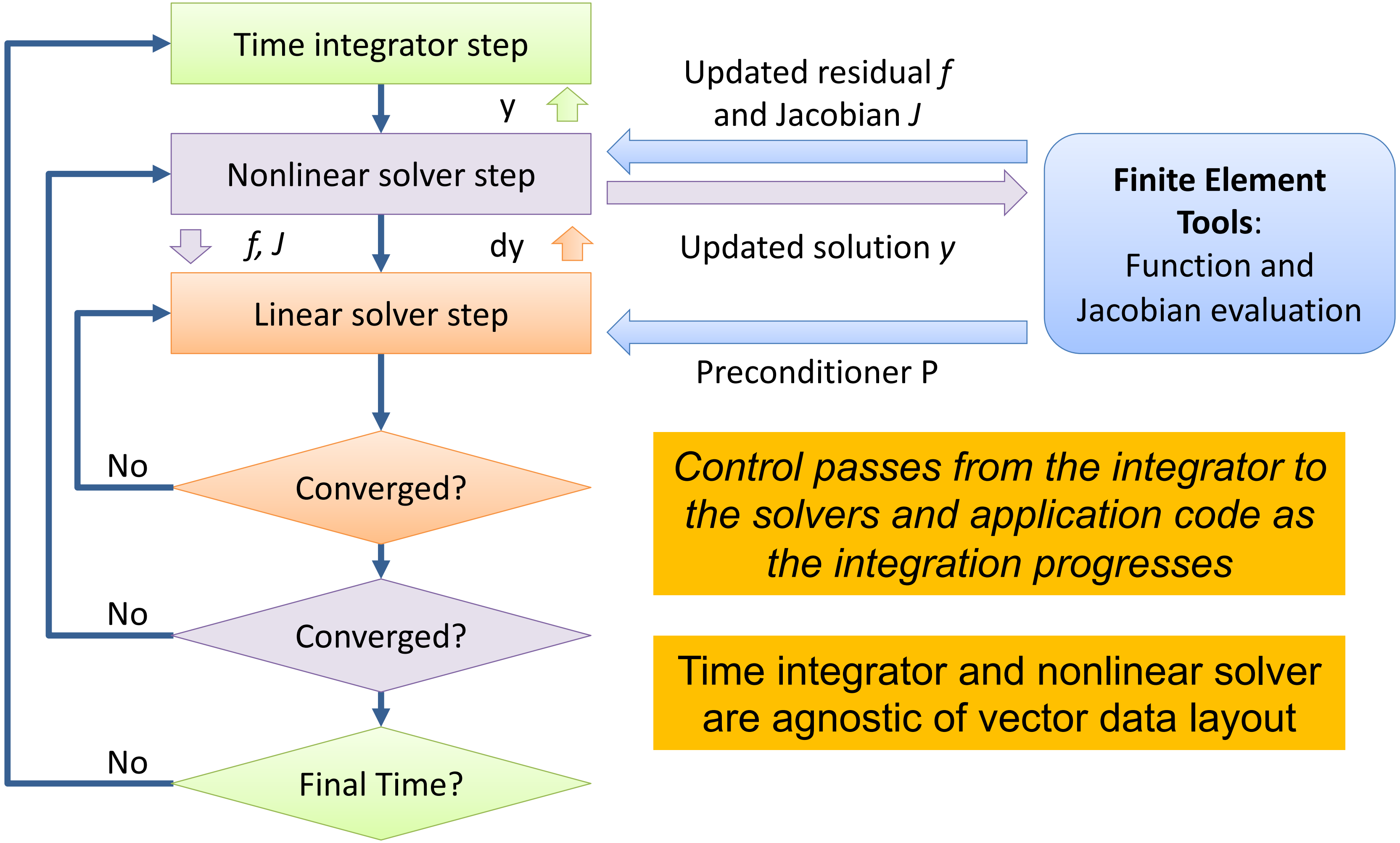
SUNDIALS

Nonlinear Solver

Linear solver

Application code

Numerical integrators and nonlinear solvers may invoke fairly complex step size control logic



Time integrator step

y

Nonlinear solver step

f, *J*

dy

Linear solver step

Converged?

No

Converged?

No

Final Time?

No

Updated residual *f* and Jacobian *J*

Updated solution *y*

Preconditioner P

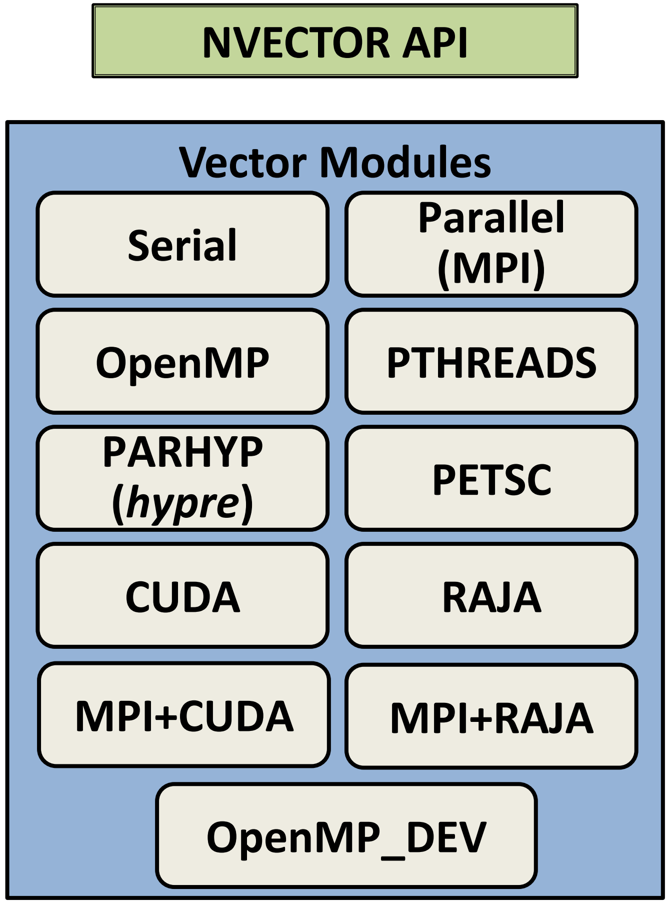
Finite Element Tools: Function and Jacobian evaluation

Control passes from the integrator to the solvers and application code as the integration progresses

Time integrator and nonlinear solver are agnostic of vector data layout

The SUNDIALS vector interface encapsulates interaction with application data

- SUNDIALS' integrators do not directly modify solution data; this is modified through vector operations e.g., vector adds, norms, etc.
- Vector "class" includes `content` and `ops` structures
  - `content` contains vector data and information on its layout, stored as a `(void *)` pointer
  - `Ops` includes all the operations SUNDIALS needs on a vector; functions are pointers stored in the vector `Ops` structure
- The `NVector` API defines the needed vector operations
- Parallelism is reflected in the vector structure, not in SUNDIALS
- Vectors should match the problem and/or algebraic solvers
- It is straightforward to implement a problem-specific vector interface tailored to the application



SUNDIALS is released with numerous optional vectors

Sensitivity Analysis: CVODES and IDAS

- Sensitivity Analysis (SA) is the study of how the variation 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 SA** – augment state system with sensitivity equations
  - **Adjoint SA** – solve a backward in time adjoint problem (user supplies the adjoint problem)

Forward sensitivity analysis results in additional sensitivity equations to integrate with the original state equation

- For a parameter dependent ODE (left) or DAE (right) system:

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

$$F(t, y, \dot{y}, p) = 0, \quad y(t_0) = y_0(p), \quad \dot{y}(t_0) = \dot{y}_0(p)$$

Find $s_i = dy/dp_i$ by simultaneously solving the original system with the N_p sensitivity systems obtained by differentiating the original system with respect to each parameter in turn:

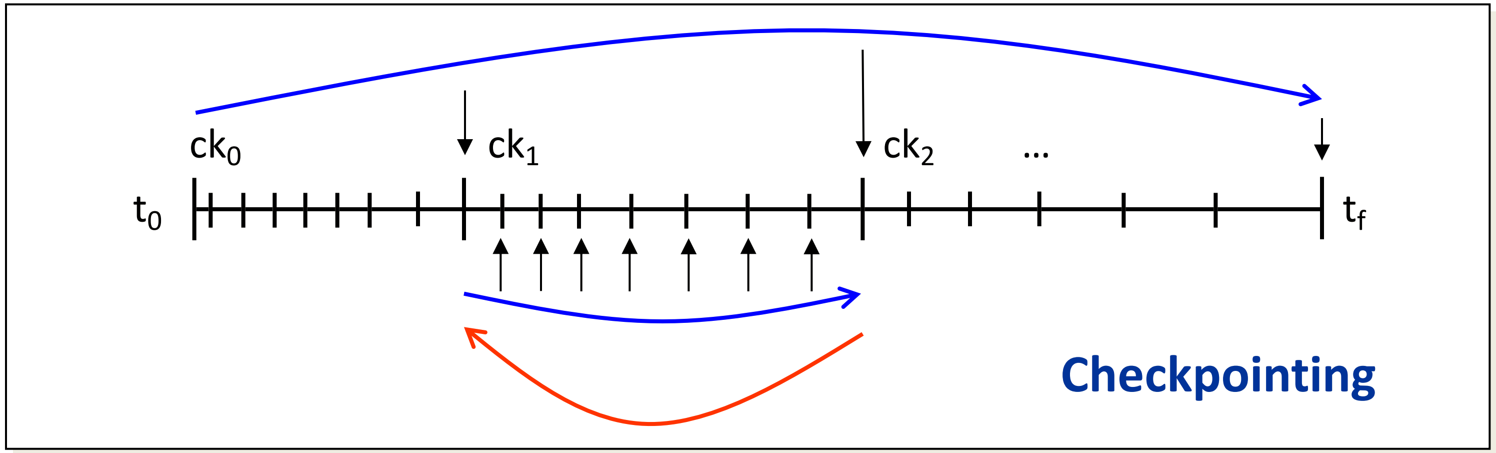
$$\dot{s}_i = \frac{\partial f}{\partial y} s_i + \frac{\partial f}{\partial p_i}$$

$$\frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial \dot{y}} \dot{s}_i + \frac{\partial F}{\partial p_i} = 0$$

- CVODES and IDAS include two methods for defining the forward sensitivity systems:
  - **Simultaneous Corrector Method:** On each time step, solve the nonlinear system simultaneously for solution and sensitivity variables
  - **Staggered Corrector Method:** On each time step, converge the nonlinear system for state variables, then iterate to solve sensitivity system

SUNDIALS supports the backward in time integration needed for adjoint sensitivity analysis

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



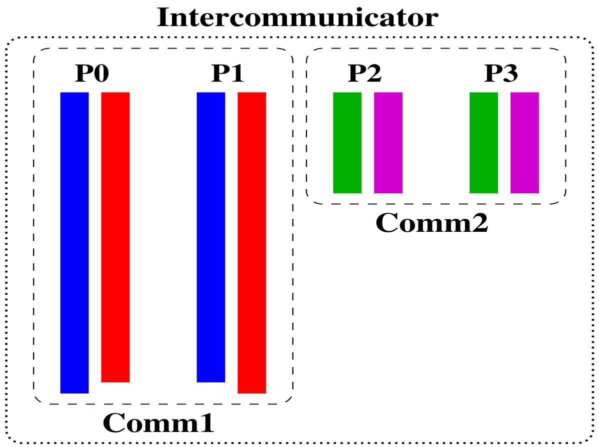
- Simulations are reproducible from each checkpoint
- Cubic Hermite or variable-degree polynomial interpolation
- Store solution and first derivative at each checkpoint
- Force Jacobian evaluation at checkpoints to avoid storing it
- Computational cost: 2 forward and 1 backward integrations

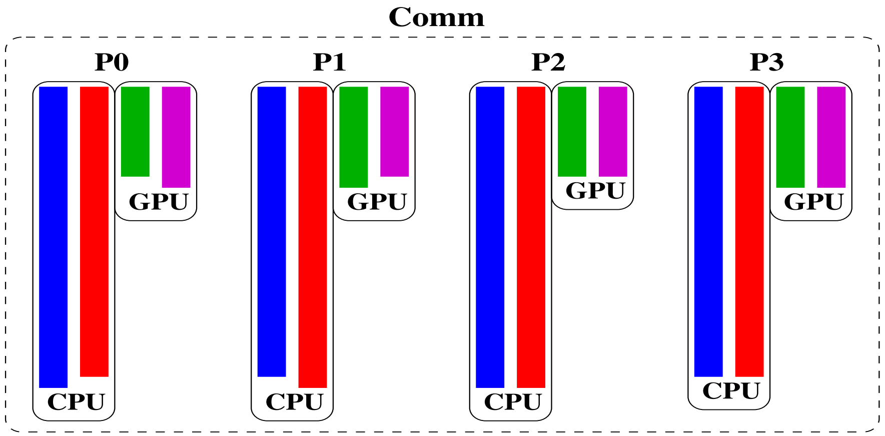
What's new in SUNDIALS?

- High-order multirate methods that can integrate different portions of the problem with different time steps - current release includes a 3rd order two-rate explicit method
- New vector modules: MPI+CUDA, MPI+RAJA, and OpenMPDEV (OpenMP 4.5+)
- Encapsulated nonlinear solvers
- Fortran 2003 interface (modernized from original F77 interface) for CVODE and all linear solvers (IDA, ARKode, and KINSOL interfaces coming soon)
- Fused vector operations increase data reuse, decrease the number of vector operation calls, and reduce parallel communication

What are we working on now?

- Many-vector capability for SUNDIALS will make use of heterogeneous architectures and development of methods for multiphysics systems easier





Left: Multiphysics many-vector, different physics operate on different processes and comms coupled with an MPI intercommunicator

Right: Data partitioning many-vector, each vector utilizes distinct processing elements within the same node

- Increased interoperability with other solver libraries

SuperLU_DIST TRILINOS PETSc MAGMA

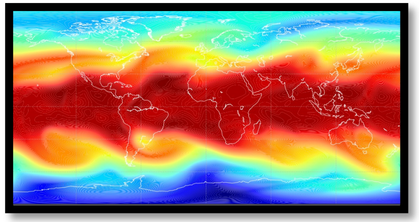
- More multirate methods, including implicit / explicit schemes

SUNDIALS: Used Worldwide in Applications from Research & Industry

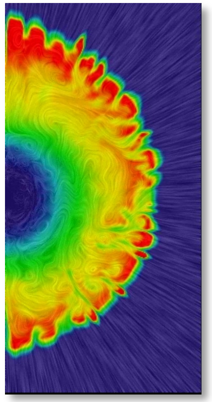
- Computational Cosmology (Nyx)
- Combustion (PELE)
- Astrophysics (CASTRO)

Used as a combustion integrator through AMReX

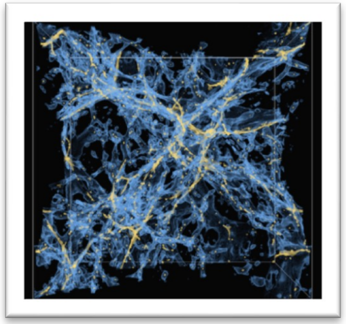
- Atmospheric dynamics (DOE E3SM)
- Fluid Dynamics (NEK5000) (ANL)
- Dislocation dynamics (LLNL)
- 3D parallel fusion (SMU, U. York, LLNL)
- Power grid modeling (RTE France, ISU, LLNL)
- Sensitivity analysis of chemically reacting flows (Sandia)
- Large-scale subsurface flows (CO Mines, LLNL)
- Micromagnetic simulations (U. Southampton)
- Chemical kinetics (Cantera)
- Released in third party packages:
  - Red Hat Extra Packages for Enterprise Linux (EPEL)
  - SciPy – python wrap of SUNDIALS
  - Cray Third Party Software Library (TPSL)



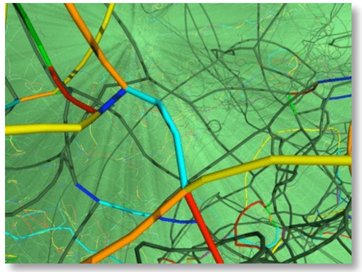
Atmospheric Dynamics



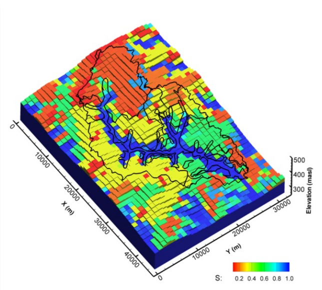
Core collapse supernova



Cosmology



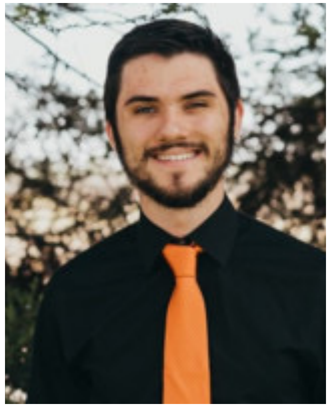
Dislocation dynamics



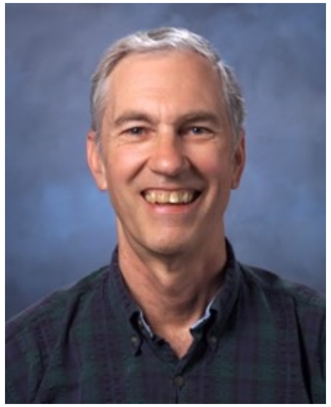
Subsurface flow

SUNDIALS Team

Current Team:


Cody Balos

David Gardner


Alan Hindmarsh

Slaven Peles

Dan Reynolds

Carol Woodward

Alumni:

Radu Serban

Scott Cohen, Scott Cohen, Peter N. Brown, George Byrne, Allan G. Taylor, Steven L. Lee, Keith E. Grant, Aaron Collier, Lawrence E. Banks, Steve Smith, Cosmin Petra, John Loffeld, Dan Shumaker, Ulrike Yang, James Almgren-Bell, Shelby Lockhart, Hilari Tiedeman, Ting Yan, Jean Sexton, and Chris White

Tutorial Outline

- Overview of SUNDIALS (Carol Woodward)
- **How to download and install SUNDIALS (Cody Balos)**
- How to use the time integrators (Daniel Reynolds)
- Which nonlinear and linear solvers are available and how to use them (David Gardner)

Acquiring SUNDIALS

- Download the tarball from the SUNDIALS website
  - https://computation.llnl.gov/projects/sundials/sundials-software
  - Latest (v4.0.1) and archived versions, and individual packages (e.g., CVODE) available
  - Most configurable

- Download the tarball from the SUNDIALS GitHub page
  - https://github.com/LLNL/sundials/releases
  - Latest and archived versions available
  - Most configurable

- Install SUNDIALS using Spack
  - “spack install sundials”
  - Latest and recent versions available
  - Highly configurable via spack variants. E.g., “spack install sundials+cuda”.

- Install SUNDIALS as part of the xSDK using Spack
  - “spack install xsdk”
  - Will install SUNDIALS v3.2.1

Preparing to Build and Install SUNDIALS from Source

- Download a SUNDIALS tarball and extract it: `tar -xzf` *`package`*`-x.y.z.tar.gz`
  - Where *package* is one of: sundials, cvode, cvodes, arkode, ida, idas, or kinsol
  - Where *x.y.z* is the package version number
  - The compressed files will be extracted to the directory *`package`*`-x.y.z`

- For the remainder of the tutorial the following conventions will be followed:
  - ***packagedir*** will refer to the *`package`*`-x.y.z` directory
  - ***builddir*** will refer to the temporary directory where SUNDIALS is built. This directory cannot be the same as *packagedir.*
  - ***instdir*** will refer to the directory where SUNDIALS exported header files and libraries will be installed. This defaults to `/usr/local` on Unix systems or `C:\Program Files` on Windows. This directory cannot be the same as *packagedir.*

- It is required that *builddir* exists before proceeding with the build process

- Building SUNDIALS minimally requires CMake 3.1.3 or greater and a working C compiler
  - Depending on desired options more requirements are imposed

Building and Installing from Source using Defaults

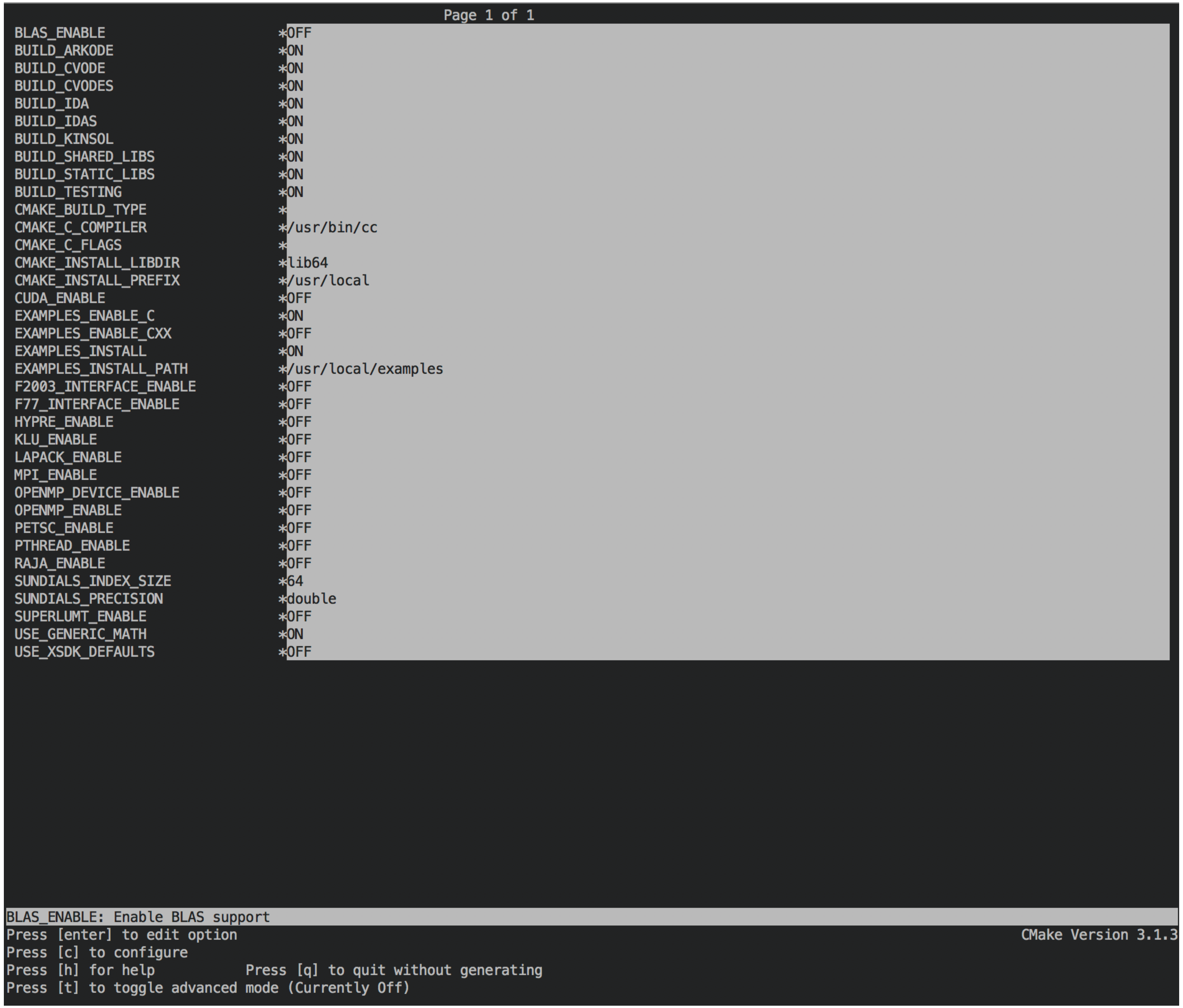
- In the next few steps, we will use the CMake curses GUI (`ccmake`) to configure SUNDIALS, however, the CMake command-line interface (`cmake`) or the more interactive CMake Qt GUI can also be utilized to obtain the same result.

1. To begin the build process, navigate to *builddir* and execute the command:

```
% ccmake packagedir
```

2. The CMake GUI will appear empty

3. Press ‘c’ to continue to the default SUNDIALS configuration screen



```
                                                   Page 1 of 1
 BLAS_ENABLE                      *OFF
 BUILD_ARKODE                     *ON
 BUILD_CVODE                      *ON
 BUILD_CVODES                     *ON
 BUILD_IDA                        *ON
 BUILD_IDAS                       *ON
 BUILD_KINSOL                     *ON
 BUILD_SHARED_LIBS                *ON
 BUILD_STATIC_LIBS                *ON
 BUILD_TESTING                    *ON
 CMAKE_BUILD_TYPE                 *
 CMAKE_C_COMPILER                 */usr/bin/cc
 CMAKE_C_FLAGS                    *
 CMAKE_INSTALL_LIBDIR             *lib64
 CMAKE_INSTALL_PREFIX             */usr/local
 CUDA_ENABLE                      *OFF
 EXAMPLES_ENABLE_C                *ON
 EXAMPLES_ENABLE_CXX              *OFF
 EXAMPLES_INSTALL                 *ON
 EXAMPLES_INSTALL_PATH            */usr/local/examples
 F2003_INTERFACE_ENABLE           *OFF
 F77_INTERFACE_ENABLE             *OFF
 HYPRE_ENABLE                     *OFF
 KLU_ENABLE                       *OFF
 LAPACK_ENABLE                    *OFF
 MPI_ENABLE                       *OFF
 OPENMP_DEVICE_ENABLE             *OFF
 OPENMP_ENABLE                    *OFF
 PETSC_ENABLE                     *OFF
 PTHREAD_ENABLE                   *OFF
 RAJA_ENABLE                      *OFF
 SUNDIALS_INDEX_SIZE              *64
 SUNDIALS_PRECISION               *double
 SUPERLUMT_ENABLE                 *OFF
 USE_GENERIC_MATH                 *ON
 USE_XSDK_DEFAULTS                *OFF

BLAS_ENABLE: Enable BLAS support
Press [enter] to edit option                                                                            CMake Version 3.1.3
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Building and Installing from Source using Defaults

1. To begin the build process, navigate to *builddir* and execute the command:

```
% ccmake packagedir
```

2. The CMake GUI will appear empty

3. Press ‘c’ to continue to the default SUNDIALS configuration screen

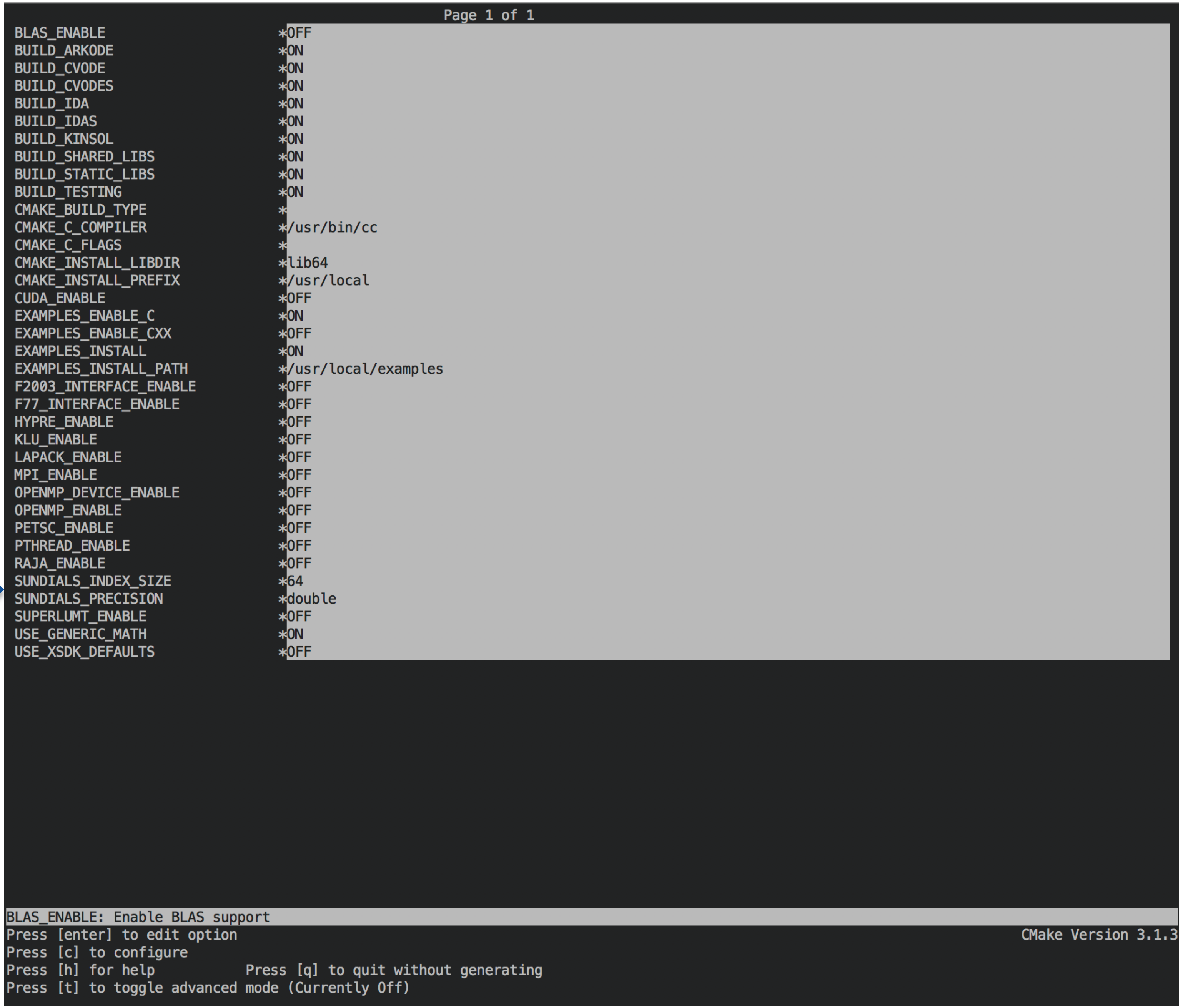
4. **To build SUNDIALS with the default settings press ‘c’ again followed by ‘g’ to generate**

5. The CMake GUI will now be closed and the build process can be completed using `make`:

```
    % make

    % make install
```

Building and Installing from Source with Non-Defaults

- SUNDIALS has many configuration options to allow for highly customized builds
- Notably:
  - `CMAKE_INSTALL_PREFIX` and `CMAKE_INSTALL_LIBDIR` options can be used to set the directory where SUNDIALS will be installed
  - `SUNDIALS_INDEX_SIZE` can be used to configure SUNDIALS for 32-bit or 64-bit indexing
    - Sets the SUNDIALS type, `sunindextype`, to the configured size
  - `SUNDIALS_PRECISION` can be used to configure SUNDIALS for single, double, or extended precision
    - Sets the SUNDIALS type, `realtype`, to the precision configured

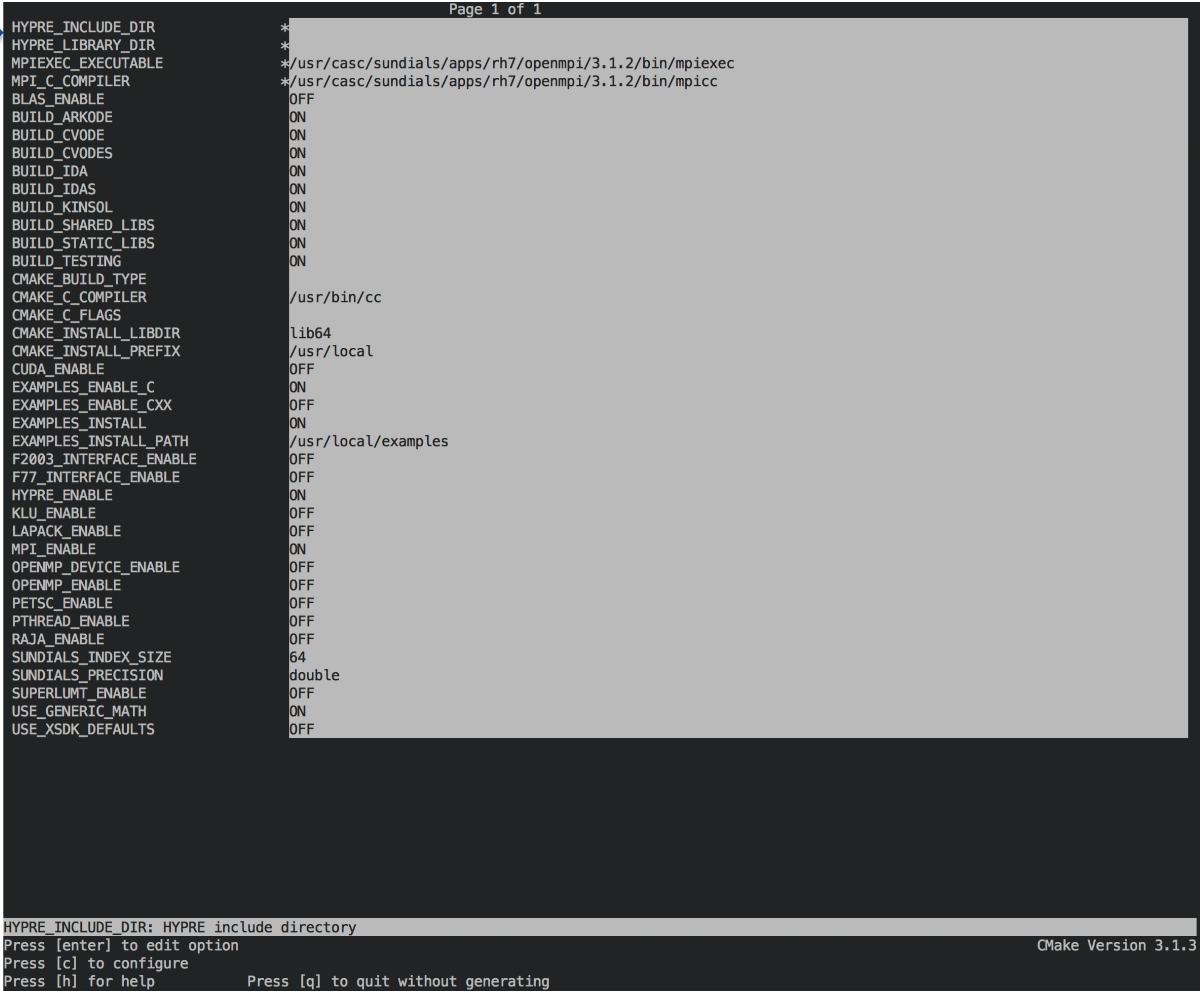


```
                                    Page 1 of 1
 BLAS_ENABLE                      *OFF
 BUILD_ARKODE                     *ON
 BUILD_CVODE                      *ON
 BUILD_CVODES                     *ON
 BUILD_IDA                        *ON
 BUILD_IDAS                       *ON
 BUILD_KINSOL                     *ON
 BUILD_SHARED_LIBS                *ON
 BUILD_STATIC_LIBS                *ON
 BUILD_TESTING                    *ON
 CMAKE_BUILD_TYPE                 *
 CMAKE_C_COMPILER                 */usr/bin/cc
 CMAKE_C_FLAGS                    *
 CMAKE_INSTALL_LIBDIR             *lib64
 CMAKE_INSTALL_PREFIX             */usr/local
 CUDA_ENABLE                      *OFF
 EXAMPLES_ENABLE_C                *ON
 EXAMPLES_ENABLE_CXX              *OFF
 EXAMPLES_INSTALL                 *ON
 EXAMPLES_INSTALL_PATH            */usr/local/examples
 F2003_INTERFACE_ENABLE           *OFF
 F77_INTERFACE_ENABLE             *OFF
 HYPRE_ENABLE                     *OFF
 KLU_ENABLE                       *OFF
 LAPACK_ENABLE                    *OFF
 MPI_ENABLE                       *OFF
 OPENMP_DEVICE_ENABLE             *OFF
 OPENMP_ENABLE                    *OFF
 PETSC_ENABLE                     *OFF
 PTHREAD_ENABLE                   *OFF
 RAJA_ENABLE                      *OFF
 SUNDIALS_INDEX_SIZE              *64
 SUNDIALS_PRECISION               *double
 SUPERLUMT_ENABLE                 *OFF
 USE_GENERIC_MATH                 *ON
 USE_XSDK_DEFAULTS                *OFF




BLAS_ENABLE: Enable BLAS support
Press [enter] to edit option                                       CMake Version 3.1.3
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Building and Installing from Source with Non-Defaults: Example

Let’s enable the MPI SUNDIALS modules and SUNDIALS interfaces to *hypre*:

1. From the *builddir* open up the CMake curses GUI (`ccmake`)
2. Use the arrow keys to navigate to the option `MPI_ENABLE`
3. Press the ‘enter’ key to toggle the option to “ON”
4. Similarly toggle the option `HYPRE_ENABLE` to “ON”
5. Press ‘c’ to configure



```
                                       Page 1 of 1
HYPRE_INCLUDE_DIR                *
HYPRE_LIBRARY_DIR                *
MPIEXEC_EXECUTABLE               */usr/casc/sundials/apps/rh7/openmpi/3.1.2/bin/mpiexec
MPI_C_COMPILER                   */usr/casc/sundials/apps/rh7/openmpi/3.1.2/bin/mpicc
BLAS_ENABLE                       OFF
BUILD_ARKODE                      ON
BUILD_CVODE                       ON
BUILD_CVODES                      ON
BUILD_IDA                         ON
BUILD_IDAS                        ON
BUILD_KINSOL                      ON
BUILD_SHARED_LIBS                 ON
BUILD_STATIC_LIBS                 ON
BUILD_TESTING                     ON
CMAKE_BUILD_TYPE
CMAKE_C_COMPILER                  /usr/bin/cc
CMAKE_C_FLAGS
CMAKE_INSTALL_LIBDIR              lib64
CMAKE_INSTALL_PREFIX              /usr/local
CUDA_ENABLE                       OFF
EXAMPLES_ENABLE_C                 ON
EXAMPLES_ENABLE_CXX               OFF
EXAMPLES_INSTALL                  ON
EXAMPLES_INSTALL_PATH             /usr/local/examples
F2003_INTERFACE_ENABLE            OFF
F77_INTERFACE_ENABLE              OFF
HYPRE_ENABLE                      ON
KLU_ENABLE                        OFF
LAPACK_ENABLE                     OFF
MPI_ENABLE                        ON
OPENMP_DEVICE_ENABLE              OFF
OPENMP_ENABLE                     OFF
PETSC_ENABLE                      OFF
PTHREAD_ENABLE                    OFF
RAJA_ENABLE                       OFF
SUNDIALS_INDEX_SIZE               64
SUNDIALS_PRECISION                double
SUPERLUMT_ENABLE                  OFF
USE_GENERIC_MATH                  ON
USE_XSDK_DEFAULTS                 OFF

HYPRE_INCLUDE_DIR: HYPRE include directory
Press [enter] to edit option                                                 CMake Version 3.1.3
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
```

Building and Installing from Source using Non-Defaults

Let’s enable the MPI SUNDIALS modules and SUNDIALS interfaces to *hypre*:

1. From the *builddir* open up the Cmake curses GUI (`ccmake`)
2. Use the arrow keys to navigate to the option `ENABLE`
3. Press the ‘enter’ key to toggle the option to “ON”
4. Similarly toggle the option `HYPRE_ENABLE` to “ON”
5. Press ‘c’ to configure
6. **Use the arrow keys to navigate to the option `HYPRE_INCLUDE_DIR` and press ‘enter’ to set the path to the include directory of the desired HYPRE installation**
7. Press ‘enter’ again to finish editing the `HYPRE_INCLUDE_DIR` option
8. Similarly, set the `HYPRE_LIBRARY_DIR` option
9. Press ‘c’ to configure followed by ‘g’ to generate
10. The CMake GUI will now be closed and the build process can be completed using `make`:

```
% make

% make install
```

CMake CLI Equivalents

- The CMake command line interface can be used to generate the same builds of SUNDIALS as the CMake curses GUI

- The command line interface is convenient for scripting a SUNDIALS build

- To build SUNDIALS with the default options:
  1. Navigate to *builddir* and run: `% cmake` *`packagedir`*
  2. Complete the build process by running: `% make && make install`

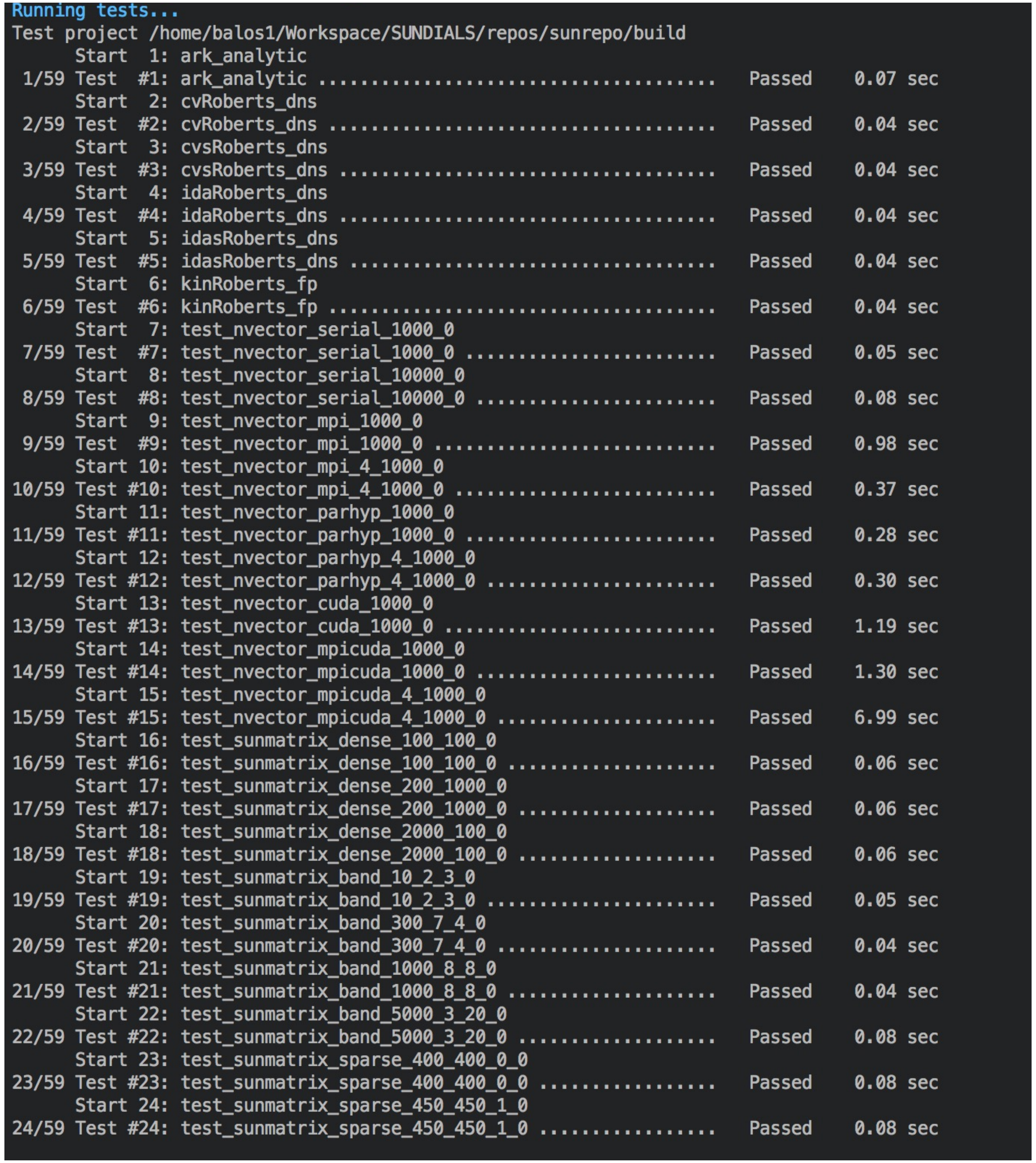
- To build SUNDIALS with MPI and *hypre* enabled:
  1. Navigate to *builddir* and run:

```
% cmake -DMPI_ENABLE=ON -DHYPRE_ENABLE=ON \
% -DHYPRE_INCLUDE_DIR=<hypre include directory> \
% -DHYPRE_LIBRARY_DIR=<hypre library directory> packagedir
```

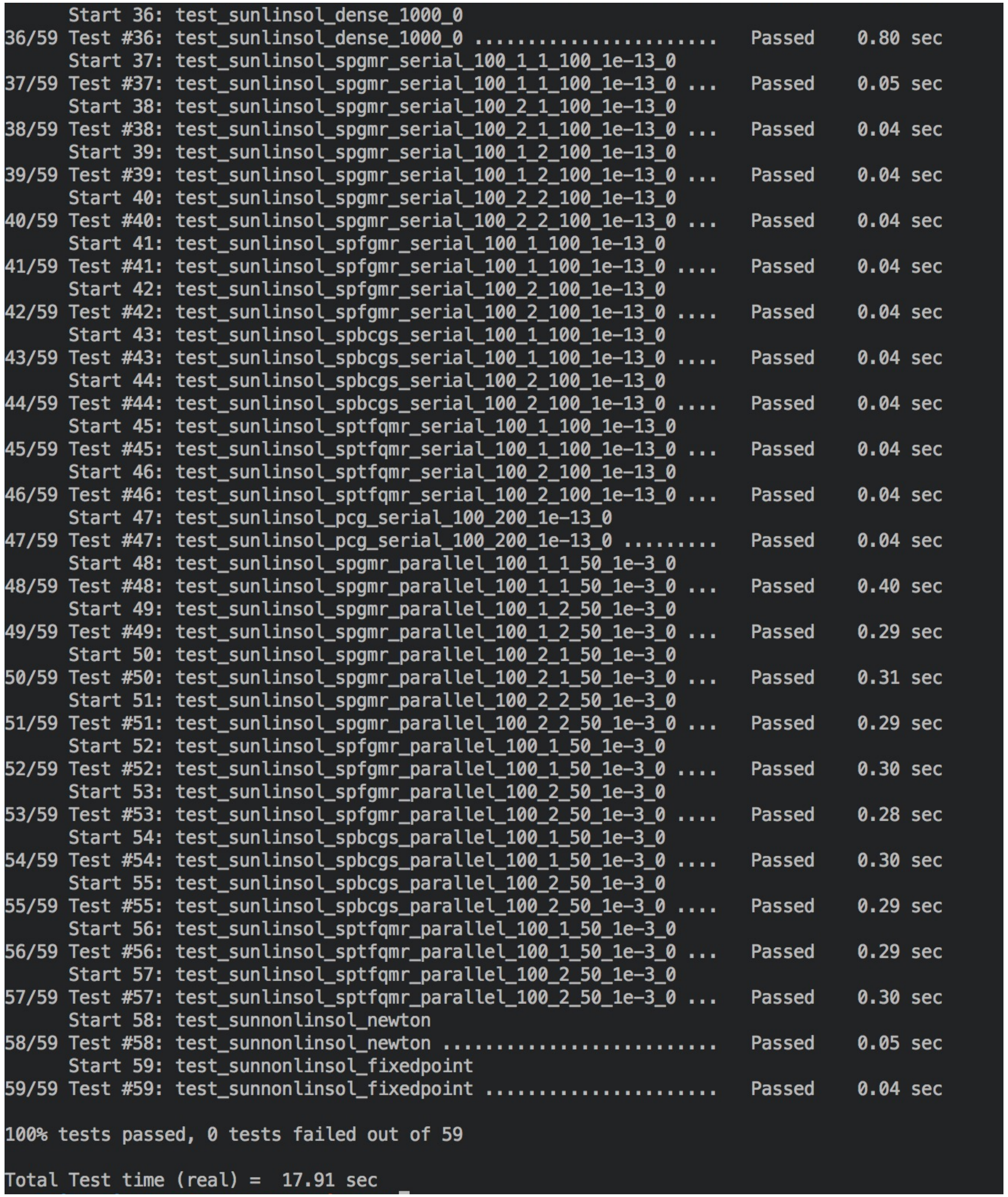
  2. Complete the build process by running: `% make && make install`

Verifying a SUNDIALS Build

- After building SUNDIALS, it is a good practice to verify that the SUNDIALS build is functional
- From *builddir,* a user can execute the command `make test` to run the short SUNDIALS test suite
  - Requires CTest and Python version 2.7 or greater
- Details about failed tests can be found in the directories `builddir/Testing/output` and `builddir/Testing/Temporary`



```
Running tests...
Test project /home/balos1/Workspace/SUNDIALS/repos/sunrepo/build
      Start  1: ark_analytic
 1/59 Test  #1: ark_analytic ......................................   Passed    0.07 sec
      Start  2: cvRoberts_dns
 2/59 Test  #2: cvRoberts_dns .....................................   Passed    0.04 sec
      Start  3: cvsRoberts_dns
 3/59 Test  #3: cvsRoberts_dns ....................................   Passed    0.04 sec
      Start  4: idaRoberts_dns
 4/59 Test  #4: idaRoberts_dns ....................................   Passed    0.04 sec
      Start  5: idasRoberts_dns
 5/59 Test  #5: idasRoberts_dns ...................................   Passed    0.04 sec
      Start  6: kinRoberts_fp
 6/59 Test  #6: kinRoberts_fp .....................................   Passed    0.04 sec
      Start  7: test_nvector_serial_1000_0
 7/59 Test  #7: test_nvector_serial_1000_0 ........................   Passed    0.05 sec
      Start  8: test_nvector_serial_10000_0
 8/59 Test  #8: test_nvector_serial_10000_0 .......................   Passed    0.08 sec
      Start  9: test_nvector_mpi_1000_0
 9/59 Test  #9: test_nvector_mpi_1000_0 ...........................   Passed    0.98 sec
      Start 10: test_nvector_mpi_4_1000_0
10/59 Test #10: test_nvector_mpi_4_1000_0 .........................   Passed    0.37 sec
      Start 11: test_nvector_parhyp_1000_0
11/59 Test #11: test_nvector_parhyp_1000_0 ........................   Passed    0.28 sec
      Start 12: test_nvector_parhyp_4_1000_0
12/59 Test #12: test_nvector_parhyp_4_1000_0 ......................   Passed    0.30 sec
      Start 13: test_nvector_cuda_1000_0
13/59 Test #13: test_nvector_cuda_1000_0 ..........................   Passed    1.19 sec
      Start 14: test_nvector_mpicuda_1000_0
14/59 Test #14: test_nvector_mpicuda_1000_0 .......................   Passed    1.30 sec
      Start 15: test_nvector_mpicuda_4_1000_0
15/59 Test #15: test_nvector_mpicuda_4_1000_0 .....................   Passed    6.99 sec
      Start 16: test_sunmatrix_dense_100_100_0
16/59 Test #16: test_sunmatrix_dense_100_100_0 ....................   Passed    0.06 sec
      Start 17: test_sunmatrix_dense_200_1000_0
17/59 Test #17: test_sunmatrix_dense_200_1000_0 ...................   Passed    0.06 sec
      Start 18: test_sunmatrix_dense_2000_100_0
18/59 Test #18: test_sunmatrix_dense_2000_100_0 ...................   Passed    0.06 sec
      Start 19: test_sunmatrix_band_10_2_3_0
19/59 Test #19: test_sunmatrix_band_10_2_3_0 ......................   Passed    0.05 sec
      Start 20: test_sunmatrix_band_300_7_4_0
20/59 Test #20: test_sunmatrix_band_300_7_4_0 .....................   Passed    0.04 sec
      Start 21: test_sunmatrix_band_1000_8_8_0
21/59 Test #21: test_sunmatrix_band_1000_8_8_0 ....................   Passed    0.04 sec
      Start 22: test_sunmatrix_band_5000_3_20_0
22/59 Test #22: test_sunmatrix_band_5000_3_20_0 ...................   Passed    0.08 sec
      Start 23: test_sunmatrix_sparse_400_400_0_0
23/59 Test #23: test_sunmatrix_sparse_400_400_0_0 .................   Passed    0.08 sec
      Start 24: test_sunmatrix_sparse_450_450_1_0
24/59 Test #24: test_sunmatrix_sparse_450_450_1_0 .................   Passed    0.08 sec
```

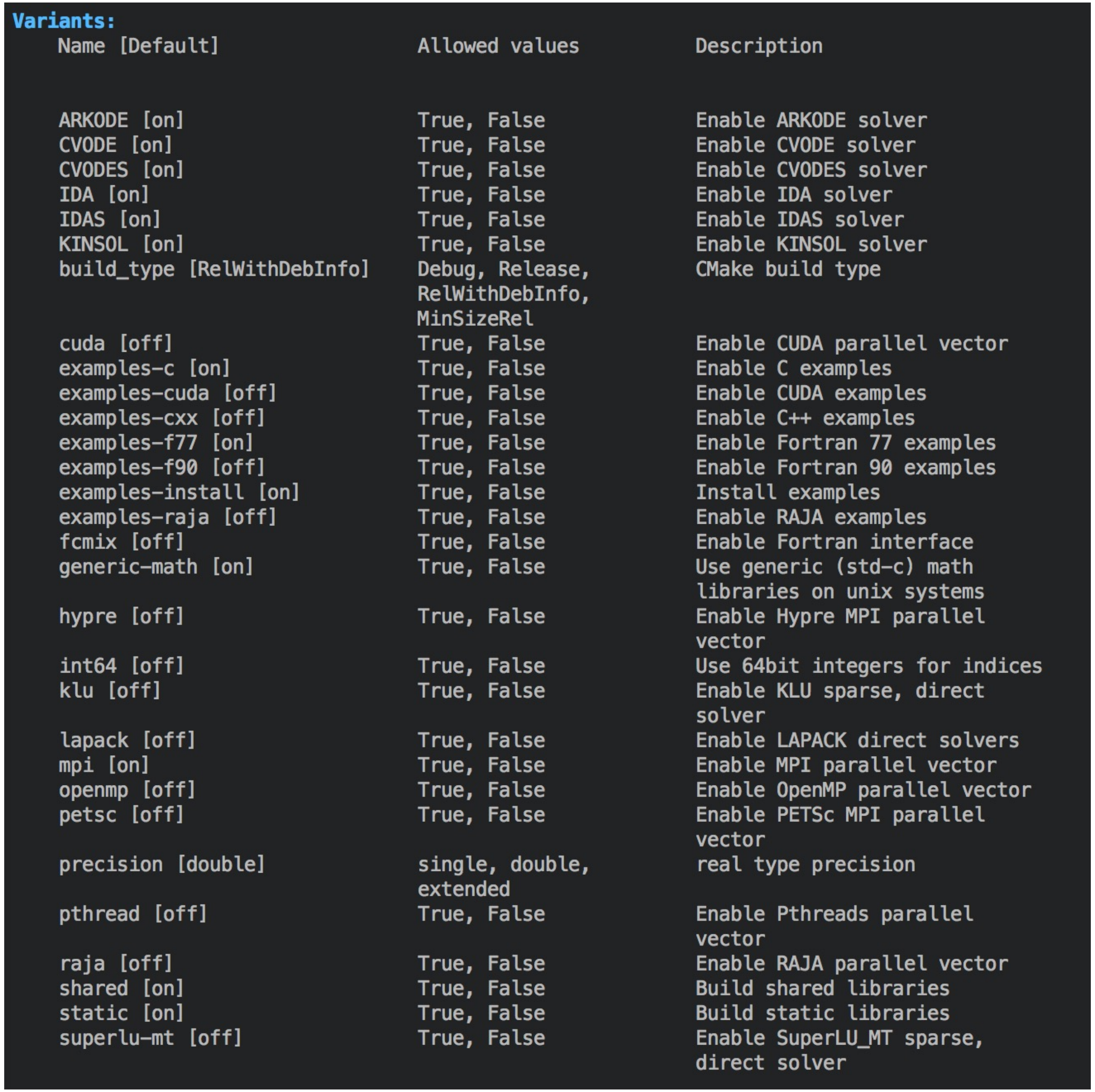


```
      Start 36: test_sunlinsol_dense_1000_0
36/59 Test #36: test_sunlinsol_dense_1000_0 .......................   Passed    0.80 sec
      Start 37: test_sunlinsol_spgmr_serial_100_1_1_100_1e-13_0
37/59 Test #37: test_sunlinsol_spgmr_serial_100_1_1_100_1e-13_0 ...   Passed    0.05 sec
      Start 38: test_sunlinsol_spgmr_serial_100_2_1_100_1e-13_0
38/59 Test #38: test_sunlinsol_spgmr_serial_100_2_1_100_1e-13_0 ...   Passed    0.04 sec
      Start 39: test_sunlinsol_spgmr_serial_100_1_2_100_1e-13_0
39/59 Test #39: test_sunlinsol_spgmr_serial_100_1_2_100_1e-13_0 ...   Passed    0.04 sec
      Start 40: test_sunlinsol_spgmr_serial_100_2_2_100_1e-13_0
40/59 Test #40: test_sunlinsol_spgmr_serial_100_2_2_100_1e-13_0 ...   Passed    0.04 sec
      Start 41: test_sunlinsol_spfgmr_serial_100_1_100_1e-13_0
41/59 Test #41: test_sunlinsol_spfgmr_serial_100_1_100_1e-13_0 ....   Passed    0.04 sec
      Start 42: test_sunlinsol_spfgmr_serial_100_2_100_1e-13_0
42/59 Test #42: test_sunlinsol_spfgmr_serial_100_2_100_1e-13_0 ....   Passed    0.04 sec
      Start 43: test_sunlinsol_spbcgs_serial_100_1_100_1e-13_0
43/59 Test #43: test_sunlinsol_spbcgs_serial_100_1_100_1e-13_0 ....   Passed    0.04 sec
      Start 44: test_sunlinsol_spbcgs_serial_100_2_100_1e-13_0
44/59 Test #44: test_sunlinsol_spbcgs_serial_100_2_100_1e-13_0 ....   Passed    0.04 sec
      Start 45: test_sunlinsol_sptfqmr_serial_100_1_100_1e-13_0
45/59 Test #45: test_sunlinsol_sptfqmr_serial_100_1_100_1e-13_0 ...   Passed    0.04 sec
      Start 46: test_sunlinsol_sptfqmr_serial_100_2_100_1e-13_0
46/59 Test #46: test_sunlinsol_sptfqmr_serial_100_2_100_1e-13_0 ...   Passed    0.04 sec
      Start 47: test_sunlinsol_pcg_serial_100_200_1e-13_0
47/59 Test #47: test_sunlinsol_pcg_serial_100_200_1e-13_0 .........   Passed    0.04 sec
      Start 48: test_sunlinsol_spgmr_parallel_100_1_1_50_1e-3_0
48/59 Test #48: test_sunlinsol_spgmr_parallel_100_1_1_50_1e-3_0 ...   Passed    0.40 sec
      Start 49: test_sunlinsol_spgmr_parallel_100_1_2_50_1e-3_0
49/59 Test #49: test_sunlinsol_spgmr_parallel_100_1_2_50_1e-3_0 ...   Passed    0.29 sec
      Start 50: test_sunlinsol_spgmr_parallel_100_2_1_50_1e-3_0
50/59 Test #50: test_sunlinsol_spgmr_parallel_100_2_1_50_1e-3_0 ...   Passed    0.31 sec
      Start 51: test_sunlinsol_spgmr_parallel_100_2_2_50_1e-3_0
51/59 Test #51: test_sunlinsol_spgmr_parallel_100_2_2_50_1e-3_0 ...   Passed    0.29 sec
      Start 52: test_sunlinsol_spfgmr_parallel_100_1_50_1e-3_0
52/59 Test #52: test_sunlinsol_spfgmr_parallel_100_1_50_1e-3_0 ....   Passed    0.30 sec
      Start 53: test_sunlinsol_spfgmr_parallel_100_2_50_1e-3_0
53/59 Test #53: test_sunlinsol_spfgmr_parallel_100_2_50_1e-3_0 ....   Passed    0.28 sec
      Start 54: test_sunlinsol_spbcgs_parallel_100_1_50_1e-3_0
54/59 Test #54: test_sunlinsol_spbcgs_parallel_100_1_50_1e-3_0 ....   Passed    0.30 sec
      Start 55: test_sunlinsol_spbcgs_parallel_100_2_50_1e-3_0
55/59 Test #55: test_sunlinsol_spbcgs_parallel_100_2_50_1e-3_0 ....   Passed    0.29 sec
      Start 56: test_sunlinsol_sptfqmr_parallel_100_1_50_1e-3_0
56/59 Test #56: test_sunlinsol_sptfqmr_parallel_100_1_50_1e-3_0 ...   Passed    0.29 sec
      Start 57: test_sunlinsol_sptfqmr_parallel_100_2_50_1e-3_0
57/59 Test #57: test_sunlinsol_sptfqmr_parallel_100_2_50_1e-3_0 ...   Passed    0.30 sec
      Start 58: test_sunnonlinsol_newton
58/59 Test #58: test_sunnonlinsol_newton ..........................   Passed    0.05 sec
      Start 59: test_sunnonlinsol_fixedpoint
59/59 Test #59: test_sunnonlinsol_fixedpoint ......................   Passed    0.04 sec

100% tests passed, 0 tests failed out of 59

Total Test time (real) =  17.91 sec
```

Installing SUNDIALS with Spack

Spack

- Spack (see https://spack.io/) is another great way to install SUNDIALS
- The SUNDIALS team maintains a spack package that allows a user to easily install SUNDIALS with one command: `spack install sundials`
- The default configuration installed with `spack install sundials` depends on the environment
- Use the command `spack spec sundials` to see what SUNDIALS options `spack install sundials` will turn on
- The SUNDIALS spack installation is configured through spack “variants”
- Run `spack info sundials` to see the available “variants” of SUNDIALS



```
Variants:
    Name [Default]                 Allowed values          Description


    ARKODE [on]                    True, False             Enable ARKODE solver
    CVODE [on]                     True, False             Enable CVODE solver
    CVODES [on]                    True, False             Enable CVODES solver
    IDA [on]                       True, False             Enable IDA solver
    IDAS [on]                      True, False             Enable IDAS solver
    KINSOL [on]                    True, False             Enable KINSOL solver
    build_type [RelWithDebInfo]    Debug, Release,         CMake build type
                                   RelWithDebInfo,
                                   MinSizeRel
    cuda [off]                     True, False             Enable CUDA parallel vector
    examples-c [on]                True, False             Enable C examples
    examples-cuda [off]            True, False             Enable CUDA examples
    examples-cxx [off]             True, False             Enable C++ examples
    examples-f77 [on]              True, False             Enable Fortran 77 examples
    examples-f90 [off]             True, False             Enable Fortran 90 examples
    examples-install [on]          True, False             Install examples
    examples-raja [off]            True, False             Enable RAJA examples
    fcmix [off]                    True, False             Enable Fortran interface
    generic-math [on]              True, False             Use generic (std-c) math
                                                           libraries on unix systems
    hypre [off]                    True, False             Enable Hypre MPI parallel
                                                           vector
    int64 [off]                    True, False             Use 64bit integers for indices
    klu [off]                      True, False             Enable KLU sparse, direct
                                                           solver
    lapack [off]                   True, False             Enable LAPACK direct solvers
    mpi [on]                       True, False             Enable MPI parallel vector
    openmp [off]                   True, False             Enable OpenMP parallel vector
    petsc [off]                    True, False             Enable PETSc MPI parallel
                                                           vector
    precision [double]             single, double,         real type precision
                                   extended
    pthread [off]                  True, False             Enable Pthreads parallel
                                                           vector
    raja [off]                     True, False             Enable RAJA parallel vector
    shared [on]                    True, False             Build shared libraries
    static [on]                    True, False             Build static libraries
    superlu-mt [off]               True, False             Enable SuperLU_MT sparse,
                                                           direct solver
```

Installing SUNDIALS with Spack

Spack

- Spack (see https://spack.io/) is another great way to install SUNDIALS
- The SUNDIALS team maintains a spack package that allows a user to easily install SUNDIALS with one command: `spack install sundials`
- The default configuration installed with `spack install sundials` depends on the environment
- Use the command `spack spec sundials` to see what SUNDIALS options `spack install sundials` will turn on
- The SUNDIALS spack installation is configured through spack "variants"
- Run `spack info sundials` to see the available "variants" of SUNDIALS available
- **SUNDIALS with MPI and *hypre* enabled can be installed with the command:**

```
% spack install sundials+mpi+hypre
```

Installing SUNDIALS via the xSDK

- The Extreme-scale Scientific Software Development Kit (xSDK) provides a foundation for an extensible scientific software ecosystem
- As a member of the xSDK, SUNDIALS is installed with the xSDK Spack package

```
% spack install xsdk
```

- SUNDIALS v3.2.1 (v4.0.1 is the newest) is included in the latest xSDK release - v0.4.0
- The variant of SUNDIALS included in v0.4.0 of the xSDK utilizes the SUNDIALS spack package defaults with the following exceptions:
  - the index size is changed to 32-bits instead of 64-bits
  - *hypre* support is enabled
- See https://xsdk.info for more information about the xSDK and getting it installed

More Help Building and Installing SUNDIALS

- An in-depth guide on building and installing SUNDIALS is contained in the root of all SUNDIALS tarballs as INSTALL_GUIDE.pdf
- The guide details how to configure SUNDIALS with CMake as well as every possible SUNDIALS CMake option
- The guide can also be found in Appendix A of the user guide for any SUNDIALS package
- Users can also check the sundials-users email list archive at: http://sundials.2283335.n4.nabble.com
- Users can post queries to the sundials-users email list. For more info see: https://computation.llnl.gov/projects/sundials/support

Tutorial Outline

- Overview of SUNDIALS (Carol Woodward)
- How to download and install SUNDIALS (Cody Balos)
- **How to use the time integrators (Daniel Reynolds)**
- Which nonlinear and linear solvers are available and how to use them (David Gardner)

Time Integrators – Outline

- Basic usage of SUNDIALS integrators
- Supplying initial conditions – vectors
- Supplying the initial-value problem – RHS and residual functions
- Integrator initialization and optional inputs
- Advancing the solutions
- Retrieving optional outputs
- Advanced features

“Solving” Initial-Value Problems with SUNDIALS

- SUNDIALS’ integrators consider initial-value problems of three basic types:
  - Explicit form [CVODE]: $\dot{y}(t) = f(t, y(t)), \quad y(t_0) = y_0$
  - Linearly-implicit, split form [ARKODE]: $M\,\dot{y}(t) = f_1(t, y(t)) + f_2(t, y(t)), \quad y(t_0) = y_0$
  - Differential-algebraic form [IDA]: $F(t, y(t), \dot{y}(t)) = 0, \quad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0$

- By “solve” we mean much more than merely following a recipe for updating the solution; we *adapt* the time step sizes to meet user-specified error tolerances:

$$\left[\frac{1}{N}\sum_{k=1}^{N}\left(\frac{\mathrm{error}_k}{\mathrm{rtol}\,|y_k| + \mathrm{atol}_k}\right)^2\right]^{1/2} < 1$$

  - $\mathrm{error} \in \mathbb{R}^N$ is the estimated temporal error in a given time step
  - $y \in \mathbb{R}^N$ is the current solution
  - $\mathrm{rtol} \in \mathbb{R}$ encodes the desired relative solution accuracy (number of significant digits)
  - $\mathrm{atol} \in \mathbb{R}^N$ is the ‘noise’ level for any solution component (protects against $y_k = 0$)

The “Skeleton” for Using SUNDIALS Integrators

1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. Create matrix, linear solver, nonlinear solver objects (if applicable); attach to integrator
   - Defaults exist for some of these, but may be replaced with problem-specific versions
   - Parallel scalability hinges on appropriate choices (discussed in last portion of tutorial)
5. Specify optional inputs to integrator and solver objects (tolerances, etc.)
6. Advance solution in time, either over specified time intervals $[a, b]$, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Supplying the Initial Condition Vector(s)

- As discussed earlier, all SUNDIALS integrators operate on data through the `NVector` API.
- Each provided vector module has a unique set of “constructors”, e.g.

```
N_Vector N_VNew_Serial(sunindextype length);

N_Vector N_VNew_Parallel(MPI_Comm comm, sunindextype loc_len, sunindextype glob_len);

N_Vector N_VMake_Cuda(MPI_Comm comm, sunindextype loc_len, sunindextype glob_len,
                      realtype *hdata, realtype *ddata);

N_Vector N_VMake_OpenMPDEV(sunindextype len, realtype *hdata, realtype *ddata);

N_Vector N_VMake_Petsc(Vec v);

N_Vector N_VMake_ParHyp(HYPRE_ParVector x);
```

- Once an application creates a vector for their data, they fill it with the initial conditions for the problem and supply it to the integrator, who “clones” it to create its workspace.

Supplying the Initial Condition Vector(s) – Fortran

- Fortran interfaces exist for most SUNDIALS vectors, with similar arguments as in C/C++. The serial, MPI-parallel and *hypre* `NVector` constructors are:

```
CALL FNVINITS(code, len, ier)

CALL FNVINITP(comm, code, loc_len, glob_len, ier)

CALL FNVINITPH(comm, code, loc_len, glob_len, ier)
```

- The `code` argument is an `INTEGER*4` flag indicating which integrator will use the vector (1 is CVODE, 2 is IDA, 3 is KINSOL, 4 is ARKODE).
- `ier` is an `INTEGER*4` return flag indicating success (0) or failure (1) of the constructor.
- The local/global length arguments are `INTEGER*8`.
- In our existing F77 interfaces we must use global memory to store the actual vector pointers; however, upcoming F2003 interfaces will streamline these interfaces (already in place for CVODE).

Supplying the IVP to the Integrator – RHS/Residual Functions

Once the problem data is encapsulated in a vector, all that remains for basic SUNDIALS usage is specification of the IVP itself:

- CVODE and ARKODE specify the IVP through right-hand side function(s):

```
 int (*RhsFn)(realtype t, N_Vector y, N_Vector ydot, void *user_data)
SUBROUTINE FCVFUN(T, Y, YDOT, IPAR, RPAR, IER)
```

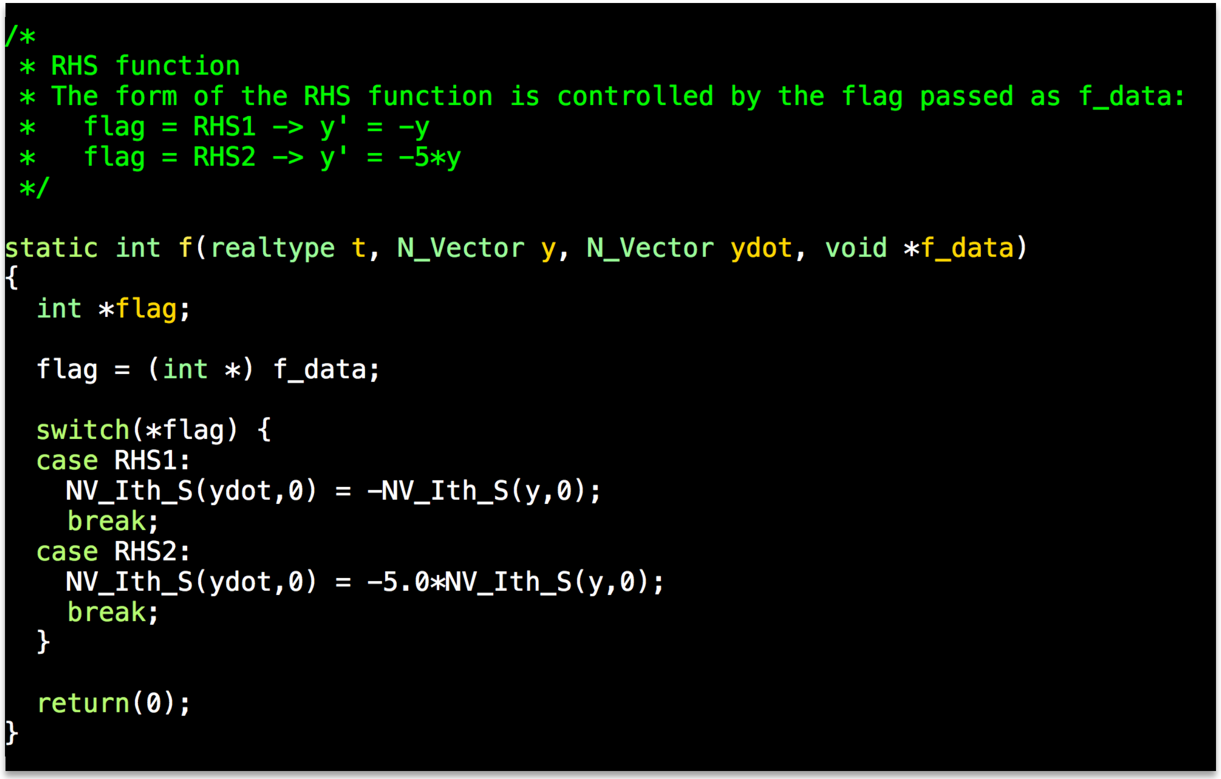
- IDA specifies the IVP through a residual function:

```
 int (*ResFn)(realtype t, N_Vector y, N_Vector ydot, N_Vector r,
                    void *user_data)
SUBROUTINE FIDARESFUN(T, Y, YDOT, R, IPAR, RPAR, IER)
```

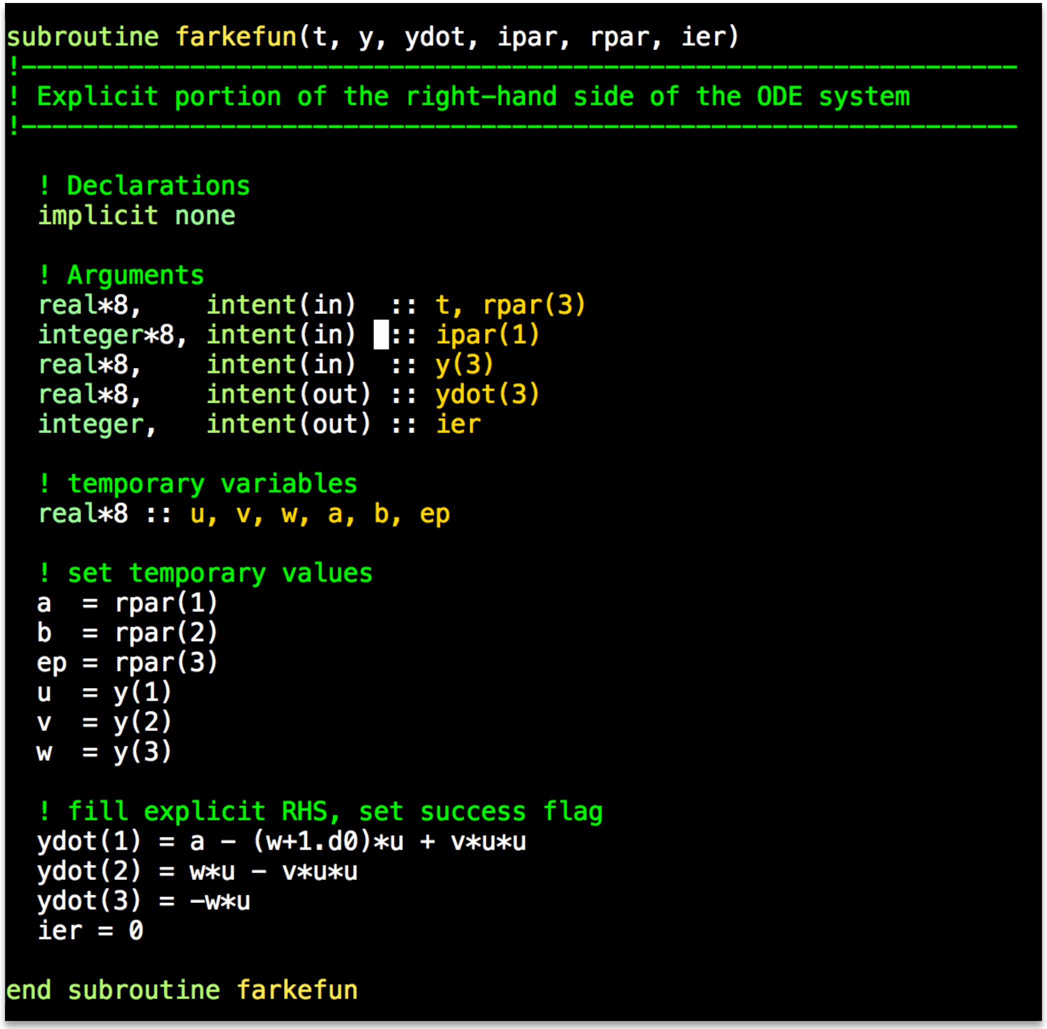
- In C/C++, `*user_data` enables problem-specific data to be passed through the SUNDIALS integrator and back to the RHS/residual routine (i.e., no global memory).

- In Fortran, this is handled through user-created `ipar` and `rpar` work arrays; many F90 codes instead use modules to handle user data.

CVODE/ARKODE RHS Functions – C (left) and F90 (right)



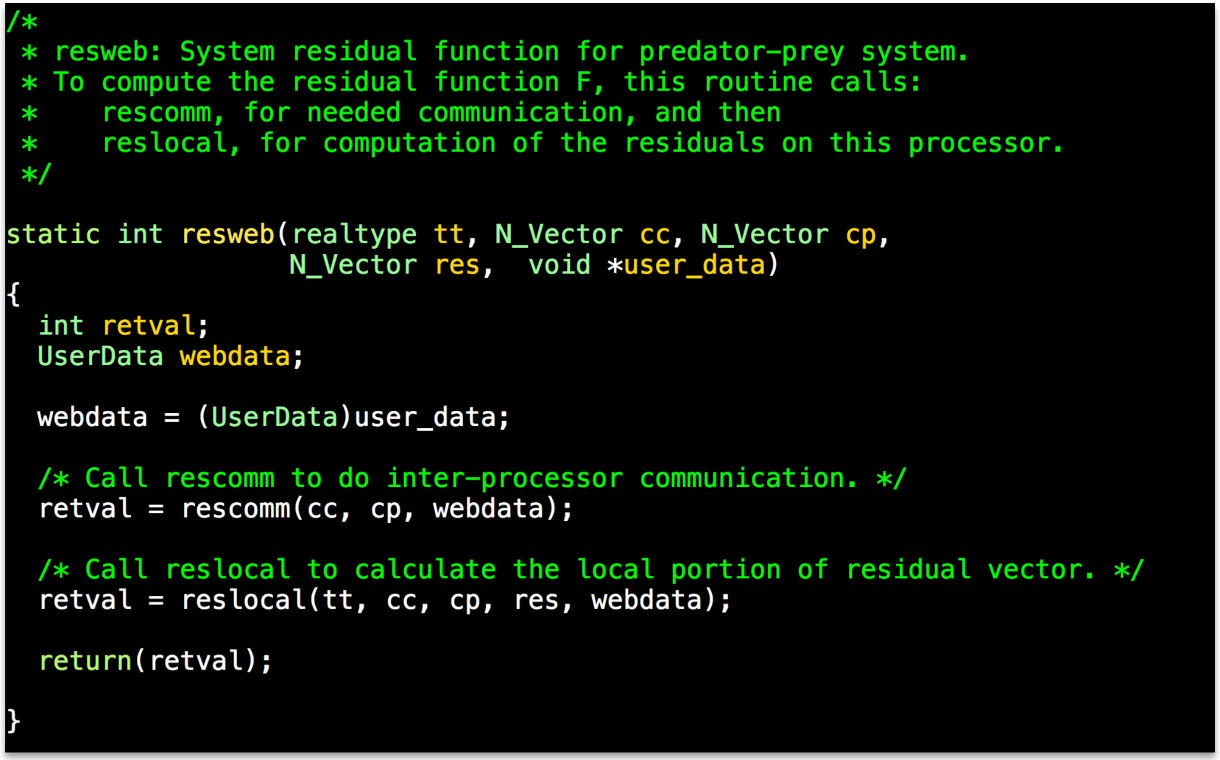
```
/*
 * RHS function
 * The form of the RHS function is controlled by the flag passed as f_data:
 *   flag = RHS1 -> y' = -y
 *   flag = RHS2 -> y' = -5*y
 */

static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
{
  int *flag;

  flag = (int *) f_data;

  switch(*flag) {
  case RHS1:
    NV_Ith_S(ydot,0) = -NV_Ith_S(y,0);
    break;
  case RHS2:
    NV_Ith_S(ydot,0) = -5.0*NV_Ith_S(y,0);
    break;
  }

  return(0);
}
```



```
subroutine farkefun(t, y, ydot, ipar, rpar, ier)
!-----------------------------------------------------------------
! Explicit portion of the right-hand side of the ODE system
!-----------------------------------------------------------------

  ! Declarations
  implicit none

  ! Arguments
  real*8,     intent(in)  :: t, rpar(3)
  integer*8,  intent(in)  :: ipar(1)
  real*8,     intent(in)  :: y(3)
  real*8,     intent(out) :: ydot(3)
  integer,    intent(out) :: ier

  ! temporary variables
  real*8 :: u, v, w, a, b, ep

  ! set temporary values
  a  = rpar(1)
  b  = rpar(2)
  ep = rpar(3)
  u  = y(1)
  v  = y(2)
  w  = y(3)

  ! fill explicit RHS, set success flag
  ydot(1) = a - (w+1.d0)*u + v*u*u
  ydot(2) = w*u - v*u*u
  ydot(3) = -w*u
  ier = 0

end subroutine farkefun
```

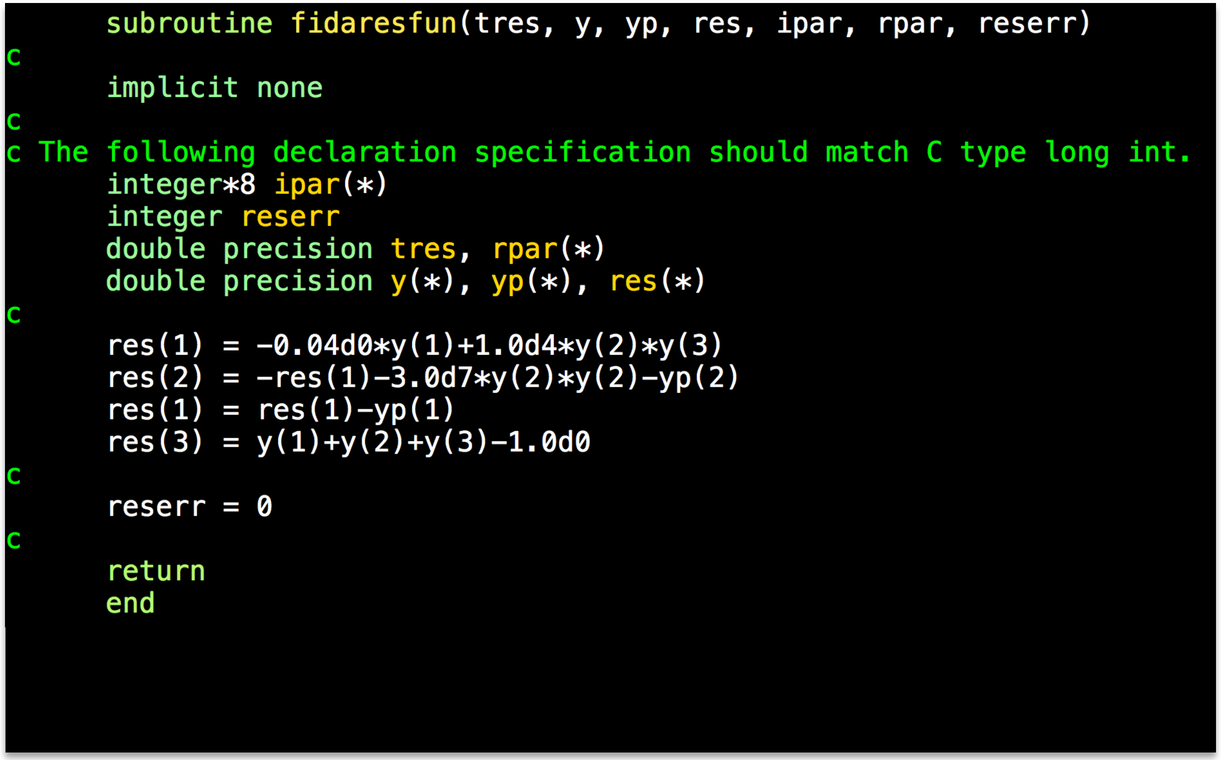
Left: cvDisc_dns.c;   Right: ark_bruss.f90

IDA Residual Function – C (left) and F77 (right)



```c
/*
 * resweb: System residual function for predator-prey system.
 * To compute the residual function F, this routine calls:
 *    rescomm, for needed communication, and then
 *    reslocal, for computation of the residuals on this processor.
 */

static int resweb(realtype tt, N_Vector cc, N_Vector cp,
                  N_Vector res,  void *user_data)
{
  int retval;
  UserData webdata;

  webdata = (UserData)user_data;

  /* Call rescomm to do inter-processor communication. */
  retval = rescomm(cc, cp, webdata);

  /* Call reslocal to calculate the local portion of residual vector. */
  retval = reslocal(tt, cc, cp, res, webdata);

  return(retval);

}
```

idaFoodWeb_kry_p.c



```fortran
      subroutine fidaresfun(tres, y, yp, res, ipar, rpar, reserr)
c
      implicit none
c
c The following declaration specification should match C type long int.
      integer*8 ipar(*)
      integer reserr
      double precision tres, rpar(*)
      double precision y(*), yp(*), res(*)
c
      res(1) = -0.04d0*y(1)+1.0d4*y(2)*y(3)
      res(2) = -res(1)-3.0d7*y(2)*y(2)-yp(2)
      res(1) = res(1)-yp(1)
      res(3) = y(1)+y(2)+y(3)-1.0d0
c
      reserr = 0
c
      return
      end
```

fidaRoberts_dns.f

Supplying the IVP to ARKODE – Mass Matrix Functions

When solving an IVP with non-identity mass matrix, users must supply either a routine to construct a mass matrix $M \in \mathbb{R}^{N\times N}$:

```
int (*ARKLsMassFn)(realtype t, SUNMatrix M, void *user_data,
                   N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
```

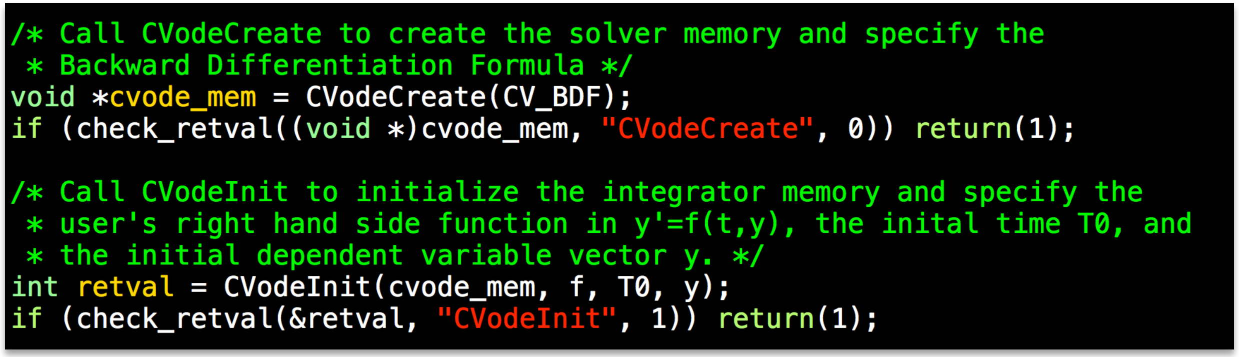
```
SUBROUTINE FARKDMASS(N, T, M, IPAR, RPAR, TMP1, TMP2, TMP3, IER)
```

or to perform the mass-matrix-vector product, $Mv : \mathbb{R}^N \to \mathbb{R}^N$:

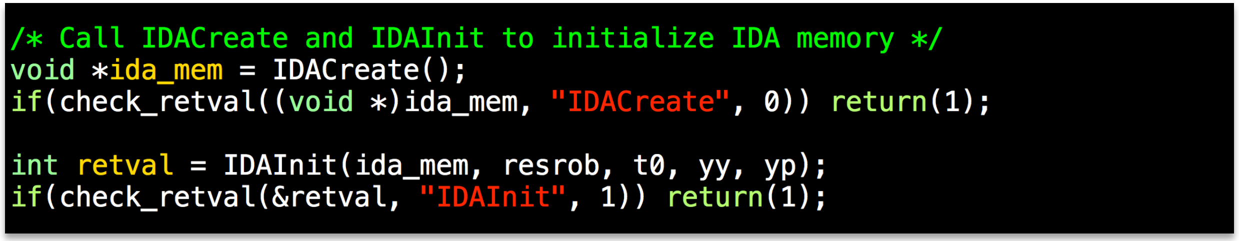
```
int (*ARKLsMassTimesSetupFn)(realtype t, void *mtimes_data);

int (*ARKLsMassTimesVecFn)(N_Vector v, N_Vector Mv, realtype t,
                           void *mtimes_data);

SUBROUTINE FARKMTSETUP(T, IPAR, RPAR, IER)

SUBROUTINE FARKMTIMES(V, MV, T, IPAR, RPAR, IER)
```

Initializing the Integrators from C/C++

The IVP inputs are supplied when constructing the integrator.

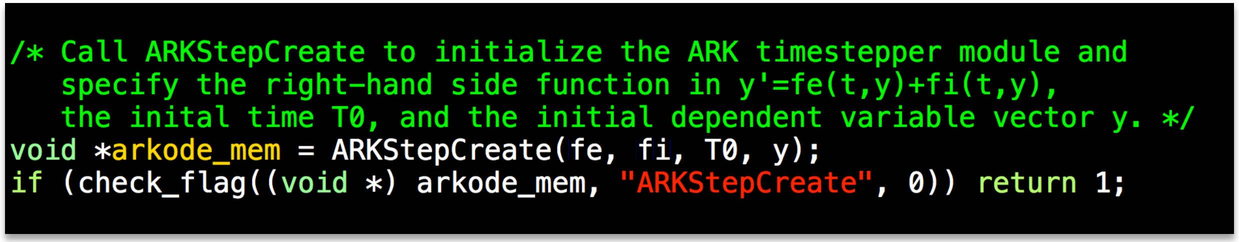


```
/* Call CVodeCreate to create the solver memory and specify the
 * Backward Differentiation Formula */
void *cvode_mem = CVodeCreate(CV_BDF);
if (check_retval((void *)cvode_mem, "CVodeCreate", 0)) return(1);

/* Call CVodeInit to initialize the integrator memory and specify the
 * user's right hand side function in y'=f(t,y), the inital time T0, and
 * the initial dependent variable vector y. */
int retval = CVodeInit(cvode_mem, f, T0, y);
if (check_retval(&retval, "CVodeInit", 1)) return(1);
```

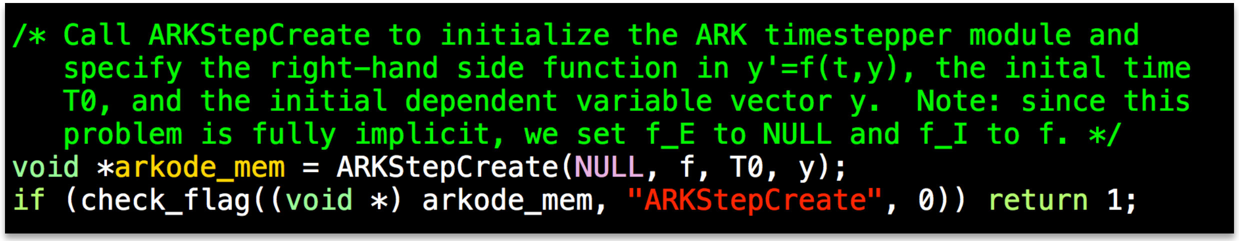


```
/* Call IDACreate and IDAInit to initialize IDA memory */
void *ida_mem = IDACreate();
if(check_retval((void *)ida_mem, "IDACreate", 0)) return(1);

int retval = IDAInit(ida_mem, resrob, t0, yy, yp);
if(check_retval(&retval, "IDAInit", 1)) return(1);
```

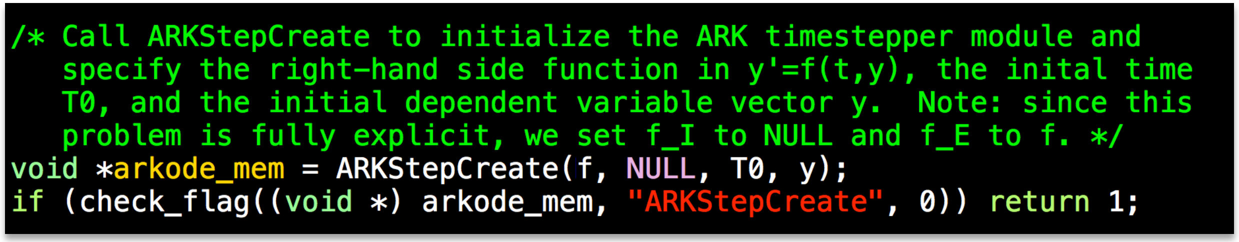
CVODE (top) and IDA (bottom)



```
/* Call ARKStepCreate to initialize the ARK timestepper module and
   specify the right-hand side function in y'=fe(t,y)+fi(t,y),
   the inital time T0, and the initial dependent variable vector y. */
void *arkode_mem = ARKStepCreate(fe, fi, T0, y);
if (check_flag((void *) arkode_mem, "ARKStepCreate", 0)) return 1;
```



```
/* Call ARKStepCreate to initialize the ARK timestepper module and
   specify the right-hand side function in y'=f(t,y), the inital time
   T0, and the initial dependent variable vector y.  Note: since this
   problem is fully implicit, we set f_E to NULL and f_I to f. */
void *arkode_mem = ARKStepCreate(NULL, f, T0, y);
if (check_flag((void *) arkode_mem, "ARKStepCreate", 0)) return 1;
```

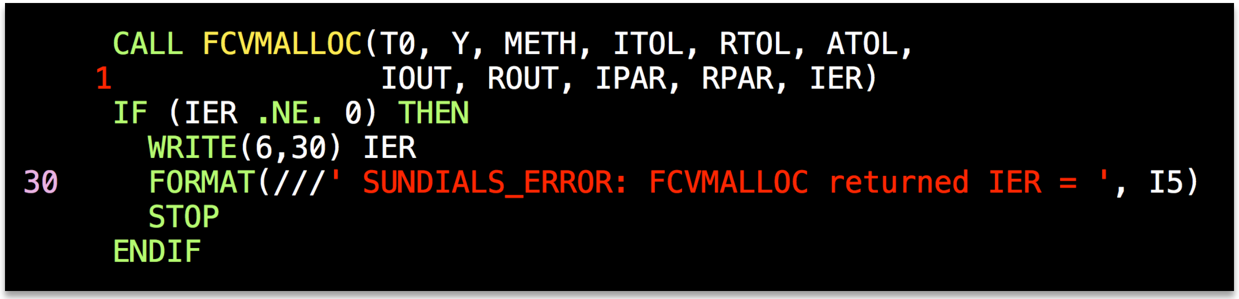


```
/* Call ARKStepCreate to initialize the ARK timestepper module and
   specify the right-hand side function in y'=f(t,y), the inital time
   T0, and the initial dependent variable vector y.  Note: since this
   problem is fully explicit, we set f_I to NULL and f_E to f. */
void *arkode_mem = ARKStepCreate(f, NULL, T0, y);
if (check_flag((void *) arkode_mem, "ARKStepCreate", 0)) return 1;
```

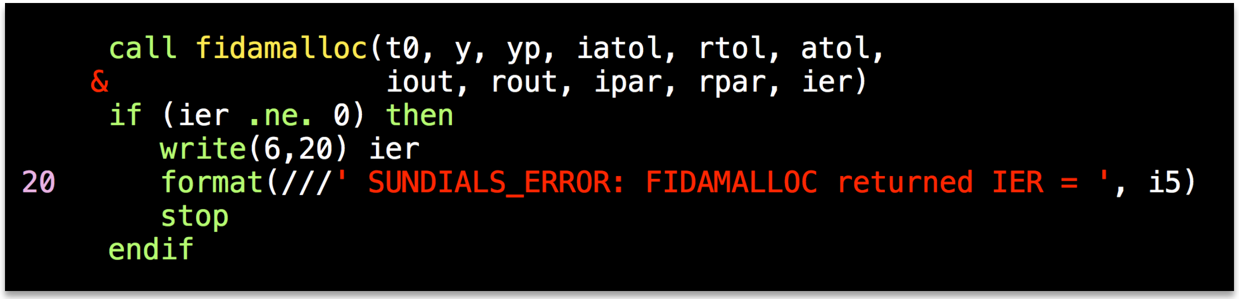
ARKODE IMEX (top), implicit (middle), explicit (bottom)

Initializing the Integrators from Fortran

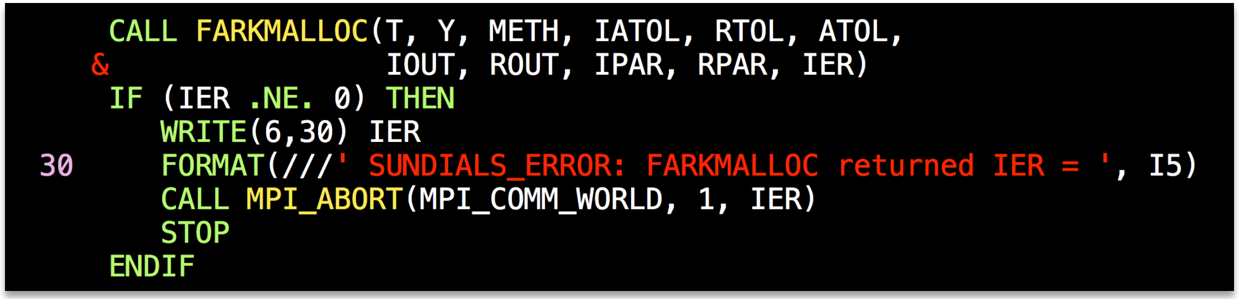
- Fortran users must provide problem-defining functions with specific names (FCVFUN, FIDARESFUN, FARKEFUN, FARKIFUN).
- Integrator options are specified with integer flags to the integrator's F*MALLOC routine.
- This is where the IPAR and RPAR user parameter arrays are supplied to the integrators, as well as initial time and initial condition(s).
- Additional IOUT and ROUT arrays are supplied to store solver statistics (returned from the integrators).



```
      CALL FCVMALLOC(T0, Y, METH, ITOL, RTOL, ATOL,
     1               IOUT, ROUT, IPAR, RPAR, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,30) IER
 30     FORMAT(///' SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
        STOP
      ENDIF
```



```
      call fidamalloc(t0, y, yp, iatol, rtol, atol,
     &                iout, rout, ipar, rpar, ier)
      if (ier .ne. 0) then
         write(6,20) ier
 20      format(///' SUNDIALS_ERROR: FIDAMALLOC returned IER = ', i5)
         stop
      endif
```



```
      CALL FARKMALLOC(T, Y, METH, IATOL, RTOL, ATOL,
     &                IOUT, ROUT, IPAR, RPAR, IER)
      IF (IER .NE. 0) THEN
         WRITE(6,30) IER
  30     FORMAT(///' SUNDIALS_ERROR: FARKMALLOC returned IER = ', I5)
         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
         STOP
      ENDIF
```

Optional Inputs (all Integrators)

A variety of optional inputs enable enhanced control over the integration process. Here we discuss the most often-utilized options (see documentation for the full set).

- Tolerance specification – rtol with scalar or vector-valued atol, or user-specified routine to compute the error weight vector

$$w_k \approx \frac{1}{\mathrm{rtol}\,|y_k| + \mathrm{atol}_k} > 0, \qquad k = 1, \ldots, N$$

- `SetNonlinearSolver`, `SetLinearSolver` – attaches desired nonlinear solver, linear solver and (optionally) matrix modules to the integrator.

- `SetUserData` – specifies the (`void *user_data`) pointer that is supplied to user routines.

- `SetMaxNumSteps`, `SetMaxStep`, `SetMinStep`, `SetInitStep` – provides guidance to time step adaptivity algorithms.

- `SetStopTime` – specifies the value of t_{stop} to use when advancing solution (this is retained until this stop time is reached or modified through a subsequent call).

Package-Specific Options (CVODE and IDA)

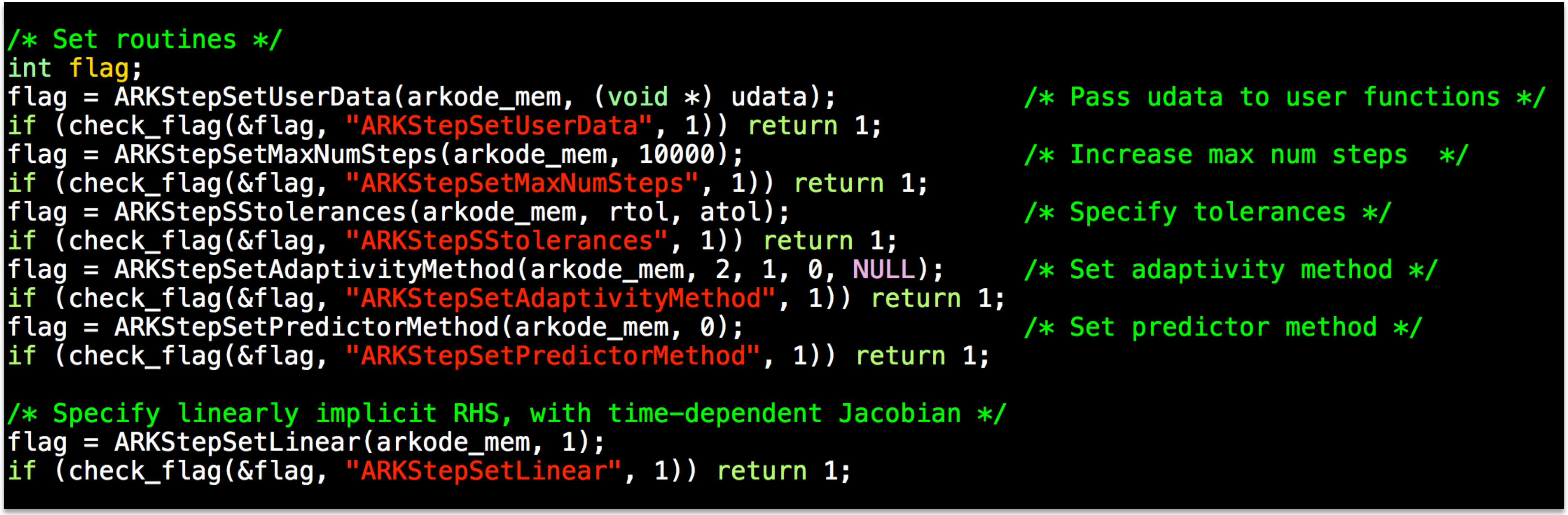
- `SetConstraints` – allows for setting positivity/negativity constraints on solution components.
- `SetMaxOrd` – specifies the maximum order of accuracy for the method (the order is adapted internally, along with the step size).
- `CalcIC` (IDA-specific) – in certain cases will help find a consistent y_0 .
  - A variety of additional routines may be used for additional control over this algorithm.
- `SetId` (IDA-specific) – specifies which variables are differential vs algebraic (useful when calling `CalcIC` above).

Package-Specific Options (ARKODE)

- `SetFixedStep` – disables time step adaptivity (and temporal error estimation/control).
- `SetLinear` – $f_1(t,y(t))$ depends *linearly* on y (disables nonlinear iteration).
- `SetOrder` – specifies the order of accuracy for the method.
- `SetTables` – allows user-specified ERK, DIRK or ARK Butcher tables.
- `SetAdaptivityFn` – allows user-provided routine for time step selection.
- New *multi-rate* time-stepping module, `MRIStep` – $f_1(t,y(t))$ and $f_2(t,y(t))$ are evolved with different user-specified time step sizes.

Supplying Options to the Integrators (C/C++)

After constructing the integrator, additional options may be supplied through various “Set” routines (example from `ark_heat1D_adapt.c`):



```
/* Set routines */
int flag;
flag = ARKStepSetUserData(arkode_mem, (void *) udata);            /* Pass udata to user functions */
if (check_flag(&flag, "ARKStepSetUserData", 1)) return 1;
flag = ARKStepSetMaxNumSteps(arkode_mem, 10000);                  /* Increase max num steps  */
if (check_flag(&flag, "ARKStepSetMaxNumSteps", 1)) return 1;
flag = ARKStepSStolerances(arkode_mem, rtol, atol);               /* Specify tolerances */
if (check_flag(&flag, "ARKStepSStolerances", 1)) return 1;
flag = ARKStepSetAdaptivityMethod(arkode_mem, 2, 1, 0, NULL);     /* Set adaptivity method */
if (check_flag(&flag, "ARKStepSetAdaptivityMethod", 1)) return 1;
flag = ARKStepSetPredictorMethod(arkode_mem, 0);                  /* Set predictor method */
if (check_flag(&flag, "ARKStepSetPredictorMethod", 1)) return 1;

/* Specify linearly implicit RHS, with time-dependent Jacobian */
flag = ARKStepSetLinear(arkode_mem, 1);
if (check_flag(&flag, "ARKStepSetLinear", 1)) return 1;
```

Supplying Custom Butcher tables to ARKODE

C/C++ users may construct custom Butcher tables and supply these to the integrator:

```
ARKodeButcherTable ARKodeButcherTable_Create(int s, int q, int p,
                      realtype *c, realtype *A, realtype *b, realtype *b2);

int ARKStepSetTables(void *arkode_mem, int q, int p,
                      ARKodeButcherTable Bi, ARKodeButcherTable Be);
```

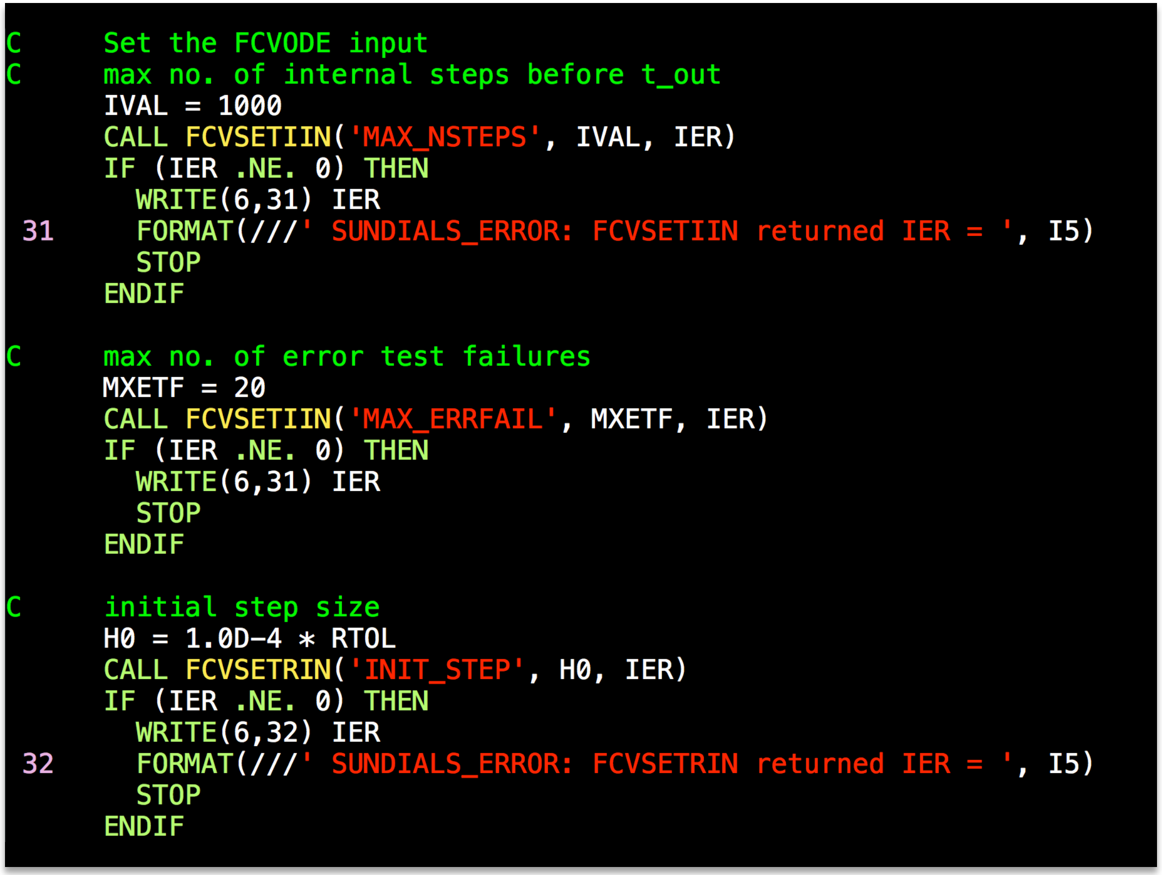
Fortran users instead provide the arrays directly:

```
CALL FARKSETERKTABLE(s, q, p, c, A, b, b2, ier)

CALL FARKSETIRKTABLE(s, q, p, c, A, b, b2, ier)

CALL FARKSETARKTABLES(s, q, p, ci, ce, Ai, Ae, bi, be, b2i, b2e,ier)
```

In each, “A” is assumed to be an array of length s^2, stored in row-major order.

Supplying Options to the Integrators (Fortran)

- After calling `F*MALLOC`, Fortran users supply most optional inputs through calling `F*SETIIN` and `F*SETRIN` routines with a set of pre-defined flags (`MAX_NSTEPS`, `MAX_ERRFAIL`, etc.).
- Integer inputs are required to correspond to the C type “`long int`” (typically, `INTEGER*8`)
- Real inputs are required to correspond to the C type “`double`” (typically, `REAL*8`)
- `IER` is always an `INTEGER*4` flag indicating success (0) or failure (1) of the “Set” routine.



```
C     Set the FCVODE input
C     max no. of internal steps before t_out
      IVAL = 1000
      CALL FCVSETIIN('MAX_NSTEPS', IVAL, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,31) IER
 31     FORMAT(///' SUNDIALS_ERROR: FCVSETIIN returned IER = ', I5)
        STOP
      ENDIF

C     max no. of error test failures
      MXETF = 20
      CALL FCVSETIIN('MAX_ERRFAIL', MXETF, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,31) IER
        STOP
      ENDIF

C     initial step size
      H0 = 1.0D-4 * RTOL
      CALL FCVSETRIN('INIT_STEP', H0, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,32) IER
 32     FORMAT(///' SUNDIALS_ERROR: FCVSETRIN returned IER = ', I5)
        STOP
      ENDIF
```

Usage Modes for SUNDIALS Integrators

While t_0 is supplied at initialization, the *direction* of integration is specified on the first call to advance the solution toward the output time t_{out}. This may occur in one of four “usage modes”:

- Normal – take internal steps until t_{out} is reached or overtaken in the direction of integration, e.g. for forward integration $t_{n-1} < t_{\text{out}} \le t_n$; the returned solution $y(t_{\text{out}})$ is then computed by interpolation.
- One-step – take a single internal step $y_{n-1} \to y_n$ and then return control back to the calling program. If this step will overtake t_{out} then $y(t_{\text{out}})$ is interpolated; otherwise y_n is returned.
- Normal + TStop – take internal steps until the next step will overtake t_{stop} ; limit the next internal step so that $t_n = t_{\text{stop}}$. No interpolation is performed.
- One-step + TStop – performs a combination of both “One-step” and “TStop” modes above.

Advancing the Solution

Once all options have been set, the integrator is called to advance the solution toward t_{out}.

```
flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
if(check_flag(&flag, "CVode", 1)) break;
```

```
flag = CVode(cvode_mem, t1, y, &t, CV_ONE_STEP);
if (check_flag((void *)&flag, "CVode", 1)) return(1);
```

```
retval = IDASolve(ida_mem, tout, &tret, cc, cp, IDA_NORMAL);
if (check_retval(&retval, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
```

```
flag = ARKStepEvolve(arkode_mem, tout, y, &t, ARK_NORMAL);
if (check_flag(&flag, "ARKStepEvolve", 1)) return 1;
```

```
flag = ARKStepEvolve(arkode_mem, Tf, y, &t, ARK_ONE_STEP);
if (check_flag(&flag, "ARKStepEvolve", 1)) return 1;
```

```
ITASK = 1
CALL FCVODE(TOUT, T, Y, ITASK, IER)
```

```
itask = 1
call fidasolve(tout, tret, y, yp, itask, ier)
```

```
call FARKode(Tout, Tcur, y, 1, ier)
if (ier < 0) then
   write(0,*) 'Solver failure, stopping integration'
   stop
end if
```

C/C++ on left; Fortran on right
CVODE top, IDA middle, ARKODE bottom
Fortran's `ITASK` provides the `*_NORMAL` or `*_ONE_STEP` argument.

Optional Outputs

Either between calls to advance the solution, or at the end of a simulation, users may retrieve a variety of optional outputs from SUNDIALS integrators.

- GetDky (Dense solution output) – using the same infrastructure that performs interpolation in "normal" use mode, users may request values $\frac{d^k}{dt^k}y(t)$ for $t_{n-1} \le t \le t_n$, where $0 \le k \le k_{\max}$.
- Time integration statistics:
  - GetNumSteps – the total number of internal time steps since initialization
  - GetCurrentStep – the current internal time step size
  - GetCurrentTime – the current internal time (since this may have passed t_{out})
  - GetCurrentOrder (IDA/CVODE) – the current method order of accuracy
  - GetActualInitStep – the size of the very first internal time step
  - GetNumErrTestFails – the number of steps that failed the temporal error test
  - GetEstLocalErrors – returns the current temporal error vector, $\mathrm{error} \in \mathbb{R}^N$

Optional Outputs – Algebraic Solver Statistics

- `GetNumNonlinSolvIters` – number of nonlinear solver iterations since initialization.
- `GetNumNonlinSolvConvFails` – number of nonlinear solver convergence failures.
- `GetNumLinSolvSetups` – number of calls to setup the linear solver or preconditioner.
- `GetNumLinIters` – number of linear solver iterations since initialization.
- `GetNumLinConvFails` – number of linear solver convergence failures.
- `GetNumJacEvals`, `GetNumJtimesEvals`, `GetNumPrecEvals`, `GetNumPrecSolves` – the number of calls to user-supplied Jacobian/preconditioner routines.

Optional Outputs – Miscellaneous Feedback

- `GetTolScaleFactor` – returns a suggested factor for scaling the user's `rtol, atol` values.

- `GetErrWeights` – returns the current error weight vector, $w \in \mathbb{R}^N$.

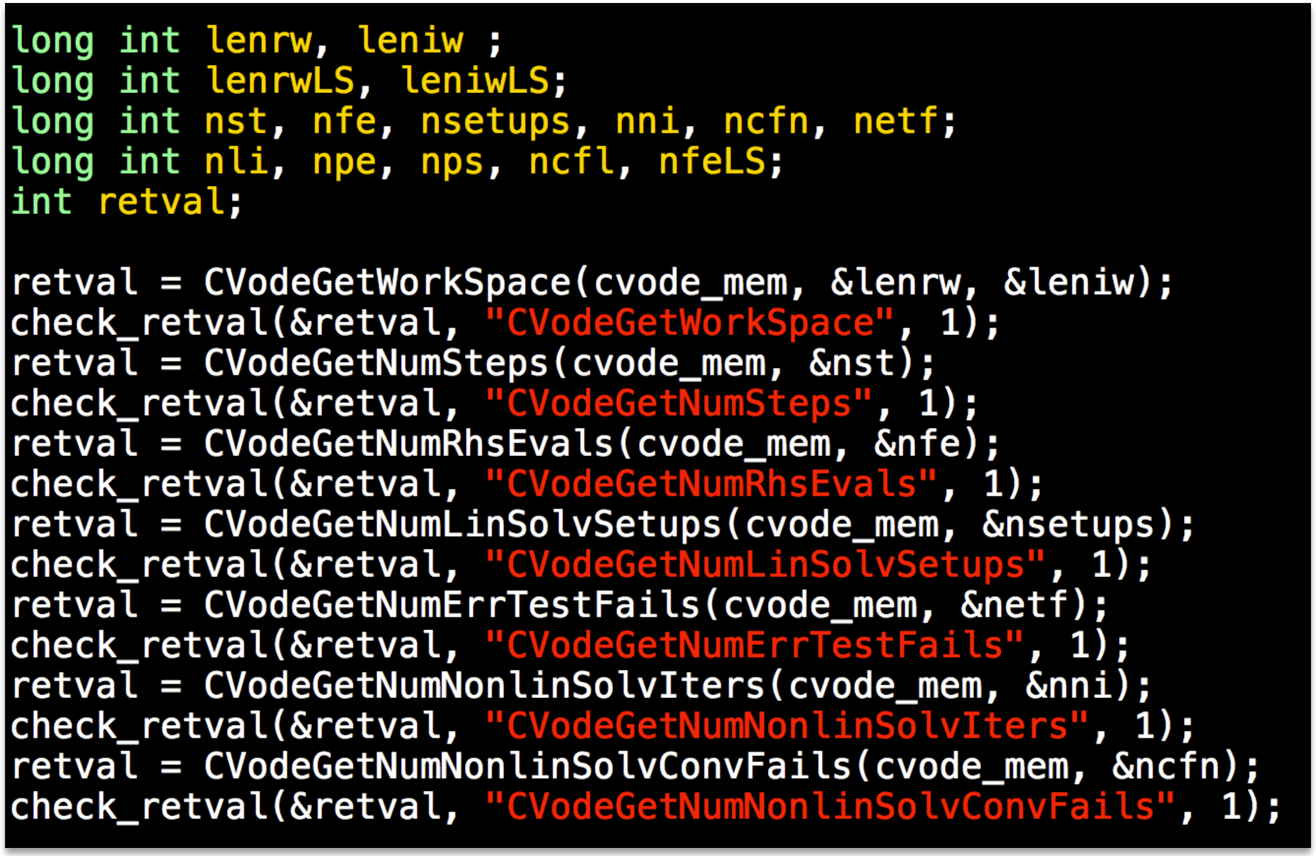
- `GetWorkspace` – returns the memory requirements for the integrator.

- `GetLinWorkspace` – returns the memory requirements for the linear solver.

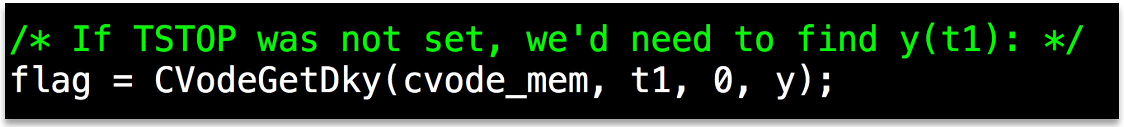
- `GetNumRhsEvals`, `GetNumResEvals` – returns the number of calls to the IVP RHS/residual function(s) by the integrator (nonlinear solve and time integration).

- `GetNumLinRhsEvals`, `GetNumLinResEvals` – returns the number of calls to the IVP RHS/residual function(s) by the linear solver (Jacobian or Jacobian-vector product approximation).

Retrieving Output from the Integrators (C/C++)



```
long int lenrw, leniw ;
long int lenrwLS, leniwLS;
long int nst, nfe, nsetups, nni, ncfn, netf;
long int nli, npe, nps, ncfl, nfeLS;
int retval;

retval = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
check_retval(&retval, "CVodeGetWorkSpace", 1);
retval = CVodeGetNumSteps(cvode_mem, &nst);
check_retval(&retval, "CVodeGetNumSteps", 1);
retval = CVodeGetNumRhsEvals(cvode_mem, &nfe);
check_retval(&retval, "CVodeGetNumRhsEvals", 1);
retval = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
check_retval(&retval, "CVodeGetNumLinSolvSetups", 1);
retval = CVodeGetNumErrTestFails(cvode_mem, &netf);
check_retval(&retval, "CVodeGetNumErrTestFails", 1);
retval = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
check_retval(&retval, "CVodeGetNumNonlinSolvIters", 1);
retval = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
check_retval(&retval, "CVodeGetNumNonlinSolvConvFails", 1);
```

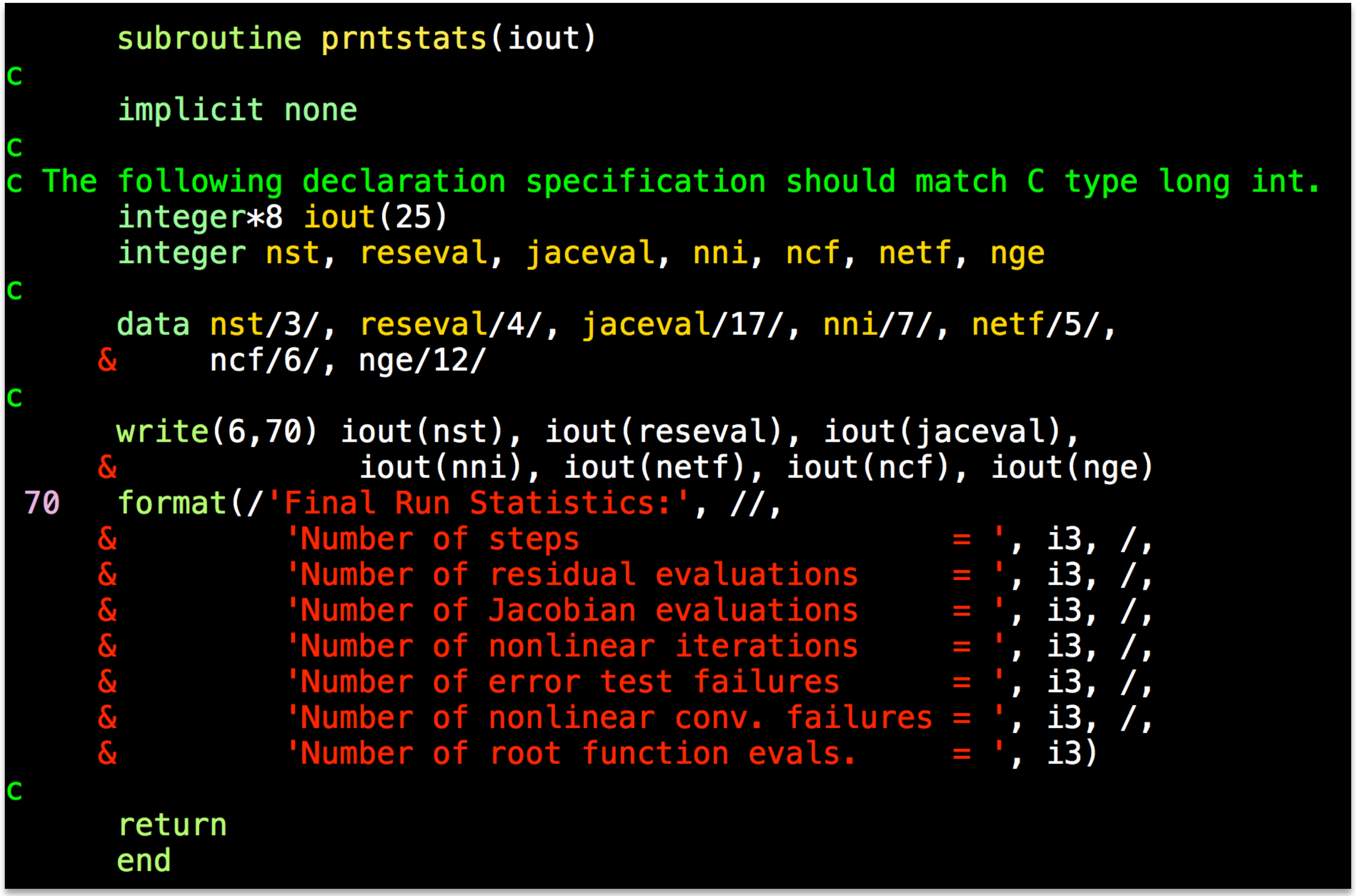


```
/* If TSTOP was not set, we'd need to find y(t1): */
flag = CVodeGetDky(cvode_mem, t1, 0, y);
```

Left: scalar-valued solver statistics from `cvAdvDiffReac_kry.c`

Right: dense solution output from `cvDisc_dns.c`

Retrieving Output from the Integrators (Fortran)



```
      subroutine prntstats(iout)
c
      implicit none
c
c The following declaration specification should match C type long int.
      integer*8 iout(25)
      integer nst, reseval, jaceval, nni, ncf, netf, nge
c
      data nst/3/, reseval/4/, jaceval/17/, nni/7/, netf/5/,
     &     ncf/6/, nge/12/
c
      write(6,70) iout(nst), iout(reseval), iout(jaceval),
     &             iout(nni), iout(netf), iout(ncf), iout(nge)
 70   format(/'Final Run Statistics:', //,
     &         'Number of steps                    = ', i3, /,
     &         'Number of residual evaluations     = ', i3, /,
     &         'Number of Jacobian evaluations     = ', i3, /,
     &         'Number of nonlinear iterations     = ', i3, /,
     &         'Number of error test failures      = ', i3, /,
     &         'Number of nonlinear conv. failures = ', i3, /,
     &         'Number of root function evals.     = ', i3)
c
      return
      end
```

Example from `fidaRoberts_dns.f`:

- The `iout` and `rout` arrays, passed to the `F*MALLOC` routines, are filled with solver statistics at the end of each call to advance the solution.
- The required lengths of these `INTEGER*8` and `REAL*8` arrays are specified in each package’s documentation

Advanced Features

This tutorial is only the beginning; SUNDIALS also supports a number of ‘advanced’ features to examine auxiliary conditions, change the IVP, and improve solver efficiency.

- Root-finding – while integrating the IVP, SUNDIALS integrators can find roots of a set of auxiliary user-defined functions $g_i(t, y(t)),\ i = 1, \ldots, N_r$; sign changes are monitored between time steps, and a modified secant iteration is used (along with GetDky) to home in on the roots.
- Reinitialization – allows reuse of existing integrator memory for a “new” problem (e.g., when integrating across a discontinuity, or integrating many independent problems of the same size). All solution history and solver statistics are erased, but no memory is (de)allocated.
- Resizing (ARKODE) – allows resizing the problem and all internal vector memory, without destruction of temporal adaptivity heuristic information or solver statistics. This is primarily useful when integrating problems with spatial adaptivity.
- Sensitivity Analysis (CVODE/IDA) – allows computation of solution sensitivities with respect to problem parameters (see overview portion of Tutorial for additional information).

Tutorial Outline

- Overview of SUNDIALS (Carol Woodward)
- How to download and install SUNDIALS (Cody Balos)
- How to use the time integrators (Daniel Reynolds)
- **Which nonlinear and linear solvers are available and how to use them (David Gardner)**

Nonlinear and Linear Solvers in SUNDIALS – Overview

- SUNDIALS' implicit integrators solve one or more nonlinear systems each time step using generic nonlinear and linear solver operations.

- SUNDIALS provides two nonlinear solver modules and several linear solver modules:
  - Nonlinear: Newton (default) and Fixed Point with optional Anderson acceleration
  - Linear (direct): Dense, Band, LAPACK Dense/Band, KLU, and SuperLU_MT
  - Linear (iterative, scaled): GMRES, FGMRES, TFQMR, BiCGStab, Conjugate Gradient

- It is also straightforward to provide problem-specific nonlinear and linear solver modules:
  - The solver *content* data structure is stored as a "black-box" pointer `(void *)`
  - Solver operations are implemented at the user level, with corresponding function pointers stored in the solver *ops* structure
  - Not all operations are required and unneeded operations may be set to `NULL`; required routines are clearly documented in the user guide

Newton Solver

- SUNDIALS' implicit integrators require solving the nonlinear systems:

$$\left.\begin{array}{ll} \text{– CVODE:} & y^n - h_n \beta_{n,0} f(t_n, y^n) - a_n = 0 \\ \text{– ARKODE:} & M z_i - h_n A^I_{i,i} f_I(t^I_{n,i}, z_i) - a_i = 0 \\ \text{– IDA:} & F(t_n, y^n, h_n^{-1} \sum_{i=0}^{q} \alpha_{n,i} y^{n-i}) = 0 \end{array}\right\} F(y) = 0$$

These can all be posed as a generic root-finding problem

- By default the integrators solve $F(y) = 0$ with a Newton iteration:

$$\left.\begin{array}{l} y^{(m+1)} = y^{(m)} + \delta^{(m+1)} \\ J(y^{(m)})\delta^{(m+1)} = -F(y^{(m)}) \qquad J \equiv \partial F / \partial y \end{array}\right\} Ax = b$$

Requires solving a general linear system each iteration

- A general linear solver is also needed when using ARKODE with a non-identity mass matrix.

Linear Solver Types

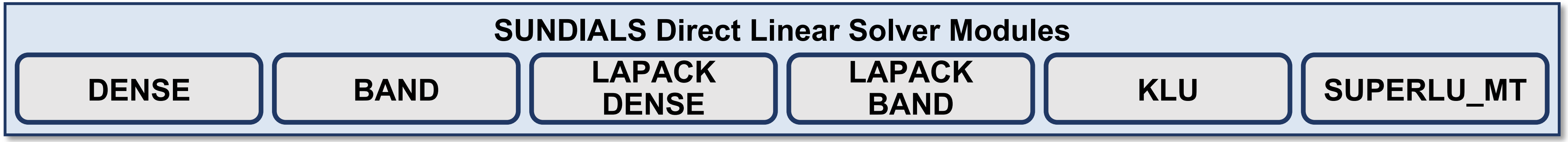
- When using the default nonlinear solver (Newton), users only need to create and attach the desired linear solver object.

- The variant of Newton's method employed depends on the linear solver type:
  - **Direct:** a matrix object is *required* and the solver computes the "exact" solution to the linear system defined by the matrix.
  - **Iterative (matrix-free):** a matrix object is *not required* and the solver computes an inexact solution to the linear system defined by the Jacobian-vector product routine.
  - **Matrix-Iterative (matrix-based):** a matrix object is *required* and the solver computes an inexact solution to the linear system defined by the matrix.

- SUNDIALS provides several direct and iterative linear solver modules.

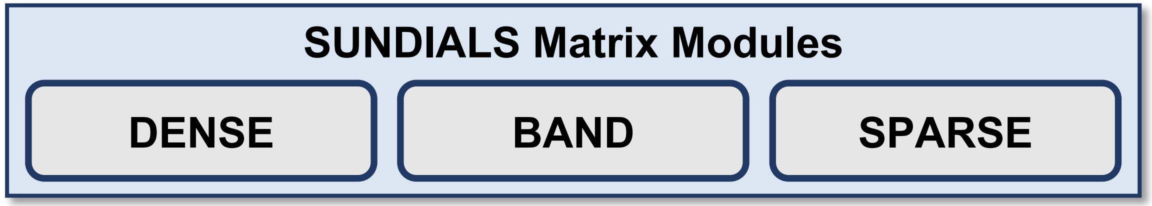
- Users may supply problem-specific direct, iterative, or matrix-iterative modules.

Direct Linear Solvers



SUNDIALS Direct Linear Solver Modules

DENSE | BAND | LAPACK DENSE | LAPACK BAND | KLU | SUPERLU_MT



SUNDIALS Matrix Modules

DENSE | BAND | SPARSE

- Direct linear solvers require the use of a compatible matrix module.
- When used with a direct linear solver the Newton iteration is a *modified Newton iteration*.
  - The Jacobian is updated infrequently to amortize the cost of matrix construction.
  - Optional integrator inputs are provided to adjust the Jacobian update frequency.

The “Skeleton” for Using SUNDIALS Integrators

1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. **Create matrix and linear solver objects; attach to integrator**
   - Using the default Newton nonlinear solver
5. Specify optional inputs to integrator and solver objects (tolerances, etc.)
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Creating & Attaching a Direct Linear Solver

- In the “Usage Skeleton,” step 4 would consist of:

a) Create an `NxN` SUNMatrix object

- `SUNMatrix A = SUNDenseMatrix(N, N)`
- `SUNMatrix A = SUNBandMatrix(N, upperwidth, lowerwidth)`
- `SUNMatrix A = SUNSparseMatrix(N, N, NNZ, type)`

b) Create the SUNLinearSolver object (`*` is the solver name)

- `SUNLinearSolver LS = SUNLinSol_*(y, A,…)`

c) Attach the linear solver to the integrator (`*` is the integrator prefix)

- `ier = *SetLinearSolver(mem, LS, A)`

The “Skeleton” for Using SUNDIALS Integrators

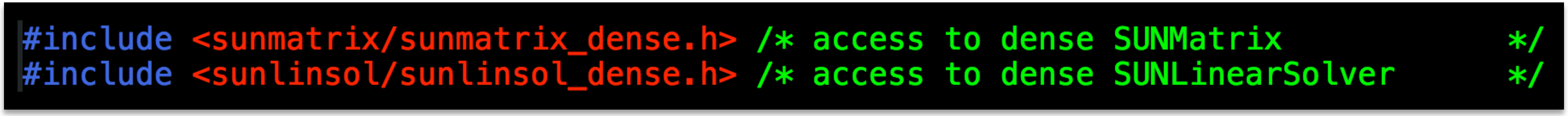
1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. Create matrix and linear solver objects; attach to integrator
5. **Specify optional inputs to integrator and solver objects (tolerances, etc.)**
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Direct Linear Solver Options

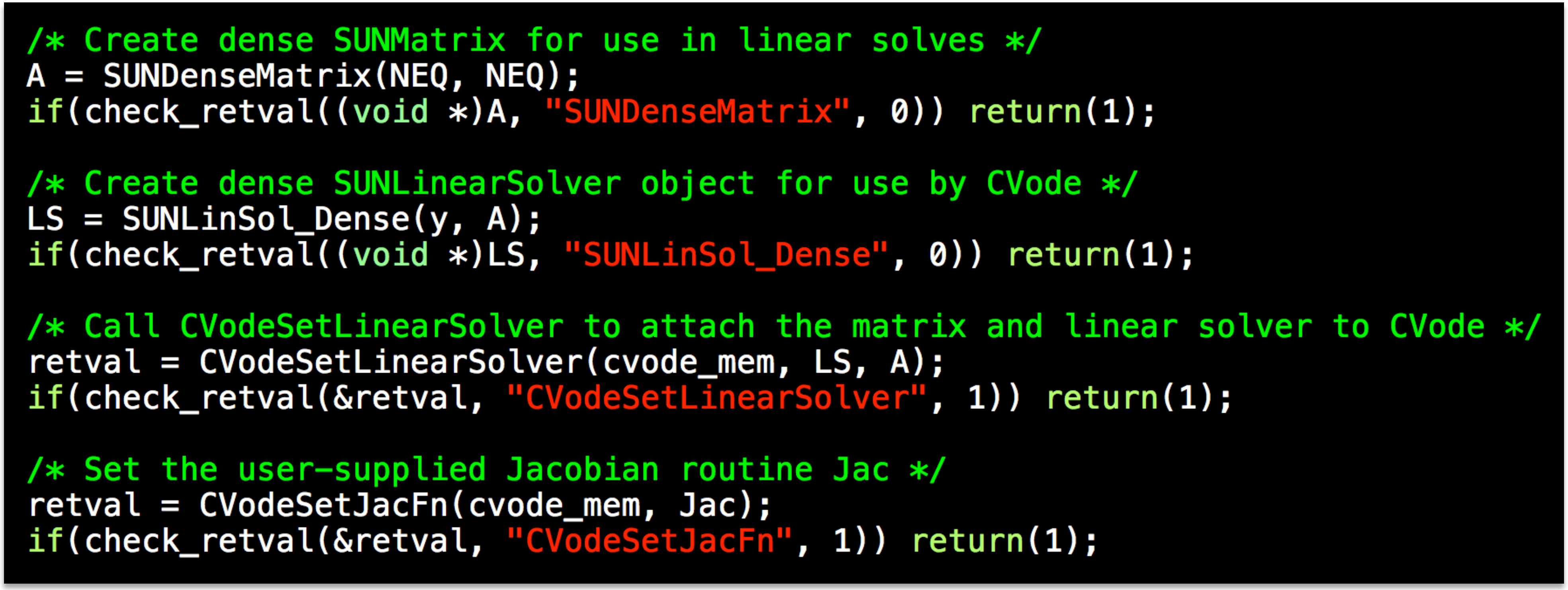
- In the “Usage Skeleton” step 5 could include the following optional inputs:
  - `SetJacFn` – specifies a user-supplied function for evaluating the Jacobian.
    - With dense and banded matrices the Jacobian of the IVP function may be computed internally with finite differences (default) or by a user-supplied function.
    - Sparse and user-supplied matrices require a user-supplied function to compute the Jacobian of the IVP function.
  - `SetMaxStepsBetweenJac` – (CVODE and ARKODE) – specifies the number of steps to wait before recomputing the Jacobian in a call to the linear solver setup routine.
  - `SetMaxStepsBetweenLSet` – (ARKODE) – specifies the number of steps between calls to the linear solver setup routine to potentially recompute the Jacobian of the IVP function.

examples/cvode/serial/cvRoberts_dns.c

- Example using a dense matrix, dense linear solver, and user supplied Jacobian routine.

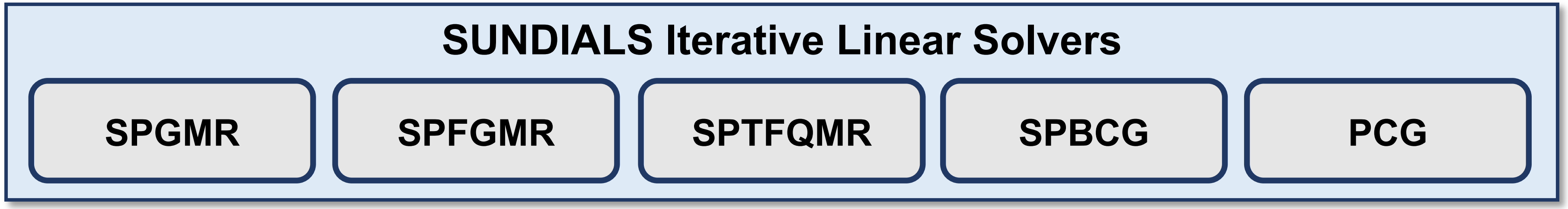


```
#include <sunmatrix/sunmatrix_dense.h> /* access to dense SUNMatrix            */
#include <sunlinsol/sunlinsol_dense.h> /* access to dense SUNLinearSolver      */
```



```
/* Create dense SUNMatrix for use in linear solves */
A = SUNDenseMatrix(NEQ, NEQ);
if(check_retval((void *)A, "SUNDenseMatrix", 0)) return(1);

/* Create dense SUNLinearSolver object for use by CVode */
LS = SUNLinSol_Dense(y, A);
if(check_retval((void *)LS, "SUNLinSol_Dense", 0)) return(1);

/* Call CVodeSetLinearSolver to attach the matrix and linear solver to CVode */
retval = CVodeSetLinearSolver(cvode_mem, LS, A);
if(check_retval(&retval, "CVodeSetLinearSolver", 1)) return(1);

/* Set the user-supplied Jacobian routine Jac */
retval = CVodeSetJacFn(cvode_mem, Jac);
if(check_retval(&retval, "CVodeSetJacFn", 1)) return(1);
```

Iterative Linear Solvers



SUNDIALS Iterative Linear Solvers

SPGMR

SPFGMR

SPTFQMR

SPBCG

PCG

- SUNDIALS iterative linear solvers support scaling and preconditioning, as applicable, to balance the error between solution components and to accelerate convergence.
  - For linear solvers that do not support scaling, the linear solver tolerance supplied is adjusted to compensate, but may be non-optimal when components vary dramatically.
- When used with an iterative linear solver the Newton iteration is an *inexact Newton iteration*.
  - The linear system is solved to a specified tolerance and the preconditioner is updated infrequently to amortize cost.
  - Optional integrator inputs are provided to adjust the linear tolerance and the frequency with which the preconditioner is updated.

The “Skeleton” for Using SUNDIALS Integrators

1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. **Create linear solver object; attach to integrator**
   - Using the default Newton nonlinear solver
5. Specify optional inputs to integrator and solver objects (tolerances, etc.)
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Creating & Attaching an Iterative Linear Solver

- In the "Usage Skeleton," step 4 would consist of:

a) Create the SUNLinearSolver object (`*` is the solver name)

- `SUNLinearSolver LS = SUNLinSol_*(y, pretype, maxl)`

b) Set linear solver optional inputs (`*` is the solver name and `**` is the option name)

- Call `SUNLinSol_*Set**` functions to change solver specific optional inputs

c) Attach the linear solver (`*` is the integrator prefix; note that a `NULL` matrix is supplied)

- `ier = *SetLinearSolver(mem, LS, NULL)`

Iterative Linear Solver Options

- Solver specific options include:
  - `SetGSType` – (GMR and FGMR) – sets the Gram-Schmidt orthogonalization type (`CLASSICAL` or `MODIFIED`); the default is modified Gram-Schmidt.
  - `SetMaxRestarts` – (GMR and FGMR) – sets the max number of GMRES restarts; the default is 0.
  - `SetMaxl` – (BCGS, TFQMR, and PCG) – updates the number of linear solver iterations; the default is 5.

The “Skeleton” for Using SUNDIALS Integrators

1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. Create linear solver object; attach to integrator
5. **Specify optional inputs to integrator and solver objects (tolerances, etc.)**
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Iterative Linear Solver Options

- In the “Usage Skeleton” step 5 could include the following optional inputs:
  - `SetJacTimes` – set user-supplied Jacobian-vector product setup and times functions.
    - By default Jacobian-vector products are computed internally using a finite difference
  - `SetEpsLin` – specifies the scaling factor used to set the linear solver tolerance.
  - `SetPreconditioner` – set the preconditioner setup and solve functions. See the next slide for more details.
  - `SetMaxStepsBetweenJac` – (CVODE and ARKODE) – specifies the number of steps to wait before recommending to update the preconditioner.
  - `SetMaxStepsBetweenLSet` – (ARKODE) – specifies the number of steps between calls to the linear solver setup routine to potentially update the preconditioner.

Iterative Linear Solvers – Supplying a Preconditioner

- The `SetPreconditioner` function sets the preconditioner setup and solve functions:
  - The preconditioner setup function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner. CVODE/ARKode example:

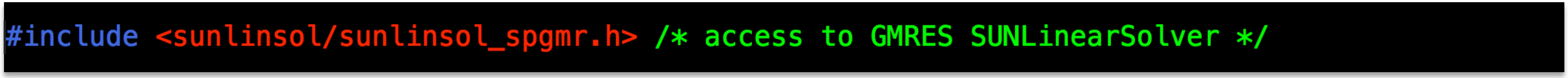
```
LsPrecSetupFn(realtype t, N_Vector y, N_Vector fy, booleantype jok,
              booleantype* jcurPtr, realtype gamma, void* user_data)
```

  - The preconditioner solve function solves the preconditioner system $Pz = r$. CVODE/ARKode example:

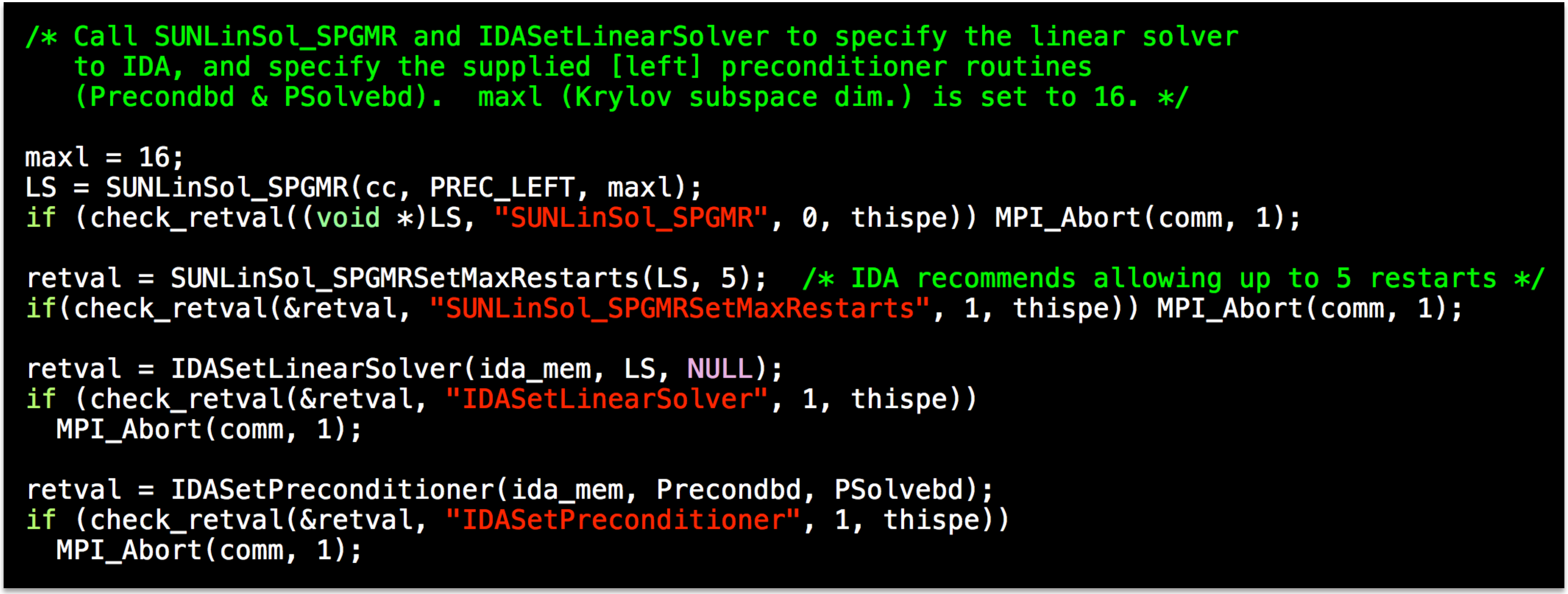
```
LsPrecSolvFn(realtype t, N_Vector y, N_Vector fy, N_Vector r,
             N_Vector z, realtype gamma, realtype delta, int lr,
             void* user_data)
```

examples/ida/parallel/idaFoodWeb_kry_p.c

- Example using GMRES with restarts and a user supplied block diagonal preconditioner.



```
#include <sunlinsol/sunlinsol_spgmr.h> /* access to GMRES SUNLinearSolver */
```



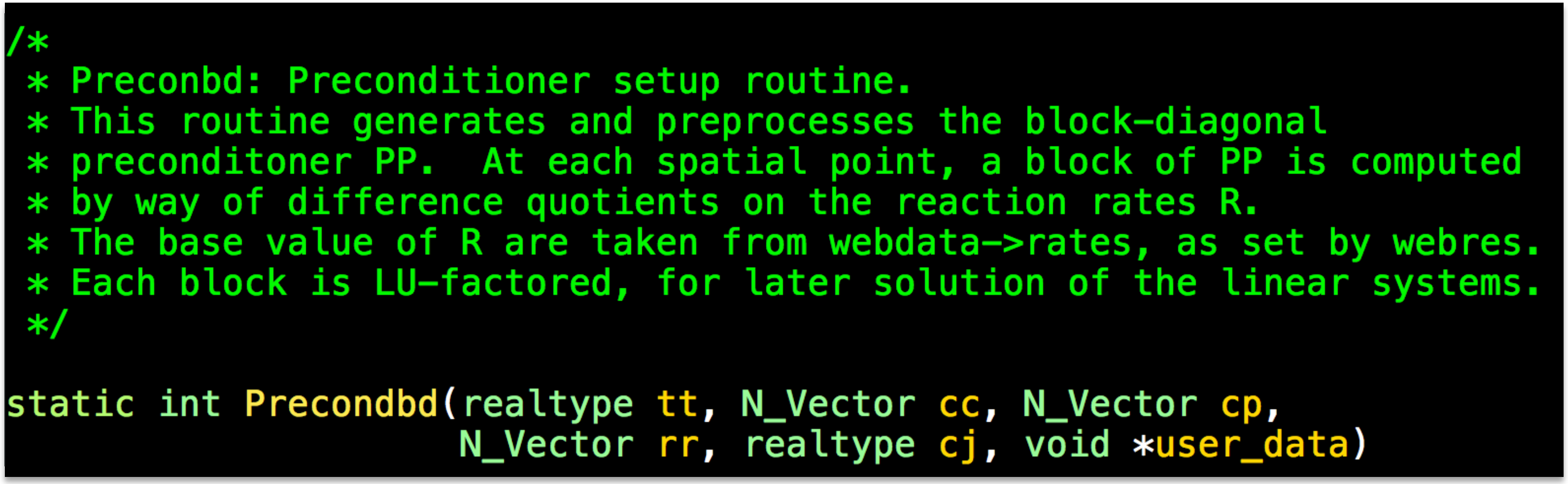
```
/* Call SUNLinSol_SPGMR and IDASetLinearSolver to specify the linear solver
   to IDA, and specify the supplied [left] preconditioner routines
   (Precondbd & PSolvebd).  maxl (Krylov subspace dim.) is set to 16. */

maxl = 16;
LS = SUNLinSol_SPGMR(cc, PREC_LEFT, maxl);
if (check_retval((void *)LS, "SUNLinSol_SPGMR", 0, thispe)) MPI_Abort(comm, 1);

retval = SUNLinSol_SPGMRSetMaxRestarts(LS, 5);  /* IDA recommends allowing up to 5 restarts */
if(check_retval(&retval, "SUNLinSol_SPGMRSetMaxRestarts", 1, thispe)) MPI_Abort(comm, 1);

retval = IDASetLinearSolver(ida_mem, LS, NULL);
if (check_retval(&retval, "IDASetLinearSolver", 1, thispe))
  MPI_Abort(comm, 1);

retval = IDASetPreconditioner(ida_mem, Precondbd, PSolvebd);
if (check_retval(&retval, "IDASetPreconditioner", 1, thispe))
  MPI_Abort(comm, 1);
```

examples/ida/parallel/idaFoodWeb_kry_p.c

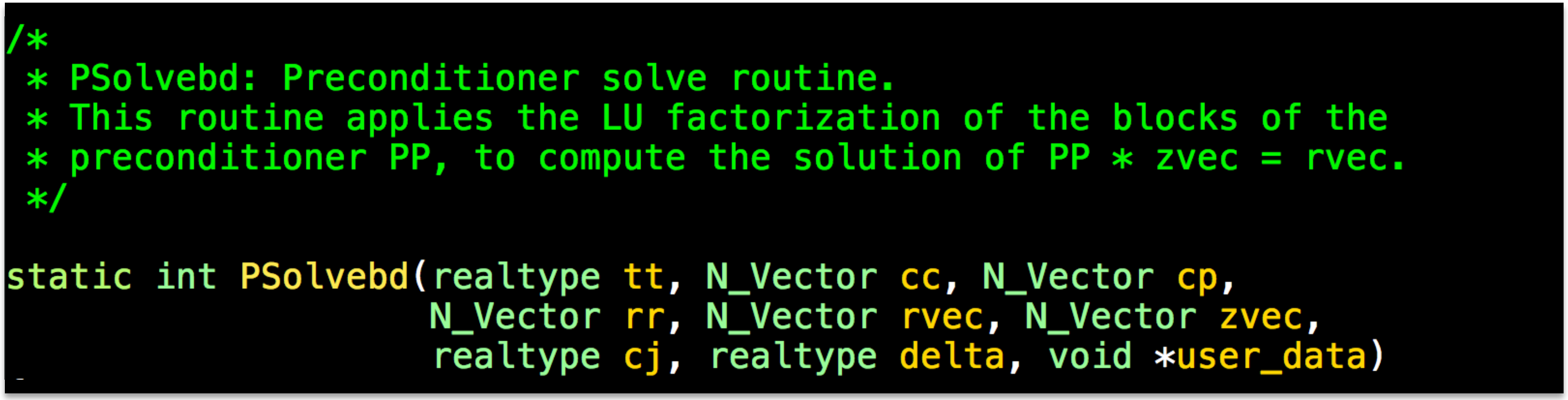
- Block diagonal preconditioner functions.

- Setup: `Precondbd`
  - Update Jacobian
  - Factor diagonal blocks

- Solve: `Psolvebd`
  - Solve the preconditioning system $Pz=r$



```
/*
 * Preconbd: Preconditioner setup routine.
 * This routine generates and preprocesses the block-diagonal
 * preconditoner PP.  At each spatial point, a block of PP is computed
 * by way of difference quotients on the reaction rates R.
 * The base value of R are taken from webdata->rates, as set by webres.
 * Each block is LU-factored, for later solution of the linear systems.
 */

static int Precondbd(realtype tt, N_Vector cc, N_Vector cp,
                     N_Vector rr, realtype cj, void *user_data)
```



```
/*
 * PSolvebd: Preconditioner solve routine.
 * This routine applies the LU factorization of the blocks of the
 * preconditioner PP, to compute the solution of PP * zvec = rvec.
 */

static int PSolvebd(realtype tt, N_Vector cc, N_Vector cp,
                    N_Vector rr, N_Vector rvec, N_Vector zvec,
                    realtype cj, realtype delta, void *user_data)
```

User-supplied Matrix-Iterative Linear Solver

- The ark_heat2D_hypre.cpp example demonstrates how to interface a problem-specific linear solver with a SUNDIALS integrator using the matrix-iterative linear solver type:
  - Matrix is supplied
  - Solve uses an iterative method

- This ARKODE example uses the default Newton iteration with *hypre* matrices, linear solvers, and preconditioners:
  - Creates a SUNMatrix wrapper for a *hypre* structured grid matrix
  - Creates a SUNLinearSolver wrapper for the *hypre* PCG solver with PFMG preconditioner

- When used with a matrix-iterative linear solver the Newton iteration is a *modified Newton iteration* and the Jacobian is updated infrequently to amortize the cost of matrix construction.

- The matrix-iterative type combines aspects of the dense and iterative types. As such, optional integrator inputs for both dense and iterative solvers apply to matrix-iterative solvers.

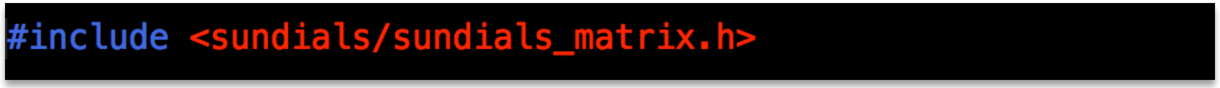
Creating a SUNMatrix Wrapper

- **Constructor** – creates a new matrix.
- **GetID**(A) – returns the matrix type.
- *Clone*(A) – returns a new matrix of the same type as A.
- *Destroy*(A) – frees memory allocated when creating A.
- Space(A, liw, lrw) – returns the storage requirements of A.
- **Zero**(A) – sets all entries of A to zero.
- *Copy*(A, B) – copies all entries from A to B.
- *ScaleAdd*(c, A, B) – performs the operation A=cA+B.
- *ScaleAddI*(c, A) – performs the operation A=cA+I.
- *Matvec*(A, x, y) – performs the operation y=Ax.

Key:
Always required
Sometimes required
Optional

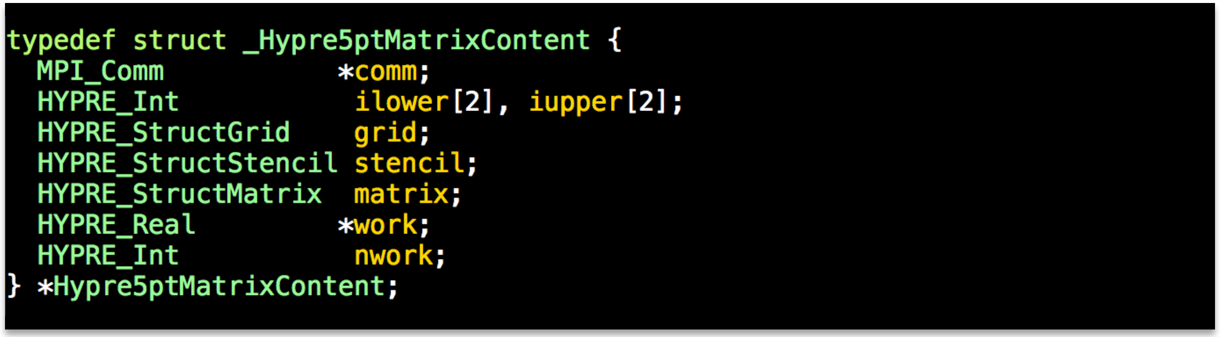
examples/arkode/CXX_parhyp/ark_heat_2D_hypre.cpp

- Header defining a generic SUNMatrix



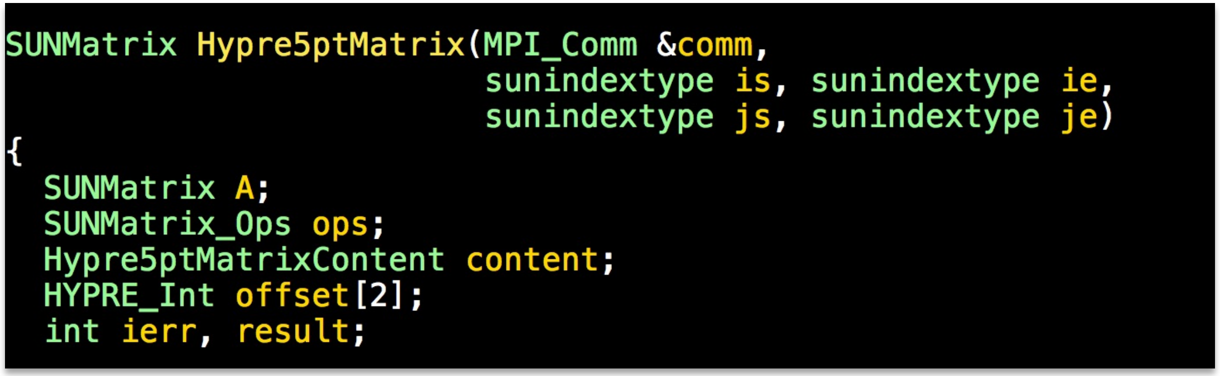
```
#include <sundials/sundials_matrix.h>
```

- Matrix specific content structure



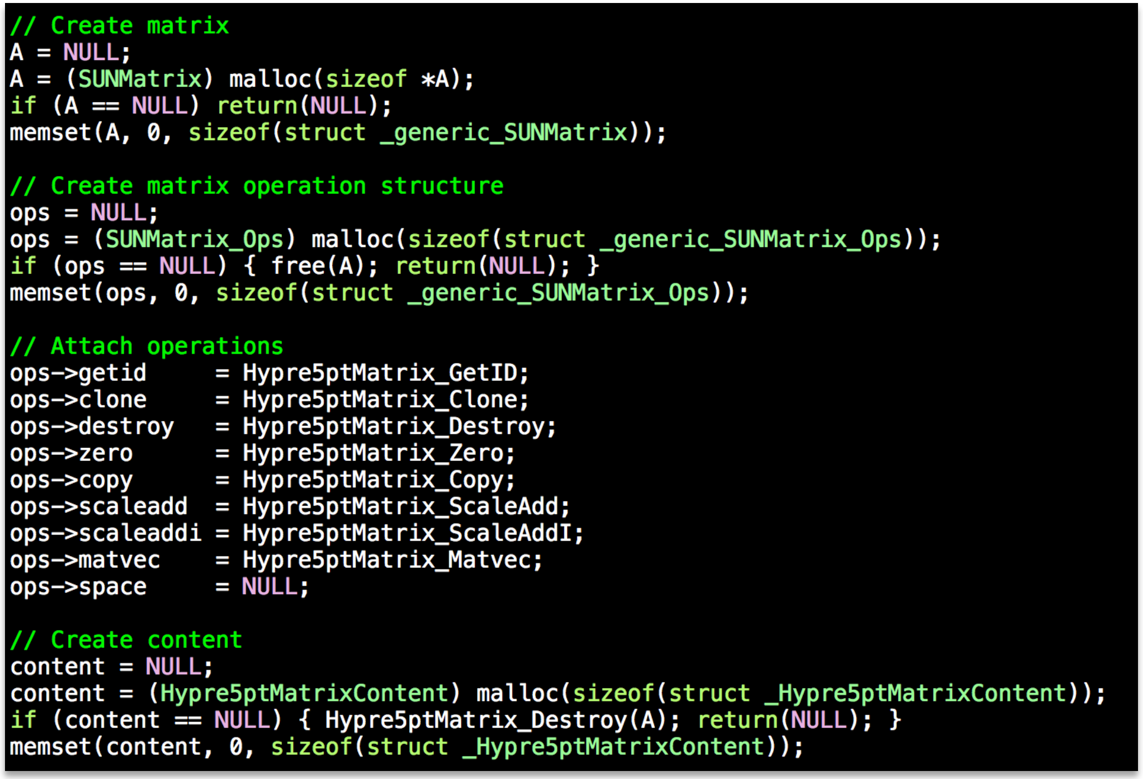
```
typedef struct _Hypre5ptMatrixContent {
  MPI_Comm              *comm;
  HYPRE_Int              ilower[2], iupper[2];
  HYPRE_StructGrid       grid;
  HYPRE_StructStencil    stencil;
  HYPRE_StructMatrix     matrix;
  HYPRE_Real            *work;
  HYPRE_Int              nwork;
} *Hypre5ptMatrixContent;
```

- Constructor to create a new matrix



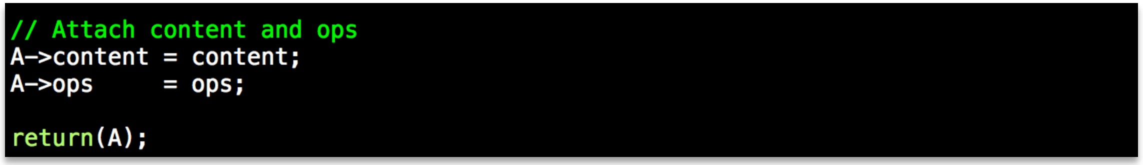
```
SUNMatrix Hypre5ptMatrix(MPI_Comm &comm,
                         sunindextype is, sunindextype ie,
                         sunindextype js, sunindextype je)
{
  SUNMatrix A;
  SUNMatrix_Ops ops;
  Hypre5ptMatrixContent content;
  HYPRE_Int offset[2];
  int ierr, result;
```

- Constructor continued



```
// Create matrix
A = NULL;
A = (SUNMatrix) malloc(sizeof *A);
if (A == NULL) return(NULL);
memset(A, 0, sizeof(struct _generic_SUNMatrix));

// Create matrix operation structure
ops = NULL;
ops = (SUNMatrix_Ops) malloc(sizeof(struct _generic_SUNMatrix_Ops));
if (ops == NULL) { free(A); return(NULL); }
memset(ops, 0, sizeof(struct _generic_SUNMatrix_Ops));

// Attach operations
ops->getid     = Hypre5ptMatrix_GetID;
ops->clone     = Hypre5ptMatrix_Clone;
ops->destroy   = Hypre5ptMatrix_Destroy;
ops->zero      = Hypre5ptMatrix_Zero;
ops->copy      = Hypre5ptMatrix_Copy;
ops->scaleadd  = Hypre5ptMatrix_ScaleAdd;
ops->scaleaddi = Hypre5ptMatrix_ScaleAddI;
ops->matvec    = Hypre5ptMatrix_Matvec;
ops->space     = NULL;

// Create content
content = NULL;
content = (Hypre5ptMatrixContent) malloc(sizeof(struct _Hypre5ptMatrixContent));
if (content == NULL) { Hypre5ptMatrix_Destroy(A); return(NULL); }
memset(content, 0, sizeof(struct _Hypre5ptMatrixContent));
```

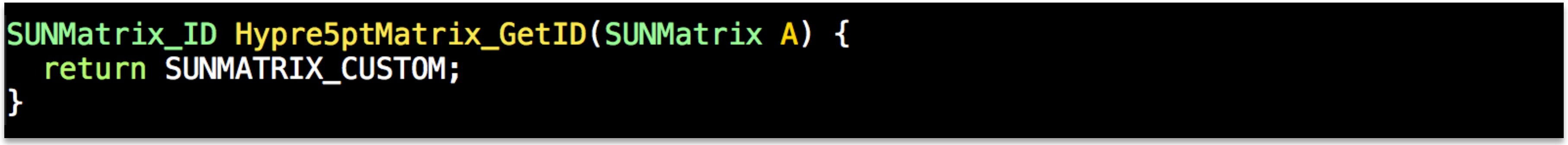
```
// Fill content
```



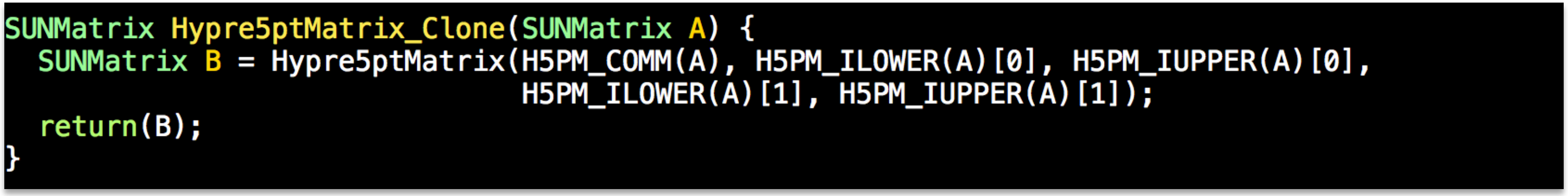
```
// Attach content and ops
A->content = content;
A->ops     = ops;

return(A);
```

examples/arkode/CXX_parhyp/ark_heat_2D_hypre.cpp

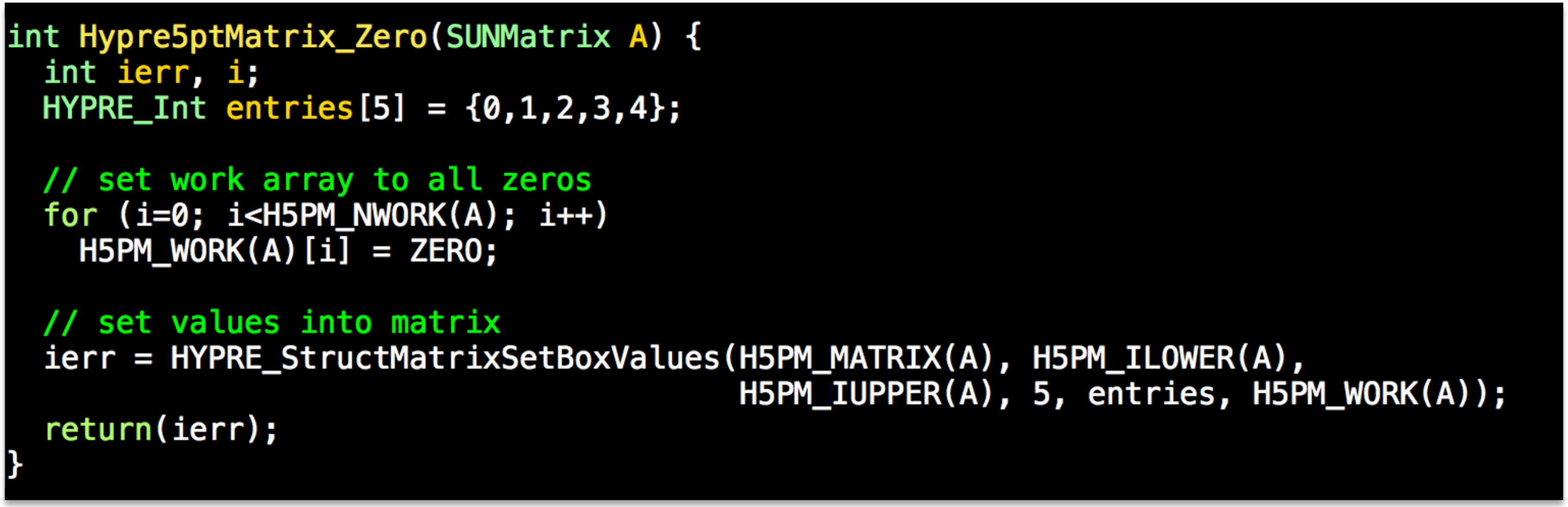
- Examples of some matrix operation implementations



```
SUNMatrix_ID Hypre5ptMatrix_GetID(SUNMatrix A) {
  return SUNMATRIX_CUSTOM;
}
```



```
SUNMatrix Hypre5ptMatrix_Clone(SUNMatrix A) {
  SUNMatrix B = Hypre5ptMatrix(H5PM_COMM(A), H5PM_ILOWER(A)[0], H5PM_IUPPER(A)[0],
                               H5PM_ILOWER(A)[1], H5PM_IUPPER(A)[1]);
  return(B);
}
```



```
int Hypre5ptMatrix_Zero(SUNMatrix A) {
  int ierr, i;
  HYPRE_Int entries[5] = {0,1,2,3,4};

  // set work array to all zeros
  for (i=0; i<H5PM_NWORK(A); i++)
    H5PM_WORK(A)[i] = ZERO;

  // set values into matrix
  ierr = HYPRE_StructMatrixSetBoxValues(H5PM_MATRIX(A), H5PM_ILOWER(A),
                                        H5PM_IUPPER(A), 5, entries, H5PM_WORK(A));
  return(ierr);
}
```

Creating a SUNLinearSolver Wrapper – Core Functions

- **`Constructor`** – creates a linear solver object and performs memory allocation as needed.

- **`GetType`** – returns the linear solver type.

- **`Initialize`** – initializes the linear solver and performs additional allocation as needed.

- **`Setup`** – called infrequently to update the Jacobian or preconditioner information.

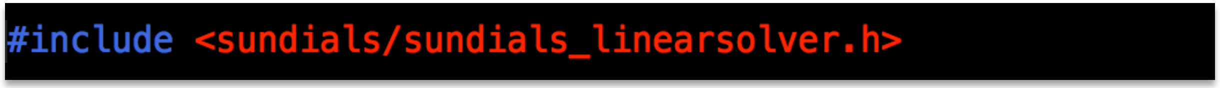
- **`Solve`** – solves the linear system $Ax=b$.

- `Free` – frees any memory allocated by the linear solver.

Key:
`Always required`
`Sometimes required`
`Optional`

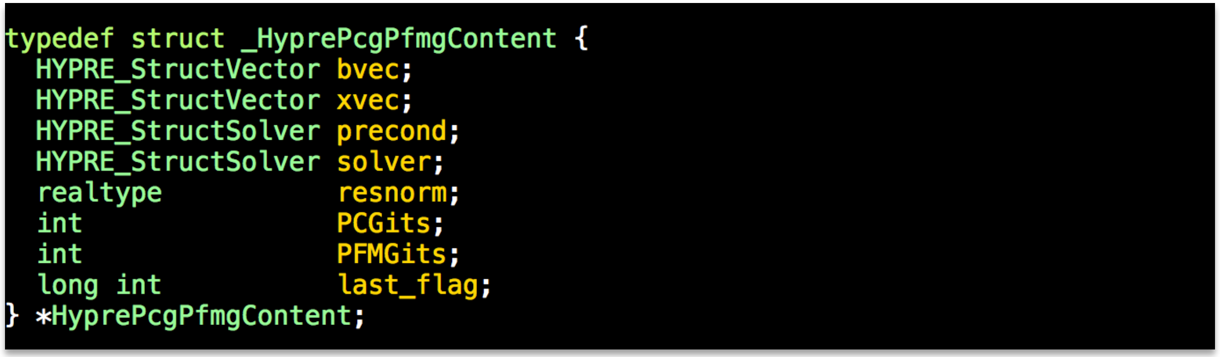
examples/arkode/CXX_parhyp/ark_heat_2D_hypre.cpp

- Header defining a generic `SUNLinearSolver`



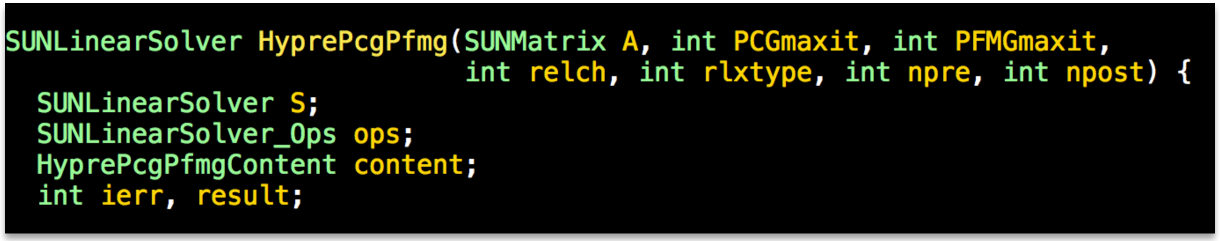
```
#include <sundials/sundials_linearsolver.h>
```

- Linear solver specific content structure



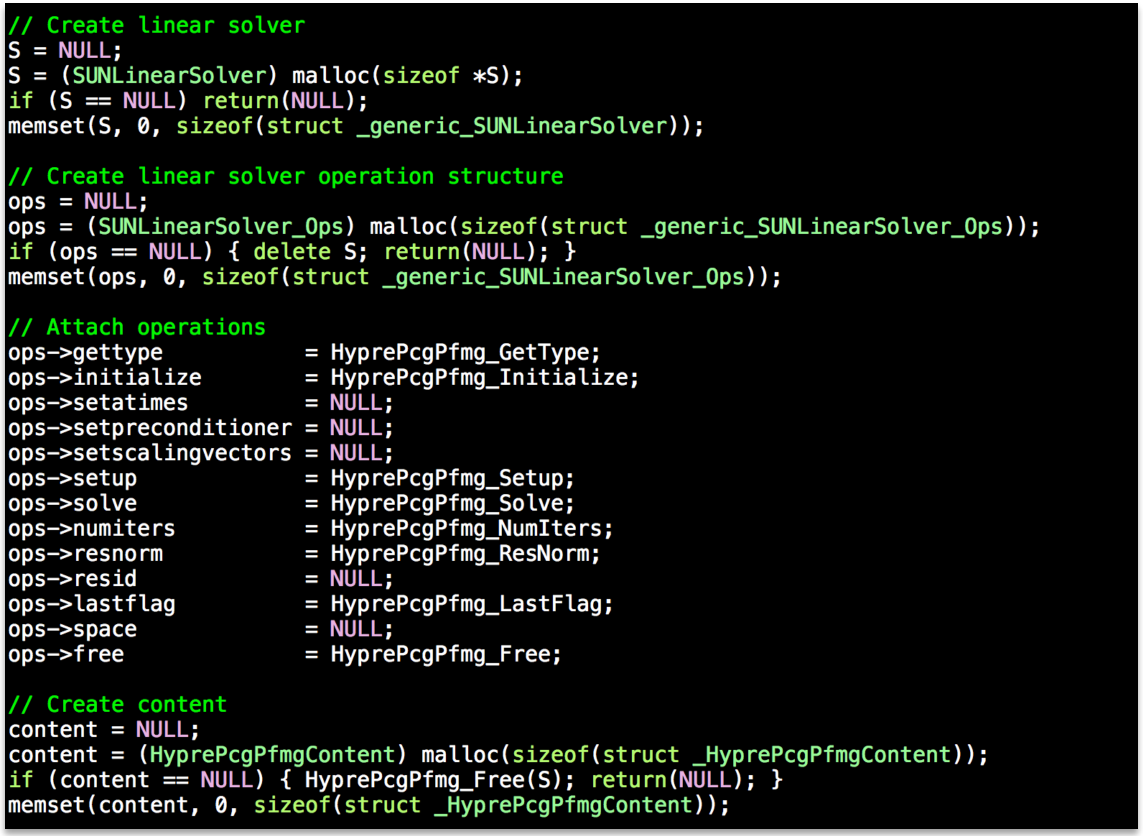
```
typedef struct _HyprePcgPfmgContent {
  HYPRE_StructVector bvec;
  HYPRE_StructVector xvec;
  HYPRE_StructSolver precond;
  HYPRE_StructSolver solver;
  realtype           resnorm;
  int                PCGits;
  int                PFMGits;
  long int           last_flag;
} *HyprePcgPfmgContent;
```

- Constructor to create a new linear solver



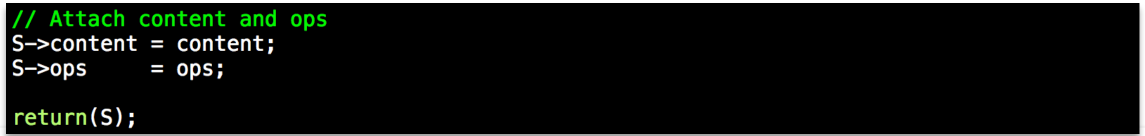
```
SUNLinearSolver HyprePcgPfmg(SUNMatrix A, int PCGmaxit, int PFMGmaxit,
                             int relch, int rlxtype, int npre, int npost) {
  SUNLinearSolver S;
  SUNLinearSolver_Ops ops;
  HyprePcgPfmgContent content;
  int ierr, result;
```

- Constructor continued



```
// Create linear solver
S = NULL;
S = (SUNLinearSolver) malloc(sizeof *S);
if (S == NULL) return(NULL);
memset(S, 0, sizeof(struct _generic_SUNLinearSolver));

// Create linear solver operation structure
ops = NULL;
ops = (SUNLinearSolver_Ops) malloc(sizeof(struct _generic_SUNLinearSolver_Ops));
if (ops == NULL) { delete S; return(NULL); }
memset(ops, 0, sizeof(struct _generic_SUNLinearSolver_Ops));

// Attach operations
ops->gettype             = HyprePcgPfmg_GetType;
ops->initialize          = HyprePcgPfmg_Initialize;
ops->setatimes           = NULL;
ops->setpreconditioner   = NULL;
ops->setscalingvectors   = NULL;
ops->setup               = HyprePcgPfmg_Setup;
ops->solve               = HyprePcgPfmg_Solve;
ops->numiters            = HyprePcgPfmg_NumIters;
ops->resnorm             = HyprePcgPfmg_ResNorm;
ops->resid               = NULL;
ops->lastflag            = HyprePcgPfmg_LastFlag;
ops->space               = NULL;
ops->free                = HyprePcgPfmg_Free;

// Create content
content = NULL;
content = (HyprePcgPfmgContent) malloc(sizeof(struct _HyprePcgPfmgContent));
if (content == NULL) { HyprePcgPfmg_Free(S); return(NULL); }
memset(content, 0, sizeof(struct _HyprePcgPfmgContent));
```

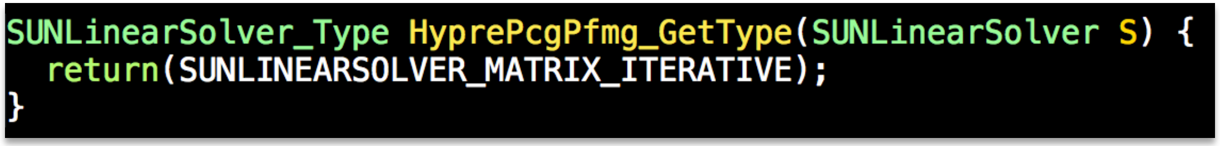
```
// Fill content
```



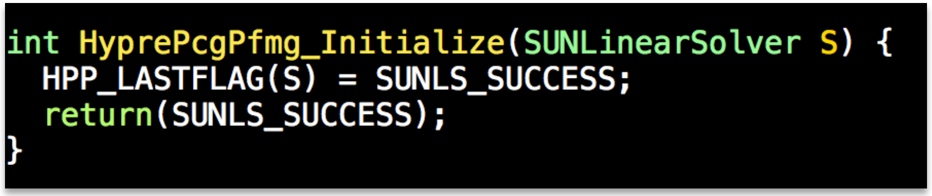
```
// Attach content and ops
S->content = content;
S->ops     = ops;

return(S);
```

examples/arkode/CXX_parhyp/ark_heat_2D_hypre.cpp

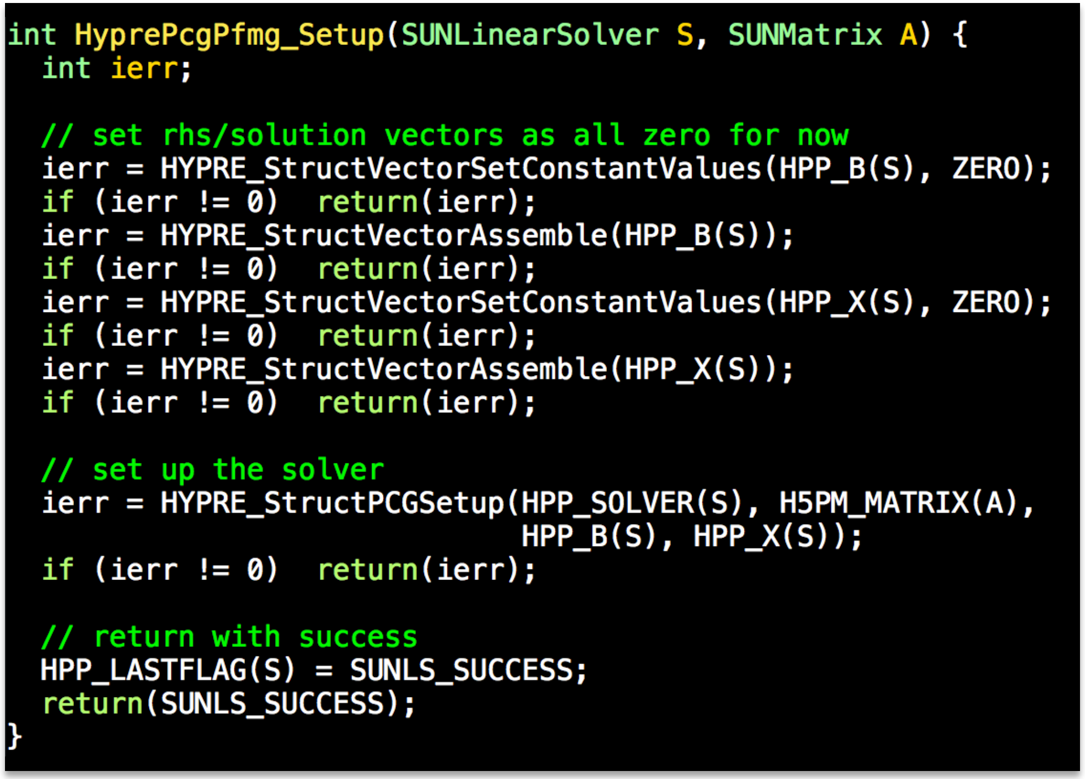
- Examples of linear solver operation implementations (some details omitted; see code for complete functions)



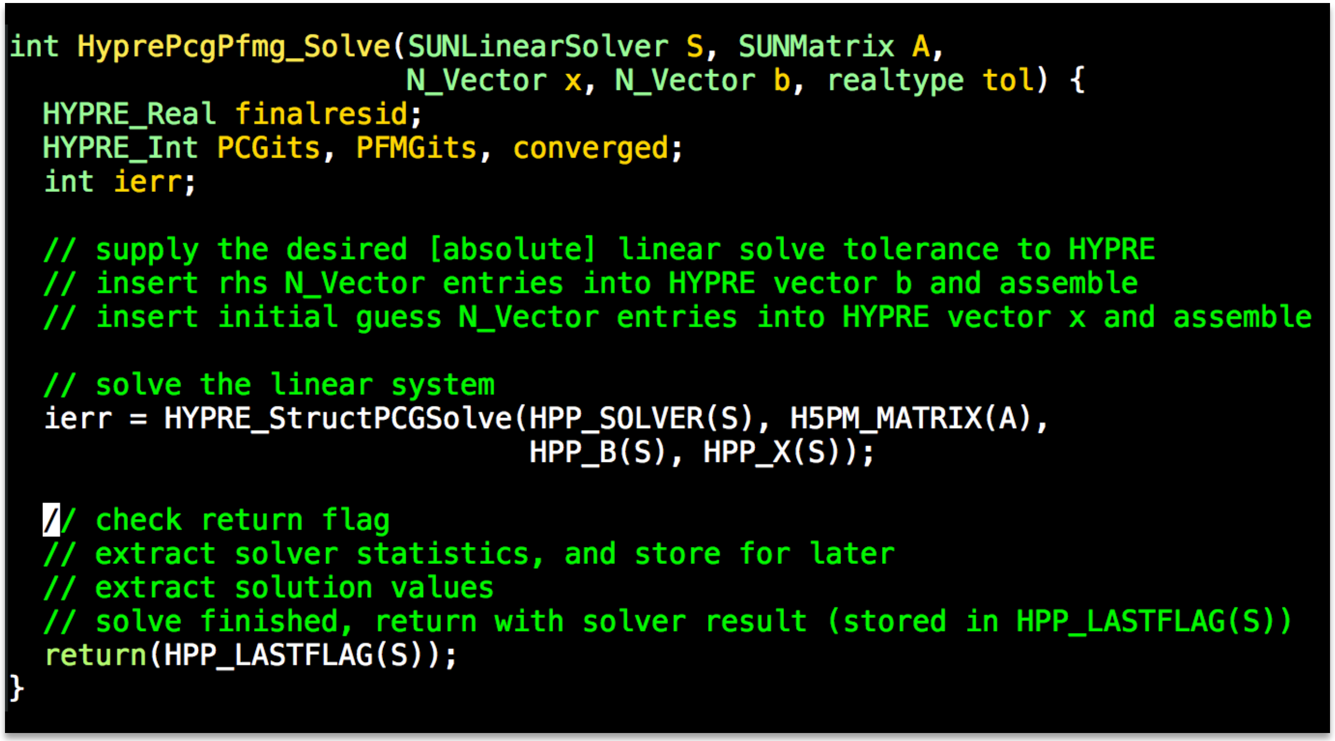
```
SUNLinearSolver_Type HyprePcgPfmg_GetType(SUNLinearSolver S) {
  return(SUNLINEARSOLVER_MATRIX_ITERATIVE);
}
```



```
int HyprePcgPfmg_Initialize(SUNLinearSolver S) {
  HPP_LASTFLAG(S) = SUNLS_SUCCESS;
  return(SUNLS_SUCCESS);
}
```



```
int HyprePcgPfmg_Setup(SUNLinearSolver S, SUNMatrix A) {
  int ierr;

  // set rhs/solution vectors as all zero for now
  ierr = HYPRE_StructVectorSetConstantValues(HPP_B(S), ZERO);
  if (ierr != 0)  return(ierr);
  ierr = HYPRE_StructVectorAssemble(HPP_B(S));
  if (ierr != 0)  return(ierr);
  ierr = HYPRE_StructVectorSetConstantValues(HPP_X(S), ZERO);
  if (ierr != 0)  return(ierr);
  ierr = HYPRE_StructVectorAssemble(HPP_X(S));
  if (ierr != 0)  return(ierr);

  // set up the solver
  ierr = HYPRE_StructPCGSetup(HPP_SOLVER(S), H5PM_MATRIX(A),
                              HPP_B(S), HPP_X(S));
  if (ierr != 0)  return(ierr);

  // return with success
  HPP_LASTFLAG(S) = SUNLS_SUCCESS;
  return(SUNLS_SUCCESS);
}
```



```
int HyprePcgPfmg_Solve(SUNLinearSolver S, SUNMatrix A,
                       N_Vector x, N_Vector b, realtype tol) {
  HYPRE_Real finalresid;
  HYPRE_Int PCGits, PFMGits, converged;
  int ierr;

  // supply the desired [absolute] linear solve tolerance to HYPRE
  // insert rhs N_Vector entries into HYPRE vector b and assemble
  // insert initial guess N_Vector entries into HYPRE vector x and assemble

  // solve the linear system
  ierr = HYPRE_StructPCGSolve(HPP_SOLVER(S), H5PM_MATRIX(A),
                              HPP_B(S), HPP_X(S));

  // check return flag
  // extract solver statistics, and store for later
  // extract solution values
  // solve finished, return with solver result (stored in HPP_LASTFLAG(S))
  return(HPP_LASTFLAG(S));
}
```

Creating a SUNLinearSolver Wrapper – Set Functions

- *`SetATimes`* – sets the function for computing Jacobian-vector products in iterative solvers.

```
int (*ATimesFn)(void *A_data, N_Vector v, N_Vector z)
```

- `SetPreconditioner` – sets the preconditioner setup and solve functions called by iterative or matrix-iterative solvers.

```
int (*PSetupFn)(void *P_data)

int (*PSolveFn)(void *P_data, N_Vector r, N_Vector z,
                realtype tol, int lr)
```

- `SetScalingVectors` – sets the scaling vectors used in iterative or matrix-iterative solvers.
  - SUNDIALS provided iterative linear solvers solve a transformed system:

$$\tilde{A}\tilde{x} = \tilde{b} \quad \begin{cases} \tilde{A} = S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} = S_1 P_1^{-1} b, \\ \tilde{x} = S_2 P_2 x, \end{cases}$$

Key:
`Always required`
`Sometimes required`
`Optional`

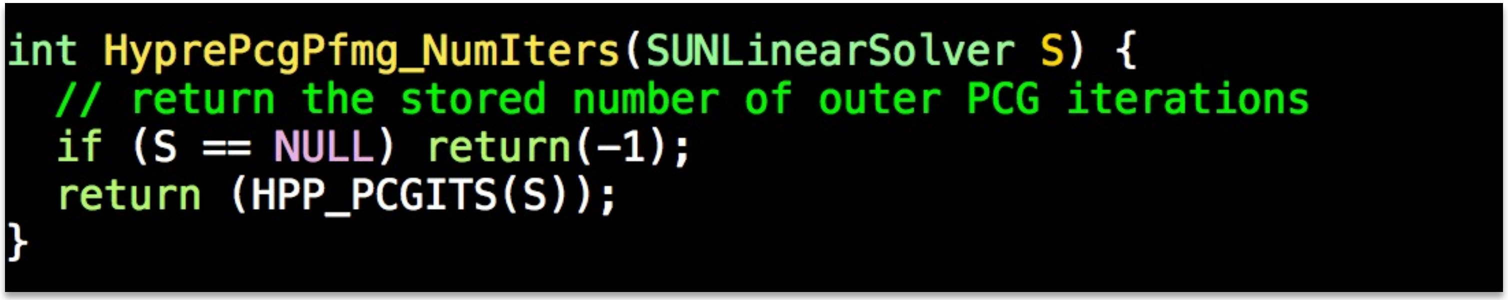
Creating a SUNLinearSolver Wrapper – Get Functions

- `NumIters` – returns the number of iterations in the last solve call.
- `ResNorm` – returns final residual norm from the last solve call.
- *`Resid`* – returns preconditioned initial residual vector.
- `LastFlag` – returns the last error flag encountered within the linear solver.
- `Space` – returns the storage requirements of the linear solver.

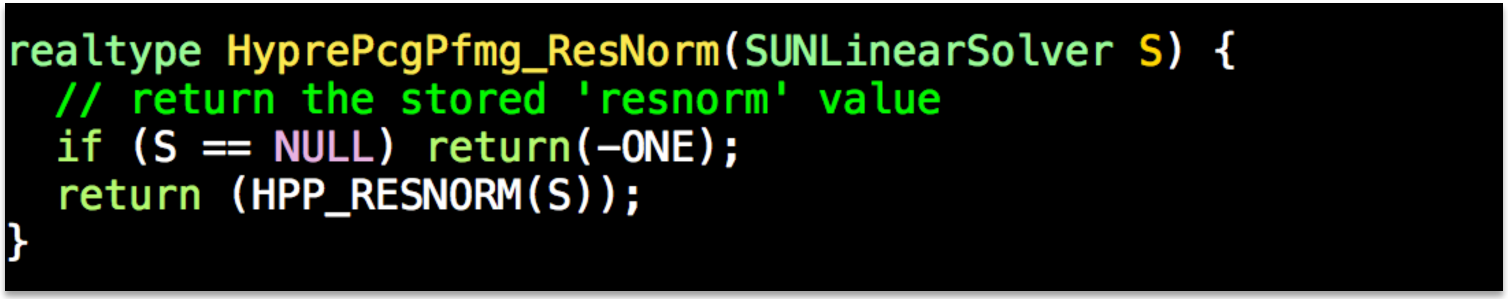
Key:
`Always required`
`Sometimes required`
`Optional`

examples/arkode/CXX_parhyp/ark_heat_2D_hypre.cpp

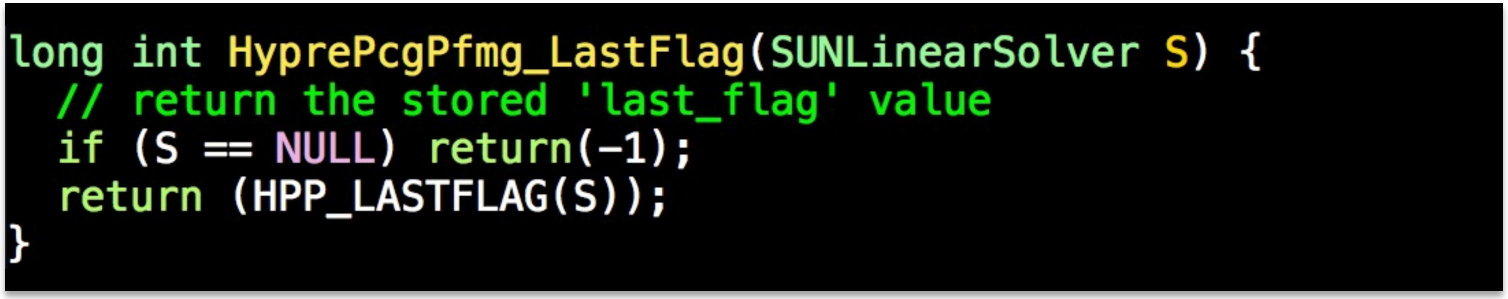
- Examples of linear solver get operation implementations



```
int HyprePcgPfmg_NumIters(SUNLinearSolver S) {
  // return the stored number of outer PCG iterations
  if (S == NULL) return(-1);
  return (HPP_PCGITS(S));
}
```



```
realtype HyprePcgPfmg_ResNorm(SUNLinearSolver S) {
  // return the stored 'resnorm' value
  if (S == NULL) return(-ONE);
  return (HPP_RESNORM(S));
}
```



```
long int HyprePcgPfmg_LastFlag(SUNLinearSolver S) {
  // return the stored 'last_flag' value
  if (S == NULL) return(-1);
  return (HPP_LASTFLAG(S));
}
```

The “Skeleton” for Using SUNDIALS Integrators

1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. **<u>Create matrix and linear solver objects; attach to integrator</u>**
   - Using the default Newton nonlinear solver
5. Specify optional inputs to integrator and solver objects (tolerances, etc.)
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Creating & Attaching the User-supplied Linear Solver

- In the “Usage Skeleton,” step 4 would consist of:

a) Create the SUNMatrix object

— `SUNMatrix A = MyNewMatrix(…)`

b) Create the SUNLinearSolver object

— `SUNLinearSolver LS = MyNewLinearSolver(…)`

c) Attach the linear solver

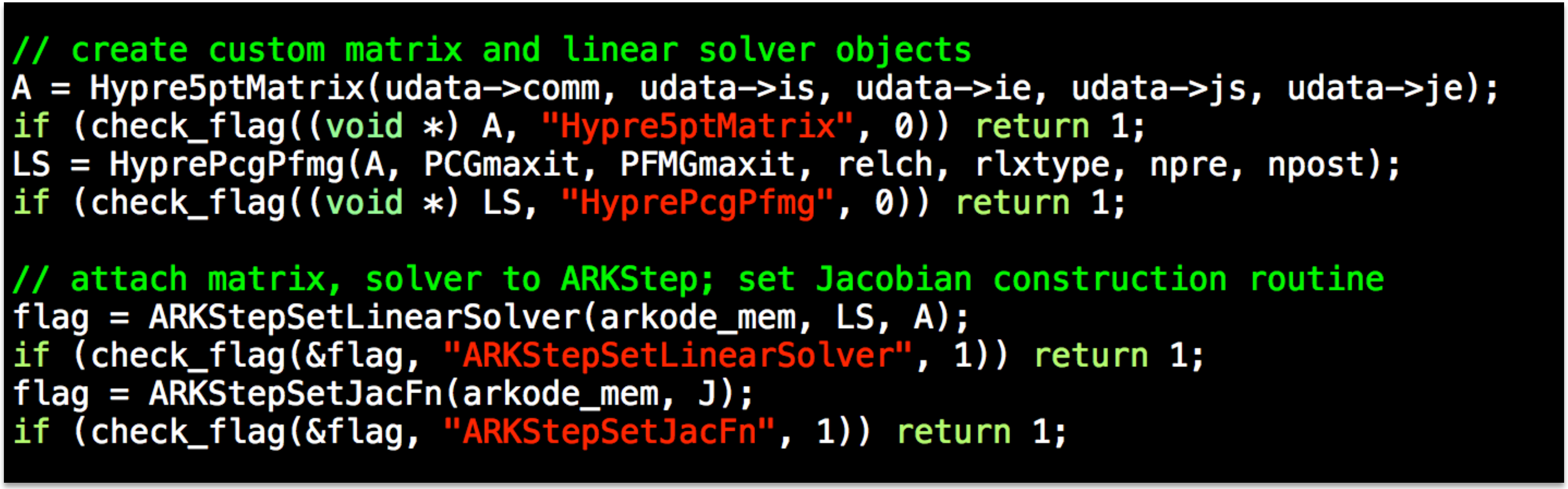
— `ier = *StepSetLinearSolver(mem, LS, A)`

d) Set the function to compute the Jacobian

— `ier = *StepSetJacFn(mem, J)`

examples/arkode/CXX_parhyp/ark_heat_2D_hypre.cpp

- Example using *hypre* structured matrix, linear solver (PCG), and preconditioner (PFMG).



```
// create custom matrix and linear solver objects
A = Hypre5ptMatrix(udata->comm, udata->is, udata->ie, udata->js, udata->je);
if (check_flag((void *) A, "Hypre5ptMatrix", 0)) return 1;
LS = HyprePcgPfmg(A, PCGmaxit, PFMGmaxit, relch, rlxtype, npre, npost);
if (check_flag((void *) LS, "HyprePcgPfmg", 0)) return 1;

// attach matrix, solver to ARKStep; set Jacobian construction routine
flag = ARKStepSetLinearSolver(arkode_mem, LS, A);
if (check_flag(&flag, "ARKStepSetLinearSolver", 1)) return 1;
flag = ARKStepSetJacFn(arkode_mem, J);
if (check_flag(&flag, "ARKStepSetJacFn", 1)) return 1;
```

Fixed Point Solver

- With CVODE and ARKODE (when M = I) the nonlinear systems can also be written as:

$$\left.\begin{array}{ll} \text{– CVODE:} & h_n \beta_{n,0} f(t_n, y^n) + a_n = y^n \\ \text{– ARKODE:} & h_n A^I_{i,i} f_I(t^I_{n,i}, z_i) + a_i = z_i \end{array}\right\} G(y) = y$$

These can both be posed as a generic fixed-point problem

- Users can elect to use a fixed point method to solve $G(y) = y$.
  - Jacobian information and a linear solver are not required in this case
  - Convergence can be accelerated using Anderson’s method

The “Skeleton” for Using SUNDIALS Integrators

1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. **<u>Create nonlinear solver object; attach to integrator</u>**
   - Using the Anderson accelerated fixed point solver
5. Specify optional inputs to integrator and solver objects (tolerances, etc.)
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Fixed Point Solver

- In the “Usage Skeleton,” step 4 would consist of:

a) Create the SUNNonlinearSolver object

— `SUNNonlinearSolver NLS = SUNNonlinSol_FixedPoint(y, m)`

b) Attach the nonlinear solver (* is the integrator prefix)

— `flag = *SetNonlinearSolver(mem, NLS)`

The “Skeleton” for Using SUNDIALS Integrators

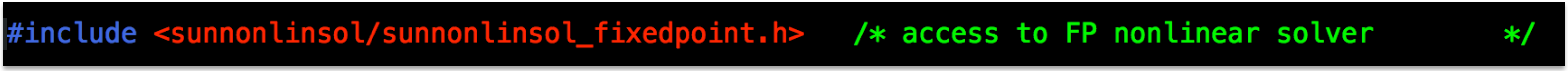
1. Initialize parallel or multi-threaded environment
2. Create vector of initial values, $y_0 \in \mathbb{R}^N$; if using IDA, also create $\dot{y}_0 \in \mathbb{R}^N$
3. Create and initialize integrator object (attaches $t_0, y_0, (\dot{y}_0)$, RHS/residual function(s))
4. Create nonlinear solver object; attach to integrator
5. **Specify optional inputs to integrator and solver objects (tolerances, etc.)**
6. Advance solution in time, either over specified time intervals *[a,b]*, or for single timesteps
7. Retrieve optional outputs
8. Free solution/solver memory; finalize MPI (if applicable)

Nonlinear Solver Options

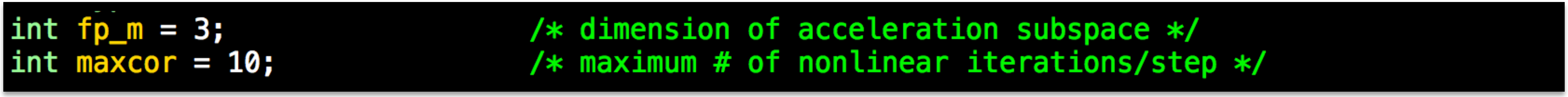
- `SetMaxNonlinIters` – sets the maximum number of nonlinear iterations.
- `SetNonlinConvCoef` – specifies the scaling factor used to set the nonlinear solver tolerance.
- Additional ARKODE options:
  - `SetNonlinear` – specifies if the implicit system nonlinear/linear.
  - `SetNonlinCRDown` – sets the nonlinear convergence rate constant.
  - `SetNonlinRDiv` – sets the nonlinear divergence ratio.

examples/arkode/C_serial/ark_brusselator_fp.c

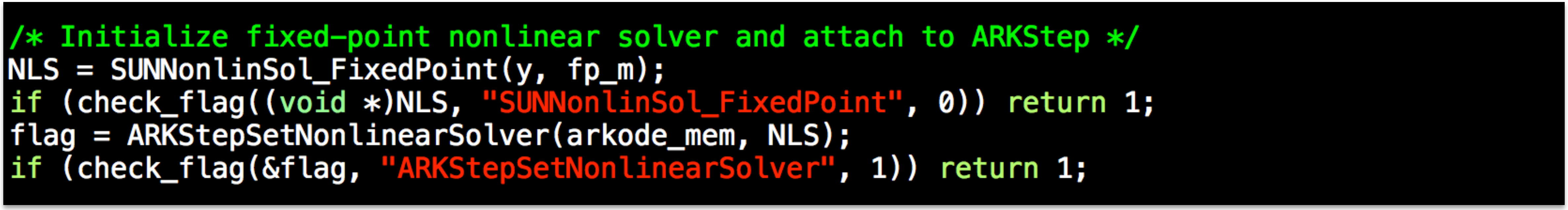
- Example using Anderson accelerated fixed point solver with non-default max iterations.



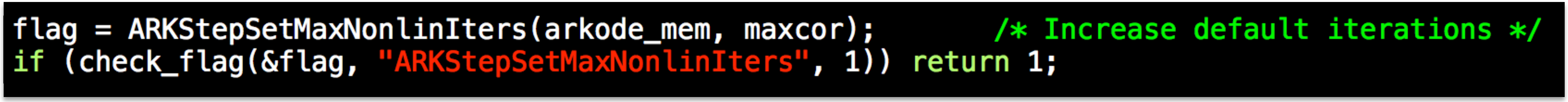
```c
#include <sunnonlinsol/sunnonlinsol_fixedpoint.h>   /* access to FP nonlinear solver        */
```



```c
int fp_m = 3;                  /* dimension of acceleration subspace */
int maxcor = 10;               /* maximum # of nonlinear iterations/step */
```



```c
/* Initialize fixed-point nonlinear solver and attach to ARKStep */
NLS = SUNNonlinSol_FixedPoint(y, fp_m);
if (check_flag((void *)NLS, "SUNNonlinSol_FixedPoint", 0)) return 1;
flag = ARKStepSetNonlinearSolver(arkode_mem, NLS);
if (check_flag(&flag, "ARKStepSetNonlinearSolver", 1)) return 1;
```



```c
flag = ARKStepSetMaxNonlinIters(arkode_mem, maxcor);        /* Increase default iterations */
if (check_flag(&flag, "ARKStepSetMaxNonlinIters", 1)) return 1;
```

Creating a SUNNonlinearSolver Wrapper – Core Functions

- **`Constructor`** – creates a nonlinear solver object and performs memory allocation as needed.
- **`GetType`** – return the solver type, `ROOTFIND`, for $F(y) = 0$ and `FIXEDPOINT` for $G(y) = y$.
- `Initialize` – initializes the nonlinear solver and performs additional allocation as needed.
- `Setup` – called before each step attempt to perform any nonlinear solver setup.
- **`Solve`** – solve the nonlinear system $F(y) = 0$ or $G(y) = y$.
- `Free` – frees any memory allocated by the nonlinear solver.

Key:
`Always required`
`Sometimes required`
`Optional`

Creating a SUNNonlinearSolver Wrapper – Set and Get Functions

- **SetSysFn** – allows the integrator to provide the nonlinear system function $F(y)$ or $G(y)$.
- *SetConvTestFn* – sets the nonlinear iteration convergence test function.
- SetMaxIters – sets the maximum number of iterations.
- GetNumIters – returns the total number of nonlinear iterations.
- *GetCurIter* – returns the current iteration number.
- GetNumConvFails – returns the number of convergence failures.

Key:
Always required
Sometimes required
Optional

Creating a SUNNonlinearSolver Wrapper – Linear Solver Interface

- If the nonlinear solver uses a SUNDIALS linear solver, then following functions are required.

- *SetLSetupFn* – allows the integrator to attach the linear solver setup function to the nonlinear solver

```
int (*LSetupFn)(N_Vector y, N_Vector F, booleantype jbad,
                booleantype* jcur, void* mem)
```

- *SetLSolveFn* – allows the integrator to attach the linear solver solve function to the nonlinear solver

```
int (*LSolveFn)(N_Vector y, N_Vector b, void* mem)
```

Key:
Always required
Sometimes required
Optional

CASC

Center for Applied
Scientific Computing

SMU®

Lawrence Livermore
National Laboratory

Disclaimer

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