SUNDIALS Capabilities and Usage

APP-FPP Collaboration

September 2025

Daniel R. Reynolds (UMBC), Yifan Hu (UMBC), David J. Gardner (LLNL)

U.S. DEPARTMENT OF ENERGY | Office of Science

SciDAC
Scientific Discovery through Advanced Computing

Lawrence Livermore National Laboratory

UMBC

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. LLNL-PRES-2012257

Outline

- **Overview of SUNDIALS**
- How to use the time integrators from modern Fortran
- Interfacing SUNDIALS with existing codes
- Closing remarks

sundials

Suite of Nonlinear and Differential-Algebraic Solvers

- Library of time integrators and nonlinear solvers
  - Packages: CVODE(S), ARKODE, IDA(S), and KINSOL
  - Written in C with modern Fortran interfaces
  - Designed to be easily incorporated into existing codes
- Modular implementation
  - Data manipulation and parallelism are encapsulated by vector, matrix, and solver classes and user-supplied callback functions
  - Native class implementations for serial, threaded, distributed, and GPU computing platforms
  - Vector, matrix, and solver classes can all be user-supplied
- Freely available under BSD 3-Clause license
  - 100,000+ downloads per year: github.com/LLNL/sundials
  - Detailed user manuals: sundials.readthedocs.io
  - Active user community supported by email list and GitHub issues

Pele Combustion Simulations on Frontier

Isosurfaces of diesel fuel entering a turbulent methane-air premixture. High temperature pockets (red and yellow) form when local kernels of diesel fuel ignite.

This simulation ran on 7,000 Frontier nodes at OLCF using SUNDIALS to solve the chemistry systems in every grid cell in a 7-layer adaptive mesh hierarchy (60B grid cells and approximately 2.4T degrees of freedom).

Courtesy of Marc Day and Jon Rood (NREL). Animation by Nicholas Brunhart-Lupo (NREL).

computing.llnl.gov/sundials

Adaptive Methods for Ordinary Differential Equation (ODE) and Differential-Algebraic Equation (DAE) Initial Value Problems (IVPs)

- **CVODE:** adaptive order and step size linear multistep methods for ODEs, $y' = f(t, y)$
  - Adams methods for non-stiff systems and BDF methods for stiff systems
- **IDA:** adaptive order and step size BDF methods for DAEs, $F(t, y, \dot{y}) = 0$
  - Targets implicit ODEs, index-1 DAEs, and Hessenberg index-2 DAEs
- **CVODES** and **IDAS** support forward and adjoint sensitivity analysis (user-suppled adjoint operator)
- **ARKODE:** infrastructure for adaptive step multistage methods

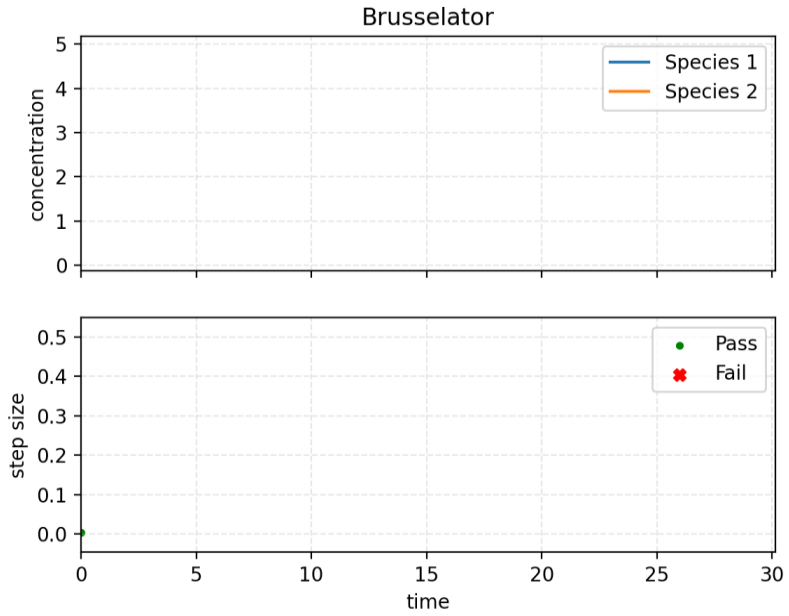
Stepper	Methods	Systems
ARKStep	ERK, DIRK, and IMEX	$M(t)\ y' = f^E(t,y) + f^I(t,y)$
ERKStep	ERK (streamlined module)	$y' = f(t,y)$
LSRKStep	Low-storage SSP and STS	$y' = f(t,y)$
MRIStep	Multirate infinitesimal (MRI) step	$y' = f^S(t,y) + f^F(t,y)$
SPRKStep	Symplectic partitioned RK	$p' = f^p(t,q), \quad q' = f^q(t,p)$
SplittingStep	Operator splitting	$y' = f^1(t,y) + \cdots + f^N(t,y)$

Adaptive Methods Minimize Local Error and Maximize Efficiency

- Vary the time step size to meet a user defined accuracy in each step
  - CVODE(S) and IDA(S) additionally adjust the method order
- Relative tolerance ($rtol$) controls error relative to the solution size
  - $rtol$ = 10^{-4} means errors are controlled to 0.01%
- Absolute tolerances ($atol_i$) control error of small components
  - Ex: if a solution component starts at a nonzero value but decays to a noise level, $atol_i$ should be set to the noise level
- Estimate the step error, $E(\Delta t)$; accept the step if $||E(\Delta t)||_{WRMS} < 1$

$$||v||_{WRMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(w_i v_i)^2} \qquad w_i = \frac{1}{rtol|y_i| + atol_i}$$

- If the test fails, reduced the step and try again; otherwise choose the next step, $\Delta t'$, so that $||E(\Delta t')||_{WRMS}$ is expected to be small
- The aim is to maximize step sizes, minimize failed steps, and maintain smooth step size transitions



Adaptivity can give much more accurate and efficient results than a fixed step size.

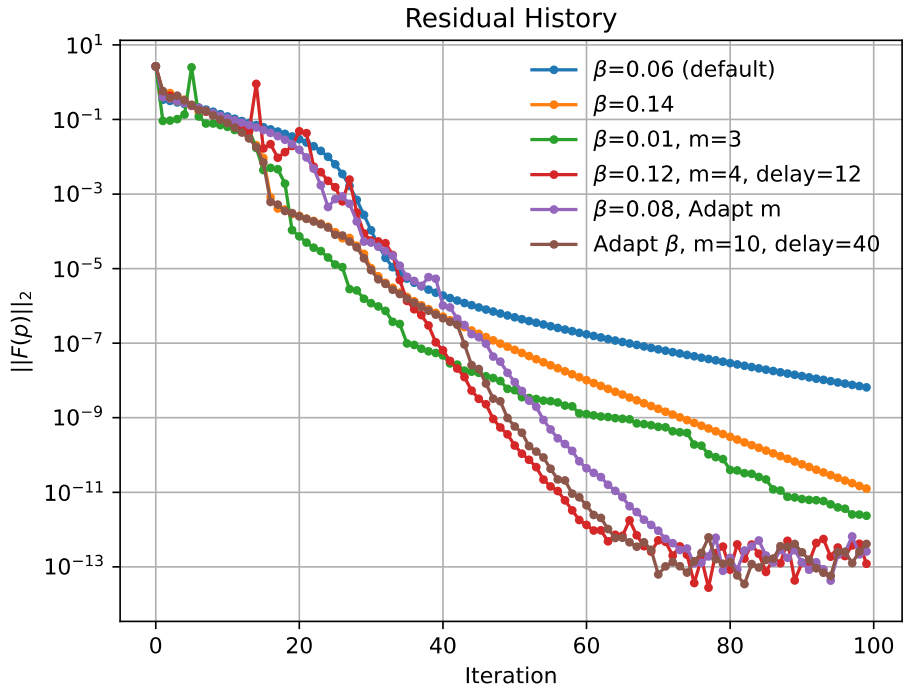
This chemical reaction problem illustrates how the time integrator dynamically adjusts the step size (lower plot) to capture the changing problem dynamics (upper plot).

KINSOL: Solvers for Systems of Nonlinear Algebraic Equations

- **Newton methods**: $u^{k+1} = u^k + \lambda\delta^k$
  - Get update by solving $J(u^{k^*})\delta^k = -F(u^k)$ where $J(u) = \frac{\partial F(u)}{\partial u}$
    - *Modified Newton* ⇒ reuse J evaluations across iterations
      - Residual monitoring and heuristic controls for updating J
    - *Inexact Newton* ⇒ approximately solve the linear system
      - Dynamic tolerances: $||F(u^k) + J(u^k)\delta^k|| \leq \eta^k ||F(u^k)||$
  - Can separately scale equations and unknowns
  - Backtracking and line search options for robustness
- **Fixed-point iteration**: $u^{k+1} = \beta_k G(u^k)$
- **Picard iteration**: $u^{k+1} = u^k - L^{-1}F(u^k)$ where $F(u) \equiv Lu - N(u)$
- **Anderson acceleration (AA)**: can improve fixed-point and Picard convergence speed and robustness

$$u_{n+1} = \beta_k \sum_{i=0}^{m_k} \alpha_i^k \, G(u_{k-m_k+i}) + (1-\beta_k) \sum_{i=0}^{m_k} \alpha_i^n \, u_{k-m_k+i}$$

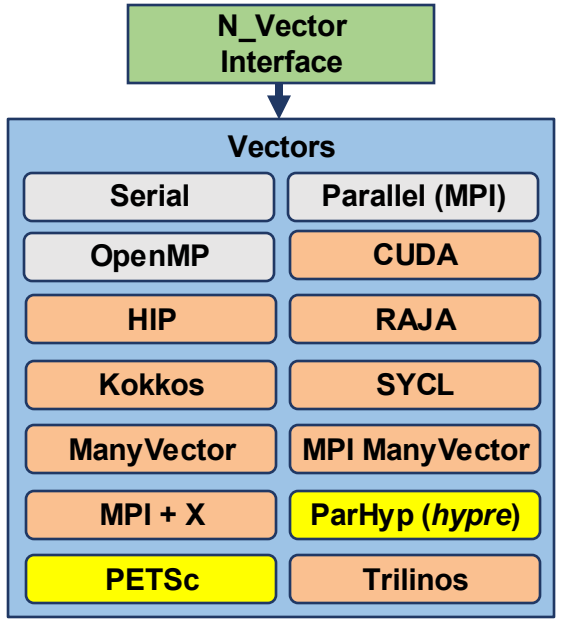
where α_i^k minimizes $\|F_k \alpha^T\|_2$ and $\sum_{i=0}^{m_k} \alpha_i = 1$

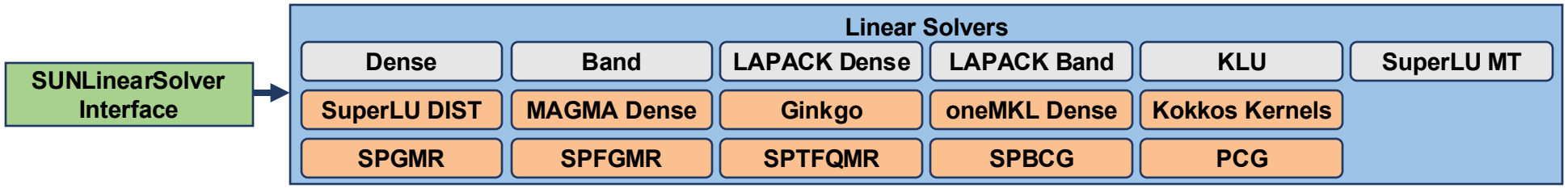


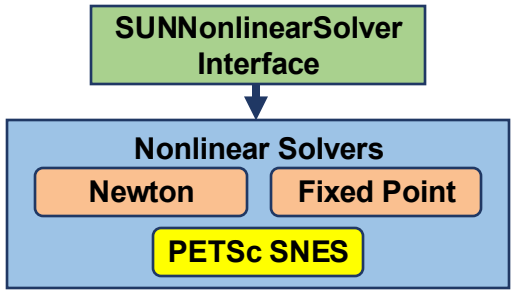
Residual history from a nonlinear diffusion equation with different fixed-point iteration configurations: fixed damping, optimized damping, and fixed or adaptive damping and AA depth. AA can improve convergence, increase robustness, and reduce sensitivity.

SUNDIALS Packages Are Written Using a Common Set of Abstract Base Classes

- Class interfaces encapsulate SUNDIALS algorithms from the data structures and parallelization employed by an implementation:
  - Provides flexibility for interfacing with new nonlinear and linear solvers
  - Simplifies porting to new architectures
  - Enables application-defined data structures and solvers
- Support for NVIDIA, AMD, and Intel GPUs
  - On-node GPU vectors: CUDA, HIP, SYCL, RAJA, and Kokkos
  - MPI distributed vectors: Parallel (MPI), ParHyp (hypre), PETSc, and Trilinos
  - The ManyVector and MPIPlusX vectors support hybrid computation







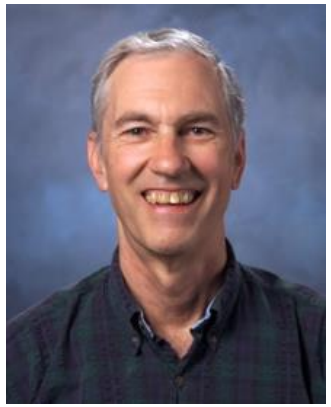
GPU-capable modules Planned GPU support

Current SUNDIALS Team

Current Team:


Cody Balos


David Gardner


Alan Hindmarsh


Dan Reynolds

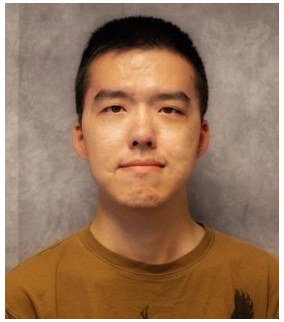

Steven Roberts


Carol Woodward

Postdocs:


Mustafa Aggul


Sylvia Amihere


Yifan Hu

Alumni:


Radu Serban

Scott D. Cohen, Peter N. Brown, George Byrne, Allan G. Taylor, Steven L. Lee, Keith E. Grant, Aaron Collier, Lawrence E. Banks, Steve G. Smith, Cosmin Petra, Homer Walker, Slaven Peles, John Loffeld, Dan Shumaker, Ulrike M. Yang, James Almgren-Bell, Shelby L. Lockhart, Rujeko Chinomona, Daniel McGreer, Hunter Schwartz, Hilari C. Tiedeman, Ting Yan, Jean M. Sexton, and Chris White

Outline

- Overview of SUNDIALS
- **How to use the time integrators from modern Fortran**
- Interfacing SUNDIALS with existing codes
- Closing remarks

C/C++ vs Fortran

While written in C, SUNDIALS fully supports applications written in C++ or modern Fortran:

- Interfaces are autogenerated with SWIG-Fortran
  - Leverages the iso_c_binding module and the bind(C) attribute from the Fortran 2003 standard
- The Fortran 2003 interfaces closely follow the C/C++ API
  - Packages and class implementations are accessed by using the corresponding Fortran module
  - Function names are prepended with an “F” (e.g., FN_VDotProd instead of N_VDotProd)
  - Constants are named exactly as they are in the C/C++ API
  - Classes (e.g., N_Vector)are interfaced as Fortran derived types allowing users to define their own implementations
- Accordingly, using SUNDIALS via the Fortran 2003 interfaces looks just like using it in C/C++

The SUNDIALS documentation thus primarily focuses on C/C++; however, since we believe that both APP-FPP collaborator codes are in Fortran, we focus on that interface here

The “Skeleton” for Using SUNDIALS Integrators

1. Create a SUNDIALS Context object
2. Create and fill the initial condition vector
3. Create and initialize the time integrator
4. Set the integrator tolerances
5. Create and attach solvers *(if necessary)*
6. Set integrator/solver inputs *(optional)*
7. Advance the solution in time
8. Get integrator/solver statistics *(optional)*
9. Free integrator and objects

```
use, intrinsic :: iso_c_binding
use fnvector_serial_mod
use farkode_arkstep_mod

integer(c_int) :: ierr
type(c_ptr) :: ctx
type(N_Vector), pointer :: y
real(c_double), pointer, dimension(nlocal) :: ydata(:)
type(c_ptr) :: ark

ierr = FSUNContext_Create(SUN_COMM_NULL, ctx)

y = FN_VNew_Serial(10, ctx)
ydata => FN_VGetArray(y)

ark = FARKStepCreate(c_funloc(f), c_null_funptr, 0.d0, y, ctx)

ierr = FARKodeSStolerances(ark, rtol, atol)

ierr = FARKodeEvolve(ark, tout, y, tret, ARK_NORMAL)

call FARKodeFree(ark)
call FN_VDestroy(y)
ierr = FSUNContext_Free(ctx)
```

Supplying the Initial Condition Vector(s) – the N_Vector API

- Each SUNDIALS-provided vector has a unique set of constructors:

```
! Include the C bindings and the relevant SUNDIALS modules
use, intrinsic :: iso_c_binding
use fnvector_serial_mod
use fnvector_parallel_mod

! Declare interface structures as pointers to their C types
type(N_Vector), pointer :: x, y
real(c_double), pointer, dimension(nlocal) :: ydata(:)

! Create SUNDIALS vectors
x = FN_VNew_Serial(neq, ctx)
y = FN_VNew_Parallel(comm, nlocal, neq, ctx)

! Use Fortran pointers to access the vector data
ydata => FN_VGetArray(y)
```

- The SUNDIALS-provided GPU vector modules are not currently supported through the Fortran interface
- Alternately, existing codes may “teach” SUNDIALS how to operate directly on their data *(more on this soon)*
- 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 IVP to the Integrator – RHS/Residual Functions

Example: ark_analytic_f2003.f90

Once problem data is encapsulated in a vector, all that remains for basic SUNDIALS usage is specifying the IVP

- CVODE and ARKODE specify the IVP through right-hand side function(s), $f(t, y)$

```
integer(c_int) function RhsFn(t, y, f, user_data) result(ierr) bind(C)
  use, intrinsic :: iso_c_binding
  implicit none
  real(c_double), value :: t          ! current time
  type(N_Vector)        :: y          ! solution vector
  type(N_Vector)        :: f          ! rhs vector
  type(c_ptr), value :: user_data     ! user-defined data
  real(c_double), pointer, dimension(1) :: ydata(:)
  real(c_double), pointer, dimension(1) :: fdata(:)

  ydata => FN_VGetArrayPointer(y)     ! access data arrays
  fdata => FN_VGetArrayPointer(f)
  fdata(1) = lambda * ydata(1)        ! fill RHS vector
  ierr = 0                            ! return status
  return
end function RhsFn
```

- user_data is a user-specified pointer for passing problem-specific data to callback functions (i.e., no global memory). This is mainly useful in C/C++, whereas Fortran users typically access such data through use statements
- For the return flag, ierr
  - 0 = success
  - >0 = recoverable error
  - <0 = fatal error

- IDA similarly specifies the IVP through a residual function, $F(t, y, \dot{y})$

```
integer(c_int) function ResFn(t, y, ydot, res, user_data) result(retval) bind(C)
```

Initializing the Integrators – CVODE and IDA

The initial conditions and IVP-defining functions are supplied when constructing the integrator.

CVODE

```
type(c_ptr) :: cvode_mem   ! CVODE memory
integer(c_int) :: retval

! Create BDF integrator
cvode_mem = FCVodeCreate(CV_BDF, ctx)
if (.not. c_associated(cvode_mem)) stop 1

! Initialize integrator with problem specification
retval = FCVodeInit(cvode_mem, c_funloc(RhsFn), T0, y)
if (retval /= 0) stop 1
```

IDA

```
type(c_ptr) :: ida_mem   ! IDA memory
integer(c_int) :: retval

! Create integrator
ida_mem = FIDACreate(ctx)
if (.not. c_associated(ida_mem)) stop 1

! Initialize integrator with problem specification
retval = FIDAInit(ida_mem, c_funloc(ResFn), T0, y, yp)
if (retval /= 0) stop 1
```

Initializing the Integrators – ARKODE

Create an **ImEx method** – for an implicit or explicit method, replace c_funloc(fe) or c_funloc(fi) with c_null_funptr, respectively

```
type(c_ptr) :: arkode_mem

arkode_mem = FARKStepCreate(c_funloc(fe), c_funloc(fi), T0, y, ctx) ! Create ImEx integrator
if (.not. c_associated(arkode_mem)) stop 1
```

Create a **multirate method** with ImEx methods at both time scales:

```
type(c_ptr) :: fast_mem, slow_mem, fast_stepper
integer(c_int) :: retval

fast_mem = FARKStepCreate(c_funloc(ffe), c_funloc(ffi), T0, y, ctx) ! Create fast integrator

! Set up fast integrator as normal…

retval = FARKStepCreateMRIStepInnerStepper(fast_mem, fast_stepper) ! Create fast stepper
if (retval /= 0) stop 1

slow_mem = FMRIStepCreate(c_funloc(fse), c_funloc(fsi), T0, y, inner_stepper, ctx) ! Create slow integrator
```

Supplying Options to the Integrators

After constructing the integrator, additional options may be supplied through various “Set” routines (example from ark_roberts_dns_f2003.f90, see the documentation for the full set of options):

```
real(c_double)  :: rtol, atol
integer(c_long) :: mxsteps
integer(c_int)  :: retval, nliters

! Set routines
rtol = 1.d-4
atol = 1.d-9
retval = FARKodeSStolerances(ark, rtol, atol) ! Scalar relative and absolute tolerances
if (retval /= 0) stop 1

retval = FARKodeSetJacFn(arkode_mem, c_funloc(jacrob))  ! Jacobian construction function
if (retval /= 0) stop 1

mxsteps = 10000
retval = FARKodeSetMaxNumSteps(arkode_mem, mxsteps)     ! Increase max steps per Evolve call
if (retval /= 0) stop 1

nliters = 8
retval = FARKodeSetMaxNonlinIters(arkode_mem, nliters)  ! Max Newton iterations
if (retval /= 0) stop 1
```

Advancing the Solution

- Once all options have been set, the integrator is called to advance the solution toward t_{out}
- The *direction* of integration is set on the first call (for forward integration $t_0 < t_{out}$)
- The solution is advanced using one of two *usage modes*:
  - **Normal** – take internal steps until t_{out} is overtaken then compute and return $y(t_{out})$ using interpolation
  - **One-step** – take a single step $t_{n-1} \rightarrow t_n$ and return $y(t_n)$ without interpolation
- When a stop time, t_{stop}, is set, the step size is limited to not overtake t_{stop} and the integrator returns $y(t_{stop})$ without interpolation

CVODE

```
real(c_double) :: tret(1)
retval = FCVode(cvode_mem, tout, y, tret(1), CV_NORMAL)
if (retval /= 0) stop 1
```

IDA

```
real(c_double) :: tret(1)
retval = FIDASolve(ida_mem, tout, tret(1), y, yp, tret(1), IDA_ONE_STEP)
if (retval /= 0) stop 1
```

ARKODE

```
real(c_double) :: tret(1)
retval = FARKodeEvolve(arkode_mem, tout, y, tret(1), ARK_NORMAL)
if (retval /= 0) stop 1
```

Retrieving Optional Outputs from the Integrators

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 via “Get” routines.

```
integer(c_long) :: nst(1), nfe(1), npre(1), npsol(1), &
                   nni(1), nli(1)
integer(c_int)  :: retval

retval = FCVodeGetNumSteps(cvode_mem, nst)
if (retval /= 0) stop 1

retval = FCVodeGetNumRhsEvals(cvode_mem, nfe)
if (retval /= 0) stop 1

retval = FCVodeGetNumPrecEvals(cvode_mem, npre)
if (retval /= 0) stop 1

retval = FCVodeGetNumPrecSolves(cvode_mem, npsol)
if (retval /= 0) stop 1

retval = FCVodeGetNumNonlinSolvIters(cvode_mem, nni)
if (retval /= 0) stop 1

retval = FCVodeGetNumLinIters(cvode_mem, nli)
if (retval /= 0) stop 1
```

```
! Get derivative at tret(1)
retval = FCVodeGetDky(cvode_mem, tret(1), 1, &
                      sun_dky)
if (retval /= 0) stop 1
```

Above: dense solution output from [cv_roberts_dns_f2003.f90]

Left: scalar-valued solver statistics from [cv_diag_kry_f2003.f90]

Advanced SUNDIALS Features

Additional features to examine auxiliary conditions, change the IVP, and improve solver efficiency:

- *Event detection (root-finding)* – find roots of a set of auxiliary user-defined functions $g_i\big(t, y(t)\big), i = 1, \ldots, N_r$; by monitoring sign changes between time steps.
- *Reinitialization* – reuse existing integrator memory for a "new" problem. All solution history and solver statistics are erased, but no memory is (de)allocated.
- *Constraint-handling* – positivity / negativity / non-positivity / non-negativity constraints may be set on individual solution components (handled through time step size adjustments).
- *Advanced error controllers* (ARKODE) – alternative step size selection algorithms.
- *Resizing* (ARKODE) – resize the problem and all internal vector memory, without destruction of temporal adaptivity heuristic information or solver statistics (useful for spatially adaptive problems).
- *Relaxation* (ARKODE) – adjusts the step size to help preserve a scalar-valued quantity of interest (e.g., total energy), without requiring use of special integrators or recomputing time steps.
- *Sensitivity Analysis* – computes sensitivities with respect to problem parameters (forward or adjoint).
- *Parallel-in-Time* (ARKODE) – interface to use multigrid reduction in time (MGRIT) methods with XBraid

Outline

- Overview of SUNDIALS
- How to use the time integrators from modern Fortran
- **Interfacing SUNDIALS with existing codes**
- Closing remarks

Custom Vector Interfaces

(Adapted from fnvector_fortran_mod.f90)

Consider a Fortran application that uses a 2D array for storing the solution. A simple module can wrap this as an N_Vector with corresponding arithmetic operations:

```
module fnvector_fortran_mod
  use, intrinsic :: iso_c_binding
  use fsundials_core_mod
  implicit none
  type, public :: FVec
    logical              :: own_data
    integer(c_int64_t)   :: length1
    integer(c_int64_t)   :: length2
    real(c_double), pointer :: data(:, :)
  end type Fvec

contains

  function FN_VNew_Fortran(n1, n2, ctx) result(y)
    implicit none
    integer(c_int64_t), value  :: n1, n2
    type(c_ptr), value         :: ctx
    type(N_Vector), pointer    :: y
    type(N_Vector_Ops), pointer :: ops
    type(FVec), pointer        :: content

    ! allocate output N_Vector structure
    y => FN_VNewEmpty(sunctx)
```

```
    ! allocate and fill content structure
    allocate (content)
    allocate (content%data(n1, n2))
    content%own_data = .true.
    content%length1 = n1
    content%length2 = n2

    ! attach the content structure to the output
    y%content = c_loc(content)

    ! access the ops struct, and set function pointers
    call c_f_pointer(y%ops, ops)
    ops%nvdestroy   = c_funloc(FN_VDestroy_Fortran)
    ops%nvgetlength = c_funloc(FN_VGetLength_Fortran)
    ops%nvconst     = c_funloc(FN_VConst_Fortran)
    ops%nvdotprod   = c_funloc(FN_VDotProd_Fortran)
    ops%nvclone     = c_funloc(FN_VClone_Fortran)
    ...

  end function FN_VNew_Fortran
```

Custom Vector Interfaces

(Adapted from fnvector_fortran_mod.f90)

Consider a Fortran application that uses a 2D array for storing the solution. A simple module can wrap this as an N_Vector with corresponding arithmetic operations

```
function FN_VGetFVec(s_x) result(x)
  implicit none
  type(N_Vector)     :: s_x
  type(FVec), pointer :: x
  ! extract Fortran matrix structure to output
  call c_f_pointer(s_x%content, x)
  return
end function FN_VGetFVec

subroutine FN_VDestroy_Fortran(s_y) bind(C)
  implicit none
  type(N_Vector), target :: s_y
  type(FVec), pointer    :: y
  ! access FVec structure
  y => FN_VGetFVec(s_y)
  ! if vector owns the data, then deallocate
  if (y%own_data) deallocate (y%data)
  ! deallocate underlying Fortran object (the content)
  deallocate (y)
  ! set N_Vector structure members to NULL and return
  s_y%content = C_NULL_PTR
  ! deallocate overall N_Vector structure
  call FN_VFreeEmpty(s_y)
  return
end subroutine FN_VDestroy_Fortran
```

```
real(c_double) function FN_VDotProd_Fortran(s_x, s_y) &
  result(a) bind(C)
  implicit none
  type(N_Vector)     :: s_x, s_y
  type(FVec), pointer :: x, y
  ! access data, and do op using Fortran intrinsics
  x => FN_VGetFVec(sun_x)
  y => FN_VGetFVec(sun_y)
  a = sum(x%data*y%data)
  return
end function FN_VDotProd_Fortran

subroutine FN_VLinearSum_Fortran(a, s_x, b, s_y, s_z) &
  bind(C)
  implicit none
  type(N_Vector)      :: s_x, s_y, s_z
  real(c_double), value :: a, b
  type(FVec), pointer  :: x, y, z
  ! extract Fortran vector structures and do whole-array op
  x => FN_VGetFVec(s_x)
  y => FN_VGetFVec(s_y)
  z => FN_VGetFVec(s_z)
  z%data = a*x%data + b*y%data
  return
end subroutine FN_VLinearSum_Fortran
end module fnvector_fortran_mod
```

Preconditioning for Implicit Solvers

(Adapted from ark_heat2D_f2003.f90)

Implicit methods for large-scale applications require iterative linear solvers with effective preconditioning. Users provide this via separate "setup" and "solve" routines:

```
integer(c_int) function PSetup(t, y, f, jok, jcurPtr, gamma, &
                               user_data) result(ierr) bind(C)
  use, intrinsic :: iso_c_binding
  implicit none
  real(c_double), value :: t      ! current time
  type(N_Vector)        :: y      ! current solution vector
  type(N_Vector)        :: f      ! current IVP RHS vector
  integer(c_int), value :: jok    ! input signal to update prec
  integer(c_int)        :: jcurPtr ! output that prec was updated
  real(c_double), value :: gamma  ! current gamma value
  type(c_ptr), value :: user_data  ! user-defined data
  ! perform application-specific preconditioner setup
  ...
  jcurPtr = 1   ! signal that preconditioner was updated
  ierr = 0      ! return with success
  return
end function PSetup
```

- Setup is performed infrequently to account for cost (e.g., factorization)
- Solve is performed every Krylov iteration
- These can reuse existing solvers, e.g., those used in operator-splitting methods

```
integer(c_int) function PSolve(t, y, f, r, z, gamma, delta, lr, &
                               user_data) result(ierr) bind(C)
  use, intrinsic :: iso_c_binding
  implicit none
  real(c_double), value :: t      ! current time
  type(N_Vector)        :: y      ! current solution vector
  type(N_Vector)        :: f      ! current IVP RHS vector
  type(N_Vector)        :: r      ! linear system rhs vector
  type(N_Vector)        :: z      ! linear system solution vector
  real(c_double), value :: gamma  ! current gamma value
  real(c_double), value :: delta  ! solve tolerance
  integer(c_int), value :: lr     ! left vs right prec
  type(c_ptr), value :: user_data ! user-defined data
  ! pointers to data in SUNDIALS vectors
  real(c_double), pointer, dimension(nxl, nyl) :: rdata(:, :)
  real(c_double), pointer, dimension(nxl, nyl) :: zdata(:, :)
  ! get data arrays from SUNDIALS vectors
  rdata(1:nxl, 1:nyl) => FN_VGetArrayPointer(r)
  zdata(1:nxl, 1:nyl) => FN_VGetArrayPointer(z)
  ! apply application-specific preconditioner to update zdata
  ...
  ierr = 0     ! return with success
  return
end function PSolve
```

Custom Algebraic Solvers

Some applications have implicit systems with exploitable structure. Users can leverage this problem structure by providing custom nonlinear and/or linear solvers. Users may additionally provide custom matrix structures to go along with their custom linear solver.

- These use the same object-oriented structure as an N_Vector
- Example templates are available:
  - Custom linear solver: fsunlinsol_fortran_mod.f90
  - Custom matrix structure: fsunmatrix_fortran_mod.f90
  - Custom nonlinear solver: ark_brusselator1D_task_local_nls_f2003.f90

Outline

- Overview of SUNDIALS
- How to use the time integrators from modern Fortran
- Interfacing SUNDIALS with existing codes
- **Closing remarks**

Where to learn more and get the software

- Visit the SUNDIALS website: computing.llnl.gov/sundials
- Tutorials are available from “presentations” list: computing.llnl.gov/projects/sundials/publications
- Visit the SUNDIALS GitHub: github.com/LLNL/sundials
- Download from the SUNDIALS website: computing.llnl.gov/projects/sundials/sundials-software
- Install SUNDIALS using Spack: spack install sundials
- View the online documentation: sundials.readthedocs.io

sundials

computing.llnl.gov/sundials

github.com/LLNL/sundials

sundials.readthedocs.io