User Documentation for ARKode v4.6.0

SUNDIALS v5.6.0

Daniel R. Reynolds[1], David J. Gardner[2], Carol S. Woodward[2], and Cody J. Balos[2]

[1]*Department of Mathematics, Southern Methodist University*
[2]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*

December 11, 2020

LLNL-SM-668082

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

CONTRIBUTORS

The SUNDIALS library has been developed over many years by a number of contributors. The current SUNDIALS team consists of Cody J. Balos, David J. Gardner, Alan C. Hindmarsh, Daniel R. Reynolds, and Carol S. Woodward. We thank Radu Serban for significant and critical past contributions.

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

Contents

This is the documentation for ARKode, an adaptive step time integration package for stiff, nonstiff and mixed stiff/nonstiff systems of ordinary differential equations (ODEs) using Runge-Kutta (i.e. one-step, multi-stage) methods. The ARKode solver is a component of the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. It is designed to have a similar user experience to the CVODE solver, including user modes to allow adaptive integration to specified output times, return after each internal step and root-finding capabilities, and for calculations in serial, using shared-memory parallelism (via OpenMP, Pthreads, CUDA, Raja) or distributed-memory parallelism (via MPI). The default integration and solver options should apply to most users, though control over nearly all internal parameters and time adaptivity algorithms is enabled through optional interface routines.

ARKode is written in C, with C++ and Fortran interfaces.

ARKode is developed by Southern Methodist University, with support by the US Department of Energy through the FASTMath SciDAC Institute, under subcontract B598130 from Lawrence Livermore National Laboratory.

Chapter 1

Introduction

The ARKode infrastructure provides adaptive-step time integration modules for stiff, nonstiff and mixed stiff/nonstiff systems of ordinary differential equations (ODEs). ARKode itself is structured to support a wide range of one-step (but multi-stage) methods, allowing for rapid development of parallel implementations of state-of-the-art time integration methods. At present, ARKode is packaged with two time-stepping modules, *ARKStep* and *ERKStep*.

ARKStep supports ODE systems posed in split, linearly-implicit form,

$$M\dot{y} = f^E(t,y) + f^I(t,y), \qquad y(t_0) = y_0, \tag{1.1}$$

where t is the independent variable, y is the set of dependent variables (in $\mathbb{R}^N$), M is a user-specified, nonsingular operator from $\mathbb{R}^N$ to $\mathbb{R}^N$, and the right-hand side function is partitioned into up to two components:

- $f^E(t,y)$ contains the "nonstiff" time scale components to be integrated explicitly, and
- $f^I(t,y)$ contains the "stiff" time scale components to be integrated implicitly.

Either of these operators may be disabled, allowing for fully explicit, fully implicit, or combination implicit-explicit (ImEx) time integration.

The algorithms used in ARKStep are adaptive- and fixed-step additive Runge Kutta methods. Such methods are defined through combining two complementary Runge-Kutta methods: one explicit (ERK) and the other diagonally implicit (DIRK). Through appropriately partitioning the ODE right-hand side into explicit and implicit components (1.1), such methods have the potential to enable accurate and efficient time integration of stiff, nonstiff, and mixed stiff/nonstiff systems of ordinary differential equations. A key feature allowing for high efficiency of these methods is that only the components in $f^I(t,y)$ must be solved implicitly, allowing for splittings tuned for use with optimal implicit solver algorithms.

This framework allows for significant freedom over the constitutive methods used for each component, and ARKode is packaged with a wide array of built-in methods for use. These built-in Butcher tables include adaptive explicit methods of orders 2-8, adaptive implicit methods of orders 2-5, and adaptive ImEx methods of orders 3-5.

ERKStep focuses specifically on problems posed in explicit form,

$$\dot{y} = f(t,y), \qquad y(t_0) = y_0. \tag{1.2}$$

allowing for increased computational efficiency and memory savings. The algorithms used in ERKStep are adaptive- and fixed-step explicit Runge Kutta methods. As with ARKStep, the ERKStep module is packaged with adaptive explicit methods of orders 2-8.

For problems that include nonzero implicit term $f^I(t, y)$, the resulting implicit system (assumed nonlinear, unless specified otherwise) is solved approximately at each integration step, using a modified Newton method, inexact Newton method, or an accelerated fixed-point solver. For the Newton-based methods and the serial or threaded NVECTOR modules in SUNDIALS, ARKode may use a variety of linear solvers provided with SUNDIALS, including both direct (dense, band, or sparse) and preconditioned Krylov iterative (GMRES *[SS1986]*, BiCGStab *[V1992]*, TFQMR *[F1993]*, FGMRES *[S1993]*, or PCG *[HS1952]*) linear solvers. When used with the MPI-based parallel, PETSc, *hypre*, CUDA, HIP, and Raja NVECTOR modules, or a user-provided vector data structure, only the Krylov solvers are available, although a user may supply their own linear solver for any data structures if desired. For the serial or threaded vector structures, we provide a banded preconditioner module called ARKBANDPRE that may be used with the Krylov solvers, while for the MPI-based parallel vector structure there is a preconditioner module called ARKBBDPRE which provides a band-block-diagonal preconditioner. Additionally, a user may supply more optimal, problem-specific preconditioner routines.

1.1 Changes from previous versions

1.1.1 Changes in 4.6.0

A new NVECTOR implementation based on the AMD ROCm HIP platform has been added. This vector can target NVIDIA or AMD GPUs. See *The NVECTOR_HIP Module* for more details. This module is considered experimental and is subject to change from version to version.

The RAJA NVECTOR implementation has been updated to support the HIP backend in addition to the CUDA backend. Users can choose the backend when configuring SUNDIALS by using the `SUNDIALS_RAJA_BACKENDS` CMake variable. This module remains experimental and is subject to change from version to version.

A new optional operation, `N_VGetDeviceArrayPointer()`, was added to the N_Vector API. This operation is useful for N_Vectors that utilize dual memory spaces, e.g. the native SUNDIALS CUDA N_Vector.

The SUNMATRIX_CUSPARSE and SUNLINEARSOLVER_CUSOLVERSP_BATCHQR implementations no longer require the SUNDIALS CUDA N_Vector. Instead, they require that the vector utilized provides the `N_VGetDeviceArrayPointer()` operation, and that the pointer returned by `N_VGetDeviceArrayPointer()` is a valid CUDA device pointer.

1.1.2 Changes in v4.5.0

Refactored the SUNDIALS build system. CMake 3.12.0 or newer is now required. Users will likely see deprecation warnings, but otherwise the changes should be fully backwards compatible for almost all users. SUNDIALS now exports CMake targets and installs a SUNDIALSConfig.cmake file.

Added support for SuperLU DIST 6.3.0 or newer.

1.1.3 Changes in v4.4.0

Added full support for time-dependent mass matrices in ARKStep, and expanded existing non-identity mass matrix infrastructure to support use of the fixed point nonlinear solver. Fixed bug for ERK method integration with static mass matrices.

An interface between ARKStep and the XBraid multigrid reduction in time (MGRIT) library *[XBraid]* has been added to enable parallel-in-time integration. See the *Multigrid Reduction in Time with XBraid* section for more information and the example codes in `examples/arkode/CXX_xbraid`. This interface required the addition of three new N_Vector operations to exchange vector data between computational nodes, see `N_VBufSize()`, `N_VBufPack()`, and `N_VBufUnpack()`. These N_Vector operations are only used within the XBraid interface and need not be implemented for any other context.

Updated the MRIStep time-stepping module in ARKode to support higher-order MRI-GARK methods *[S2019]*, including methods that involve solve-decoupled, diagonally-implicit treatment of the slow time scale.

Added the functions *ARKStepSetLSNormFactor()*, *ARKStepSetMassLSNormFactor()*, and *MRIStepSetLSNormFactor()* to specify the factor for converting between integrator tolerances (WRMS norm) and linear solver tolerances (L2 norm) i.e., `tol_L2 = nrmfac * tol_WRMS`.

Added new reset functions *ARKStepReset()*, *ERKStepReset()*, and *MRIStepReset()* to reset the stepper time and state vector to user-provided values for continuing the integration from that point while retaining the integration history. These function complement the reinitialization functions *ARKStepReInit()*, *ERKStepReInit()*, and *MRIStepReInit()* which reinitialize the stepper so that the problem integration should resume as if started from scratch.

Added new functions *ARKStepComputeState()*, *ARKStepGetNonlinearSystemData()*, *MRIStepComputeState()*, and *MRIStepGetNonlinearSystemData()* which advanced users might find useful if providing a custom *SUNNonlinSolSysFn()*.

The expected behavior of *SUNNonlinSolGetNumIters()* and *SUNNonlinSolGetNumConvFails()* in the SUNNonlinearSolver API have been updated to specify that they should return the number of nonlinear solver iterations and convergence failures in the most recent solve respectively rather than the cumulative number of iterations and failures across all solves respectively. The API documentation and SUNDIALS provided SUNNonlinearSolver implementations have been updated accordingly. As before, the cumulative number of nonlinear iterations may be retrieved by calling *ARKStepGetNumNonlinSolvIters()*, the cumulative number of failures with *ARKStepGetNumNonlinSolvConvFails()*, or both with *ARKStepGetNonlinSolvStats()*.

A minor bug in checking the Jacobian evaluation frequency has been fixed. As a result codes using using a non-default Jacobian update frequency through a call to `ARKStepSetMaxStepsBetweenJac()` will need to increase the provided value by 1 to achieve the same behavior as before. Additionally, for greater clarity the functions `ARKStepSetMaxStepsBetweenLSet()` and `ARKStepSetMaxStepsBetweenJac()` have been deprecated and replaced with *ARKStepSetLSetupFrequency()* and *ARKStepSetJacEvalFrequency()* respectively.

The `NVECTOR_RAJA` module has been updated to mirror the `NVECTOR_CUDA` module. Notably, the update adds managed memory support to the `NVECTOR_RAJA` module. Users of the module will need to update any calls to the `N_VMake_Raja` function because that signature was changed. This module remains experimental and is subject to change from version to version.

The `NVECTOR_TRILINOS` module has been updated to work with Trilinos 12.18+. This update changes the local ordinal type to always be an `int`.

Added support for CUDA v11.

1.1.4 Changes in v4.3.0

Fixed a bug in ARKode where the prototypes for *ERKStepSetMinReduction()* and *ARKStepSetMinReduction()* were not included in `arkode_erkstep.h` and `arkode_arkstep.h` respectively.

Fixed a bug where inequality constraint checking would need to be disabled and then re-enabled to update the inequality constraint values after resizing a problem. Resizing a problem will now disable constraints and a call to *ARKStepSetConstraints()* or *ERKStepSetConstraints()* is required to re-enable constraint checking for the new problem size.

Fixed a bug in the iterative linear solver modules where an error is not returned if the Atimes function is `NULL` or, if preconditioning is enabled, the PSolve function is `NULL`.

Added the ability to control the CUDA kernel launch parameters for the `NVECTOR_CUDA` and `SUNMATRIX_CUSPARSE` modules. These modules remain experimental and are subject to change from version to version. In addition, the `NVECTOR_CUDA` kernels were rewritten to be more flexible. Most users should see

equivalent performance or some improvement, but a select few may observe minor performance degradation with the default settings. Users are encouraged to contact the SUNDIALS team about any perfomance changes that they notice.

Added the optional function *ARKStepSetJacTimesRhsFn()* to specify an alternative implicit right-hand side function for computing Jacobian-vector products with the internal difference quotient approximation.

Added new capabilities for monitoring the solve phase in the SUNNONLINSOL_NEWTON and SUNNONLINSOL_FIXEDPOINT modules, and the SUNDIALS iterative linear solver modules. SUNDIALS must be built with the CMake option SUNDIALS_BUILD_WITH_MONITORING to use these capabilties.

1.1.5 Changes in v4.2.0

Fixed a build system bug related to the Fortran 2003 interfaces when using the IBM XL compiler. When building the Fortran 2003 interfaces with an XL compiler it is recommended to set CMAKE_Fortran_COMPILER to f2003, xlf2003, or xlf2003_r.

Fixed a bug in how ARKode interfaces with a user-supplied, iterative, unscaled linear solver. In this case, ARKode adjusts the linear solver tolerance in an attempt to account for the lack of support for left/right scaling matrices. Previously, ARKode computed this scaling factor using the error weight vector, ewt; this fix changes that to the residual weight vector, rwt, that can differ from ewt when solving problems with non-identity mass matrix.

Fixed a similar bug in how ARKode interfaces with scaled linear solvers when solving problems with non-identity mass matrices. Here, the left scaling matrix should correspond with rwt and the right scaling matrix with ewt; these were reversed but are now correct.

Fixed a bug where a non-default value for the maximum allowed growth factor after the first step would be ignored.

The function *ARKStepSetLinearSolutionScaling()* was added to enable or disable the scaling applied to linear system solutions with matrix-based linear solvers to account for a lagged value of γ in the linear system matrix e.g., $M - \gamma J$ or $I - \gamma J$. Scaling is enabled by default when using a matrix-based linear solver.

Added two new functions, *ARKStepSetMinReduction()* and *ERKStepSetMinReduction()*, to change the minimum allowed step size reduction factor after an error test failure.

Added a new SUNMatrix implementation, *The SUNMATRIX_CUSPARSE Module*, that interfaces to the sparse matrix implementation from the NVIDIA cuSPARSE library. In addition, the *The SUNLinSol_cuSolverSp_batchQR Module* SUNLinearSolver has been updated to use this matrix, as such, users of this module will need to update their code. These modules are still considered to be experimental, thus they are subject to breaking changes even in minor releases.

Added a new "stiff" interpolation module, based on Lagrange polynomial interpolation, that is accessible to each of the ARKStep, ERKStep and MRIStep time-stepping modules. This module is designed to provide increased interpolation accuracy when integrating stiff problems, as opposed to the ARKode-standard Hermite interpolation module that can suffer when the IVP right-hand side has large Lipschitz constant. While the Hermite module remains the default, the new Lagrange module may be enabled using one of the routines *ARKStepSetInterpolantType()*, *ERKStepSetInterpolantType()*, or *MRIStepSetInterpolantType()*. The serial example problem ark_brusselator.c has been converted to use this Lagrange interpolation module. Created accompanying routines *ARKStepSetInterpolantDegree()*, *ARKStepSetInterpolantDegree()* and *ARKStepSetInterpolantDegree()* to provide user control over these interpolating polynomials. While the routines *ARKStepSetDenseOrder()*, *ARKStepSetDenseOrder()* and *ARKStepSetDenseOrder()* still exist, these have been deprecated and will be removed in a future release.

1.1.6 Changes in v4.1.0

Fixed a build system bug related to finding LAPACK/BLAS.

Fixed a build system bug related to checking if the KLU library works.

Fixed a build system bug related to finding PETSc when using the CMake variables `PETSC_INCLUDES` and `PETSC_LIBRARIES` instead of `PETSC_DIR`.

Added a new build system option, `CUDA_ARCH`, that can be used to specify the CUDA architecture to compile for.

Fixed a bug in the Fortran 2003 interfaces to the ARKode Butcher table routines and structure. This includes changing the `ARKodeButcherTable` type to be a `type(c_ptr)` in Fortran.

Added two utility functions, `SUNDIALSFileOpen` and `SUNDIALSFileClose` for creating/destroying file pointers that are useful when using the Fortran 2003 interfaces.

Added support for a user-supplied function to update the prediction for each implicit stage solution in ARKStep. If supplied, this routine will be called *after* any existing ARKStep predictor algorithm completes, so that the predictor may be modified by the user as desired. The new user-supplied routine has type `ARKStepStagePredictFn`, and may be set by calling *`ARKStepSetStagePredictFn()`*.

The MRIStep module has been updated to support attaching different user data pointers to the inner and outer integrators. If applicable, user codes will need to add a call to *`ARKStepSetUserData()`* to attach their user data pointer to the inner integrator memory as *`MRIStepSetUserData()`* will not set the pointer for both the inner and outer integrators. The MRIStep examples have been updated to reflect this change.

Added support for constant damping to the `SUNNonlinearSolver_FixedPoint` module when using Anderson acceleration. See *SUNNonlinearSolver_FixedPoint description* and the *`SUNNonlinSolSetDamping_FixedPoint()`* for more details.

1.1.7 Changes in v4.0.0

Build system changes

Increased the minimum required CMake version to 3.5 for most SUNDIALS configurations, and 3.10 when CUDA or OpenMP with device offloading are enabled.

The CMake option `BLAS_ENABLE` and the variable `BLAS_LIBRARIES` have been removed to simplify builds as SUNDIALS packages do not use BLAS directly. For third party libraries that require linking to BLAS, the path to the BLAS library should be included in the `_LIBRARIES` variable for the third party library e.g., `SUPERLUDIST_LIBRARIES` when enabling SuperLU_DIST.

Fixed a bug in the build system that prevented the PThreads NVECTOR module from being built.

NVECTOR module changes

Two new functions were added to aid in creating custom NVECTOR objects. The constructor *`N_VNewEmpty()`* allocates an "empty" generic NVECTOR with the object's content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the NVECTOR API by ensuring only required operations need to be set. Additionally, the function *`N_VCopyOps()`* has been added to copy the operation function pointers between vector objects. When used in clone routines for custom vector objects these functions also will ease the introduction of any new optional operations to the NVECTOR API by ensuring all operations are copied when cloning objects.

Two new NVECTOR implementations, NVECTOR_MANYVECTOR and NVECTOR_MPIMANYVECTOR, have been created to support flexible partitioning of solution data among different processing elements (e.g., CPU + GPU) or for multi-physics problems that couple distinct MPI-based simulations together. This implementation is accompanied by additions to user documentation and SUNDIALS examples.

One new required vector operation and ten new optional vector operations have been added to the NVECTOR API. The new required operation, *`N_VGetLength()`*, returns the global length of an `N_Vector`. The optional operations have been added to support the new NVECTOR_MPIMANYVECTOR implementation.

The operation *N_VGetCommunicator()* must be implemented by subvectors that are combined to create an NVECTOR_MPIMANYVECTOR, but is not used outside of this context. The remaining nine operations are optional local reduction operations intended to eliminate unnecessary latency when performing vector reduction operations (norms, etc.) on distributed memory systems. The optional local reduction vector operations are *N_VDotProdLocal()*, *N_VMaxNormLocal()*, *N_VMinLocal()*, *N_VL1NormLocal()*, *N_VWSqrSumLocal()*, *N_VWSqrSumMaskLocal()*, *N_VInvTestLocal()*, *N_VConstrMaskLocal()*, and *N_VMinQuotientLocal()*. If an NVECTOR implementation defines any of the local operations as `NULL`, then the NVECTOR_MPIMANYVECTOR will call standard NVECTOR operations to complete the computation.

An additional NVECTOR implementation, NVECTOR_MPIPLUSX, has been created to support the MPI+X paradigm where X is a type of on-node parallelism (*e.g.*, OpenMP, CUDA). The implementation is accompanied by additions to user documentation and SUNDIALS examples.

The `*_MPICuda` and `*_MPIRaja` functions have been removed from the NVECTOR_CUDA and NVECTOR_RAJA implementations respectively. Accordingly, the `nvector_mpicuda.h`, `nvector_mpiraja.h`, `libsundials_nvecmpicuda.lib`, and `libsundials_nvecmpicudaraja.lib` files have been removed. Users should use the NVECTOR_MPIPLUSX module coupled in conjunction with the NVECTOR_CUDA or NVECTOR_RAJA modules to replace the functionality. The necessary changes are minimal and should require few code modifications. See the programs in `examples/ida/mpicuda` and `examples/ida/mpiraja` for examples of how to use the NVECTOR_MPIPLUSX module with the NVECTOR_CUDA and NVECTOR_RAJA modules respectively.

Fixed a memory leak in the NVECTOR_PETSC module clone function.

Made performance improvements to the NVECTOR_CUDA module. Users who utilize a non-default stream should no longer see default stream synchronizations after memory transfers.

Added a new constructor to the NVECTOR_CUDA module that allows a user to provide custom allocate and free functions for the vector data array and internal reduction buffer.

Added new Fortran 2003 interfaces for most NVECTOR modules. See the *Using ARKode for Fortran Applications* section for more details.

Added three new NVECTOR utility functions, `N_VGetVecAtIndexVectorArray()` `N_VSetVecAtIndexVectorArray()`, and `N_VNewVectorArray()`, for working with `N_Vector` arrays when using the Fortran 2003 interfaces.

SUNMatrix module changes

Two new functions were added to aid in creating custom SUNMATRIX objects. The constructor *SUNMatNewEmpty()* allocates an "empty" generic SUNMATRIX with the object's content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the SUNMATRIX API by ensuring only required operations need to be set. Additionally, the function *SUNMatCopyOps()* has been added to copy the operation function pointers between matrix objects. When used in clone routines for custom matrix objects these functions also will ease the introduction of any new optional operations to the SUNMATRIX API by ensuring all operations are copied when cloning objects.

A new operation, *SUNMatMatvecSetup()*, was added to the SUNMATRIX API. Users who have implemented custom SUNMATRIX modules will need to at least update their code to set the corresponding `ops` structure member, `matvecsetup`, to `NULL`.

A new operation, *SUNMatMatvecSetup()*, was added to the SUNMATRIX API to perform any setup necessary for computing a matrix-vector product. This operation is useful for SUNMATRIX implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product. Users who have implemented custom SUNMATRIX modules will need to at least update their code to set the corresponding `ops` structure member, `matvecsetup`, to `NULL`.

The generic SUNMATRIX API now defines error codes to be returned by SUNMATRIX operations. Operations which return an integer flag indiciating success/failure may return different values than previously.

A new SUNMATRIX (and SUNLINEARSOLVER) implementation was added to facilitate the use of the SuperLU_DIST library with SUNDIALS.

Added new Fortran 2003 interfaces for most SUNMATRIX modules. See the *Using ARKode for Fortran Applications* section for more details.

SUNLinearSolver module changes

A new function was added to aid in creating custom SUNLINEARSOLVER objects. The constructor `SUNLinSolNewEmpty()` allocates an "empty" generic SUNLINEARSOLVER with the object's content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the SUNLINEARSOLVER API by ensuring only required operations need to be set.

The return type of the SUNLINEARSOLVER API function `SUNLinSolLastFlag()` has changed from `long int` to `sunindextype` to be consistent with the type used to store row indices in dense and banded linear solver modules.

Added a new optional operation to the SUNLINEARSOLVER API, `SUNLinSolGetID()`, that returns a `SUNLinearSolver_ID` for identifying the linear solver module.

The SUNLINEARSOLVER API has been updated to make the initialize and setup functions optional.

A new SUNLINEARSOLVER (and SUNMATRIX) implementation was added to facilitate the use of the SuperLU_DIST library with SUNDIALS.

Added a new SUNLinearSolver implementation, `SUNLinearSolver_cuSolverSp_batchQR`, which leverages the NVIDIA cuSOLVER sparse batched QR method for efficiently solving block diagonal linear systems on NVIDIA GPUs.

Added three new accessor functions to the SUNLinSol_KLU module, `SUNLinSol_KLUGetSymbolic()`, `SUNLinSol_KLUGetNumeric()`, and `SUNLinSol_KLUGetCommon()`, to provide user access to the underlying KLU solver structures.

Added new Fortran 2003 interfaces for most SUNLINEARSOLVER modules. See the *Using ARKode for Fortran Applications* section for more details.

SUNNonlinearSolver module changes

A new function was added to aid in creating custom SUNNONLINEARSOLVER objects. The constructor `SUNNonlinSolNewEmpty()` allocates an "empty" generic SUNNONLINEARSOLVER with the object's content pointer and the function pointers in the operations structure initialized to `NULL`. When used in the constructor for custom objects this function will ease the introduction of any new optional operations to the SUNNONLINEARSOLVER API by ensuring only required operations need to be set.

To facilitate the use of user supplied nonlinear solver convergence test functions the `SUNNonlinSolSetConvTestFn()` function in the SUNNONLINEARSOLVER API has been updated to take a `void*` data pointer as input. The supplied data pointer will be passed to the nonlinear solver convergence test function on each call.

The inputs values passed to the first two inputs of the `SUNNonlinSolSolve()` function in the SUNNONLINEARSOLVER have been changed to be the predicted state and the initial guess for the correction to that state. Additionally, the definitions of `SUNNonlinSolLSetupFn` and `SUNNonlinSolLSolveFn` in the SUNNONLINEARSOLVER API have been updated to remove unused input parameters.

Added a new `SUNNonlinearSolver` implementation, `SUNNonlinsol_PetscSNES`, which interfaces to the PETSc SNES nonlinear solver API.

Added new Fortran 2003 interfaces for most SUNNONLINEARSOLVER modules. See the *Using ARKode for Fortran Applications* section for more details.

ARKode changes

The MRIStep module has been updated to support explicit, implicit, or IMEX methods as the fast integrator using the ARKStep module. As a result some function signatures have been changed including *MRIStepCreate()* which now takes an ARKStep memory structure for the fast integration as an input.

Fixed a bug in the ARKStep time-stepping module that would result in an infinite loop if the nonlinear solver failed to converge more than the maximum allowed times during a single step.

Fixed a bug that would result in a "too much accuracy requested" error when using fixed time step sizes with explicit methods in some cases.

Fixed a bug in ARKStep where the mass matrix linear solver setup function was not called in the Matrix-free case.

Fixed a minor bug in ARKStep where an incorrect flag is reported when an error occurs in the mass matrix setup or Jacobian-vector product setup functions.

Fixed a memeory leak in FARKODE when not using the default nonlinear solver.

The reinitialization functions *ERKStepReInit()*, *ARKStepReInit()*, and *MRIStepReInit()* have been updated to retain the minimum and maxiumum step size values from before reinitialization rather than resetting them to the default values.

Removed extraneous calls to *N_VMin()* for simulations where the scalar valued absolute tolerance, or all entries of the vector-valued absolute tolerance array, are strictly positive. In this scenario, ARKode will remove at least one global reduction per time step.

The ARKLS interface has been updated to only zero the Jacobian matrix before calling a user-supplied Jacobian evaluation function when the attached linear solver has type `SUNLINEARSOLVER_DIRECT`.

A new linear solver interface function *ARKLsLinSysFn()* was added as an alternative method for evaluating the linear system $A = M - \gamma J$.

Added two new embedded ARK methods of orders 4 and 5 to ARKode (from *[KC2019]*).

Support for optional inequality constraints on individual components of the solution vector has been added the ARKode ERKStep and ARKStep modules. See the descriptions of *ERKStepSetConstraints()* and *ARKStepSetConstraints()* for more details. Note that enabling constraint handling requires the NVECTOR operations *N_VMinQuotient()*, *N_VConstrMask()*, and *N_VCompare()* that were not previously required by ARKode.

Added two new 'Get' functions to ARKStep, *ARKStepGetCurrentGamma()*, and *ARKStepGetCurrentState()*, that may be useful to users who choose to provide their own nonlinear solver implementation.

Add two new 'Set' functions to MRIStep, *MRIStepSetPreInnerFn()* and *MRIStepSetPostInnerFn()* for performing communication or memory transfers needed before or after the inner integration.

A new Fortran 2003 interface to ARKode was added. This includes Fortran 2003 interfaces to the ARKStep, ERKStep, and MRIStep time-stepping modules. See the *Using ARKode for Fortran Applications* section for more details.

1.1.8 Changes in v3.1.0

An additional NVECTOR implementation was added for the Tpetra vector from the Trilinos library to facilitate interoperability between SUNDIALS and Trilinos. This implementation is accompanied by additions to user documentation and SUNDIALS examples.

A bug was fixed where a nonlinear solver object could be freed twice in some use cases.

The `EXAMPLES_ENABLE_RAJA` CMake option has been removed. The option `EXAMPLES_ENABLE_CUDA` enables all examples that use CUDA including the RAJA examples with a CUDA back end (if the RAJA NVECTOR is enabled).

The implementation header file *arkode_impl.h* is no longer installed. This means users who are directly manipulating the `ARKodeMem` structure will need to update their code to use ARKode's public API.

Python is no longer required to run `make test` and `make test_install`.

Fixed a bug in `ARKodeButcherTable_Write` when printing a Butcher table without an embedding.

1.1.9 Changes in v3.0.2

Added information on how to contribute to SUNDIALS and a contributing agreement.

1.1.10 Changes in v3.0.1

A bug in ARKode where single precision builds would fail to compile has been fixed.

1.1.11 Changes in v3.0.0

The ARKode library has been entirely rewritten to support a modular approach to one-step methods, which should allow rapid research and development of novel integration methods without affecting existing solver functionality. To support this, the existing ARK-based methods have been encapsulated inside the new `ARKStep` time-stepping module. Two new time-stepping modules have been added:

- The `ERKStep` module provides an optimized implementation for explicit Runge-Kutta methods with reduced storage and number of calls to the ODE right-hand side function.
- The `MRIStep` module implements two-rate explicit-explicit multirate infinitesimal step methods utilizing different step sizes for slow and fast processes in an additive splitting.

This restructure has resulted in numerous small changes to the user interface, particularly the suite of "Set" routines for user-provided solver parameters and "Get" routines to access solver statistics, that are now prefixed with the name of time-stepping module (e.g., `ARKStep` or `ERKStep`) instead of `ARKode`. Aside from affecting the names of these routines, user-level changes have been kept to a minimum. However, we recommend that users consult both this documentation and the ARKode example programs for further details on the updated infrastructure.

As part of the ARKode restructuring an *`ARKodeButcherTable`* structure has been added for storing Butcher tables. Functions for creating new Butcher tables and checking their analytic order are provided along with other utility routines. For more details see *Butcher Table Data Structure*.

Two changes were made in the initial step size algorithm:

- Fixed an efficiency bug where an extra call to the right hand side function was made.
- Changed the behavior of the algorithm if the max-iterations case is hit. Before the algorithm would exit with the step size calculated on the penultimate iteration. Now it will exit with the step size calculated on the final iteration.

ARKode's dense output infrastructure has been improved to support higher-degree Hermite polynomial interpolants (up to degree 5) over the last successful time step.

ARKode's previous direct and iterative linear solver interfaces, ARKDLS and ARKSPILS, have been merged into a single unified linear solver interface, ARKLS, to support any valid SUNLINSOL module. This includes `DIRECT` and `ITERATIVE` types as well as the new `MATRIX_ITERATIVE` type. Details regarding how ARKLS utilizes linear solvers of each type as well as discussion regarding intended use cases for user-supplied SUNLinSol implementations are included in the chapter *Description of the SUNLinearSolver module*. All ARKode examples programs and the standalone linear solver examples have been updated to use the unified linear solver interface.

The user interface for the new ARKLS module is very similar to the previous ARKDLS and ARKSPILS interfaces. Additionally, we note that Fortran users will need to enlarge their `iout` array of optional integer outputs, and update the indices that they query for certain linear-solver-related statistics.

The names of all constructor routines for SUNDIALS-provided SUNLinSol implementations have been updated to follow the naming convention `SUNLinSol_*` where `*` is the name of the linear solver. The new names are `SUNLinSol_Band`, `SUNLinSol_Dense`, `SUNLinSol_KLU`, `SUNLinSol_LapackBand`, `SUNLinSol_LapackDense`, `SUNLinSol_PCG`, `SUNLinSol_SPBCGS`, `SUNLinSol_SPFGMR`, `SUNLinSol_SPGMR`, `SUNLinSol_SPTFQMR`, and `SUNLinSol_SuperLUMT`. Solver-specific "set" routine names have been similarly standardized. To minimize challenges in user migration to the new names, the previous routine names may still be used; these will be deprecated in future releases, so we recommend that users migrate to the new names soon. All ARKode example programs and the standalone linear solver examples have been updated to use the new naming convention.

The `SUNBandMatrix` constructor has been simplified to remove the storage upper bandwidth argument.

SUNDIALS integrators have been updated to utilize generic nonlinear solver modules defined through the SUNNONLINSOL API. This API will ease the addition of new nonlinear solver options and allow for external or user-supplied nonlinear solvers. The SUNNONLINSOL API and SUNDIALS provided modules are described in *Description of the SUNNonlinearSolver Module* and follow the same object oriented design and implementation used by the NVector, SUNMatrix, and SUNLinSol modules. Currently two SUNNONLINSOL implementations are provided, SUNNonlinSol_Newton and SUNNonlinSol_FixedPoint. These replicate the previous integrator specific implementations of a Newton iteration and an accelerated fixed-point iteration, respectively. Example programs using each of these nonlinear solver modules in a standalone manner have been added and all ARKode example programs have been updated to use generic SUNNonlinSol modules.

As with previous versions, ARKode will use the Newton solver (now provided by SUNNonlinSol_Newton) by default. Use of the *`ARKStepSetLinear()`* routine (previously named `ARKodeSetLinear`) will indicate that the problem is linearly-implicit, using only a single Newton iteration per implicit stage. Users wishing to switch to the accelerated fixed-point solver are now required to create a SUNNonlinSol_FixedPoint object and attach that to ARKode, instead of calling the previous `ARKodeSetFixedPoint` routine. See the documentation sections *A skeleton of the user's main program*, *Nonlinear solver interface functions*, and *The SUNNonlinearSolver_FixedPoint implementation* for further details, or the serial C example program `ark_brusselator_fp.c` for an example.

Three fused vector operations and seven vector array operations have been added to the NVECTOR API. These *optional* operations are disabled by default and may be activated by calling vector specific routines after creating an NVector (see *Description of the NVECTOR Modules* for more details). The new operations are intended to increase data reuse in vector operations, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. The fused operations are `N_VLinearCombination`, `N_VScaleAddMulti`, and `N_VDotProdMulti`, and the vector array operations are `N_VLinearCombinationVectorArray`, `N_VScaleVectorArray`, `N_VConstVectorArray`, `N_VWrmsNormVectorArray`, `N_VWrmsNormMaskVectorArray`, `N_VScaleAddMultiVectorArray`, and `N_VLinearCombinationVectorArray`. If an NVector implementation defines any of these operations as `NULL`, then standard NVector operations will automatically be called as necessary to complete the computation.

Multiple changes to the CUDA NVECTOR were made:

- Changed the `N_VMake_Cuda` function to take a host data pointer and a device data pointer instead of an `N_VectorContent_Cuda` object.
- Changed `N_VGetLength_Cuda` to return the global vector length instead of the local vector length.
- Added `N_VGetLocalLength_Cuda` to return the local vector length.
- Added `N_VGetMPIComm_Cuda` to return the MPI communicator used.
- Removed the accessor functions in the namespace `suncudavec`.
- Added the ability to set the `cudaStream_t` used for execution of the CUDA NVECTOR kernels. See the function `N_VSetCudaStreams_Cuda`.

- Added N_VNewManaged_Cuda, N_VMakeManaged_Cuda, and N_VIsManagedMemory_Cuda functions to accommodate using managed memory with the CUDA NVECTOR.

Multiple changes to the RAJA NVECTOR were made:

- Changed N_VGetLength_Raja to return the global vector length instead of the local vector length.
- Added N_VGetLocalLength_Raja to return the local vector length.
- Added N_VGetMPIComm_Raja to return the MPI communicator used.
- Removed the accessor functions in the namespace sunrajavec.

A new NVECTOR implementation for leveraging OpenMP 4.5+ device offloading has been added, NVECTOR_OpenMPDEV. See *The NVECTOR_OPENMPDEV Module* for more details.

1.1.12 Changes in v2.2.1

Fixed a bug in the CUDA NVECTOR where the N_VInvTest operation could write beyond the allocated vector data.

Fixed library installation path for multiarch systems. This fix changes the default library installation path to CMAKE_INSTALL_PREFIX/CMAKE_INSTALL_LIBDIR from CMAKE_INSTALL_PREFIX/lib. CMAKE_INSTALL_LIBDIR is automatically set, but is available as a CMAKE option that can modified.

1.1.13 Changes in v2.2.0

Fixed a problem with setting sunindextype which would occur with some compilers (e.g. armclang) that did not define __STDC_VERSION__.

Added hybrid MPI/CUDA and MPI/RAJA vectors to allow use of more than one MPI rank when using a GPU system. The vectors assume one GPU device per MPI rank.

Changed the name of the RAJA NVECTOR library to libsundials_nveccudaraja.lib from libsundials_nvecraja.lib to better reflect that we only support CUDA as a backend for RAJA currently.

Several changes were made to the build system:

- CMake 3.1.3 is now the minimum required CMake version.
- Deprecate the behavior of the SUNDIALS_INDEX_TYPE CMake option and added the SUNDIALS_INDEX_SIZE CMake option to select the sunindextype integer size.
- The native CMake FindMPI module is now used to locate an MPI installation.
- If MPI is enabled and MPI compiler wrappers are not set, the build system will check if CMAKE_<language>_COMPILER can compile MPI programs before trying to locate and use an MPI installation.
- The previous options for setting MPI compiler wrappers and the executable for running MPI programs have been have been depreated. The new options that align with those used in native CMake FindMPI module are MPI_C_COMPILER, MPI_CXX_COMPILER, MPI_Fortran_COMPILER, and MPIEXEC_EXECUTABLE.
- When a Fortran name-mangling scheme is needed (e.g., ENABLE_LAPACK is ON) the build system will infer the scheme from the Fortran compiler. If a Fortran compiler is not available or the inferred or default scheme needs to be overridden, the advanced options SUNDIALS_F77_FUNC_CASE and SUNDIALS_F77_FUNC_UNDERSCORES can be used to manually set the name-mangling scheme and bypass trying to infer the scheme.
- Parts of the main CMakeLists.txt file were moved to new files in the src and example directories to make the CMake configuration file structure more modular.

1.1.14 Changes in v2.1.2

Updated the minimum required version of CMake to 2.8.12 and enabled using rpath by default to locate shared libraries on OSX.

Fixed Windows specific problem where sunindextype was not correctly defined when using 64-bit integers for the SUNDIALS index type. On Windows sunindextype is now defined as the MSVC basic type `__int64`.

Added sparse SUNMatrix "Reallocate" routine to allow specification of the nonzero storage.

Updated the KLU SUNLinearSolver module to set constants for the two reinitialization types, and fixed a bug in the full reinitialization approach where the sparse SUNMatrix pointer would go out of scope on some architectures.

Updated the "ScaleAdd" and "ScaleAddI" implementations in the sparse SUNMatrix module to more optimally handle the case where the target matrix contained sufficient storage for the sum, but had the wrong sparsity pattern. The sum now occurs in-place, by performing the sum backwards in the existing storage. However, it is still more efficient if the user-supplied Jacobian routine allocates storage for the sum $I + \gamma J$ or $M + \gamma J$ manually (with zero entries if needed).

Changed LICENSE install path to `instdir/include/sundials`.

1.1.15 Changes in v2.1.1

Fixed a potential memory leak in the SPGMR and SPFGMR linear solvers: if "Initialize" was called multiple times then the solver memory was reallocated (without being freed).

Fixed a minor bug in the ARKReInit routine, where a flag was incorrectly set to indicate that the problem had been resized (instead of just re-initialized).

Fixed C++11 compiler errors/warnings about incompatible use of string literals.

Updated KLU SUNLinearSolver module to use a `typedef` for the precision-specific solve function to be used (to avoid compiler warnings).

Added missing typecasts for some `(void*)` pointers (again, to avoid compiler warnings).

Bugfix in `sunmatrix_sparse.c` where we had used `int` instead of `sunindextype` in one location.

Added missing `#include <stdio.h>` in NVECTOR and SUNMATRIX header files.

Added missing prototype for `ARKSpilsGetNumMTSetups`.

Fixed an indexing bug in the CUDA NVECTOR implementation of `N_VWrmsNormMask` and revised the RAJA NVECTOR implementation of `N_VWrmsNormMask` to work with mask arrays using values other than zero or one. Replaced `double` with `realtype` in the RAJA vector test functions.

Fixed compilation issue with GCC 7.3.0 and Fortran programs that do not require a SUNMatrix or SUNLinearSolver module (e.g. iterative linear solvers, explicit methods, fixed point solver, etc.).

1.1.16 Changes in v2.1.0

Added NVECTOR print functions that write vector data to a specified file (e.g. `N_VPrintFile_Serial`).

Added `make test` and `make test_install` options to the build system for testing SUNDIALS after building with `make` and installing with `make install` respectively.

1.1.17 Changes in v2.0.0

All interfaces to matrix structures and linear solvers have been reworked, and all example programs have been updated. The goal of the redesign of these interfaces was to provide more encapsulation and ease in interfacing custom linear solvers and interoperability with linear solver libraries.

Specific changes include:

- Added generic SUNMATRIX module with three provided implementations: dense, banded and sparse. These replicate previous SUNDIALS Dls and Sls matrix structures in a single object-oriented API.
- Added example problems demonstrating use of generic SUNMATRIX modules.
- Added generic SUNLINEARSOLVER module with eleven provided implementations: dense, banded, LAPACK dense, LAPACK band, KLU, SuperLU_MT, SPGMR, SPBCGS, SPTFQMR, SPFGMR, PCG. These replicate previous SUNDIALS generic linear solvers in a single object-oriented API.
- Added example problems demonstrating use of generic SUNLINEARSOLVER modules.
- Expanded package-provided direct linear solver (Dls) interfaces and scaled, preconditioned, iterative linear solver (Spils) interfaces to utilize generic SUNMATRIX and SUNLINEARSOLVER objects.
- Removed package-specific, linear solver-specific, solver modules (e.g. CVDENSE, KINBAND, IDAKLU, ARKSPGMR) since their functionality is entirely replicated by the generic Dls/Spils interfaces and SUNLINEARSOLVER/SUNMATRIX modules. The exception is CVDIAG, a diagonal approximate Jacobian solver available to CVODE and CVODES.
- Converted all SUNDIALS example problems to utilize new generic SUNMATRIX and SUNLINEARSOLVER objects, along with updated Dls and Spils linear solver interfaces.
- Added Spils interface routines to ARKode, CVODE, CVODES, IDA and IDAS to allow specification of a user-provided "JTSetup" routine. This change supports users who wish to set up data structures for the user-provided Jacobian-times-vector ("JTimes") routine, and where the cost of one JTSetup setup per Newton iteration can be amortized between multiple JTimes calls.

Two additional NVECTOR implementations were added – one for CUDA and one for RAJA vectors. These vectors are supplied to provide very basic support for running on GPU architectures. Users are advised that these vectors both move all data to the GPU device upon construction, and speedup will only be realized if the user also conducts the right-hand-side function evaluation on the device. In addition, these vectors assume the problem fits on one GPU. Further information about RAJA, users are referred to the web site, https://software.llnl.gov/RAJA/. These additions are accompanied by additions to various interface functions and to user documentation.

All indices for data structures were updated to a new `sunindextype` that can be configured to be a 32- or 64-bit integer data index type. `sunindextype` is defined to be `int32_t` or `int64_t` when portable types are supported, otherwise it is defined as `int` or `long int`. The Fortran interfaces continue to use `long int` for indices, except for their sparse matrix interface that now uses the new `sunindextype`. This new flexible capability for index types includes interfaces to PETSc, *hypre*, SuperLU_MT, and KLU with either 32-bit or 64-bit capabilities depending how the user configures SUNDIALS.

To avoid potential namespace conflicts, the macros defining `booleantype` values `TRUE` and `FALSE` have been changed to `SUNTRUE` and `SUNFALSE` respectively.

Temporary vectors were removed from preconditioner setup and solve routines for all packages. It is assumed that all necessary data for user-provided preconditioner operations will be allocated and stored in user-provided data structures.

The file `include/sundials_fconfig.h` was added. This file contains SUNDIALS type information for use in Fortran programs.

Added functions SUNDIALSGetVersion and SUNDIALSGetVersionNumber to get SUNDIALS release version information at runtime.

The build system was expanded to support many of the xSDK-compliant keys. The xSDK is a movement in scientific software to provide a foundation for the rapid and efficient production of high-quality, sustainable extreme-scale scientific applications. More information can be found at, https://xsdk.info.

In addition, numerous changes were made to the build system. These include the addition of separate `BLAS_ENABLE` and `BLAS_LIBRARIES` CMake variables, additional error checking during CMake configuration, minor bug fixes, and renaming CMake options to enable/disable examples for greater clarity and an added option to enable/disable Fortran 77 examples. These changes included changing `ENABLE_EXAMPLES` to `ENABLE_EXAMPLES_C`, changing `CXX_ENABLE` to `EXAMPLES_ENABLE_CXX`, changing `F90_ENABLE` to `EXAMPLES_ENABLE_F90`, and adding an `EXAMPLES_ENABLE_F77` option.

Corrections and additions were made to the examples, to installation-related files, and to the user documentation.

1.1.18 Changes in v1.1.0

We have included numerous bugfixes and enhancements since the v1.0.2 release.

The bugfixes include:

- For each linear solver, the various solver performance counters are now initialized to 0 in both the solver specification function and in the solver's `linit` function. This ensures that these solver counters are initialized upon linear solver instantiation as well as at the beginning of the problem solution.
- The choice of the method vs embedding the Billington and TRBDF2 explicit Runge-Kutta methods were swapped, since in those the lower-order coefficients result in an A-stable method, while the higher-order coefficients do not. This change results in significantly improved robustness when using those methods.
- A bug was fixed for the situation where a user supplies a vector of absolute tolerances, and also uses the vector Resize() functionality.
- A bug was fixed wherein a user-supplied Butcher table without an embedding is supplied, and the user is running with either fixed time steps (or they do adaptivity manually); previously this had resulted in an error since the embedding order was below 1.
- Numerous aspects of the documentation were fixed and/or clarified.

The feature changes/enhancements include:

- Two additional NVECTOR implementations were added – one for Hypre (parallel) ParVector vectors, and one for PETSc vectors. These additions are accompanied by additions to various interface functions and to user documentation.
- Each NVECTOR module now includes a function, `N_VGetVectorID`, that returns the NVECTOR module name.
- A memory leak was fixed in the banded preconditioner and banded-block-diagonal preconditioner interfaces. In addition, updates were done to return integers from linear solver and preconditioner 'free' routines.
- The Krylov linear solver Bi-CGstab was enhanced by removing a redundant dot product. Various additions and corrections were made to the interfaces to the sparse solvers KLU and SuperLU_MT, including support for CSR format when using KLU.
- The ARKode implicit predictor algorithms were updated: methods 2 and 3 were improved slightly, a new predictor approach was added, and the default choice was modified.
- The underlying sparse matrix structure was enhanced to allow both CSR and CSC matrices, with CSR supported by the KLU linear solver interface. ARKode interfaces to the KLU solver from both C and Fortran were updated to enable selection of sparse matrix type, and a Fortran-90 CSR example program was added.
- The missing `ARKSpilsGetNumMtimesEvals()` function was added – this had been included in the previous documentation but had not been implemented.

- The handling of integer codes for specifying built-in ARKode Butcher tables was enhanced. While a global numbering system is still used, methods now have #defined names to simplify the user interface and to streamline incorporation of new Butcher tables into ARKode.
- The maximum number of Butcher table stages was increased from 8 to 15 to accommodate very high order methods, and an 8th-order adaptive ERK method was added.
- Support was added for the explicit and implicit methods in an additive Runge-Kutta method to utilize different stage times, solution and embedding coefficients, to support new SSP-ARK methods.
- The FARKODE interface was extended to include a routine to set scalar/array-valued residual tolerances, to support Fortran applications with non-identity mass-matrices.

1.2 Reading this User Guide

This user guide is a combination of general usage instructions and specific example programs. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

The structure of this document is as follows:

- In the next section we provide a thorough presentation of the underlying *mathematics* used within the ARKode family of solvers.
- We follow this with an overview of how the source code for ARKode is *organized*.
- The largest section follows, providing a full account of the ARKStep module user interface, including a description of all user-accessible functions and outlines for usage in serial and parallel applications. Since ARKode is written in C, we first present a section on *using ARKStep for C and C++ applications*, followed with a separate section on *using ARKode within Fortran applications*.
- The much smaller section describing the ERKStep time-stepping module, *using ERKStep for C and C++ applications*, follows.
- Subsequent sections discuss shared features between ARKode and the rest of the SUNDIALS library: *vector data structures*, *matrix data structures*, *linear solver data structures*, and the *installation procedure*.
- The final sections catalog the full set of *ARKode constants*, that are used for both input specifications and return codes, and the full set of *Butcher tables* that are packaged with ARKode.

1.3 SUNDIALS Release License

All SUNDIALS packages are released open source, under the BSD 3-Clause license. The only requirements of the license are preservation of copyright and a standard disclaimer of liability. The full text of the license and an additional notice are provided below and may also be found in the LICENSE and NOTICE files provided with all SUNDIALS packages.

PLEASE NOTE If you are using SUNDIALS with any third party libraries linked in (e.g., LAPACK, KLU, SuperLU_MT, PETSc, or *hypre*), be sure to review the respective license of the package as that license may have more restrictive terms than the SUNDIALS license. For example, if someone builds SUNDIALS with a statically linked KLU, the build is subject to terms of the more-restrictive LGPL license (which is what KLU is released with) and *not* the SUNDIALS BSD license anymore.

1.3.1 BSD 3-Clause License

1.3.2 Additional Notice

This work was produced under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

1.3.3 SUNDIALS Release Numbers

LLNL-CODE-667205 (ARKODE)

UCRL-CODE-155951 (CVODE)

UCRL-CODE-155950 (CVODES)

UCRL-CODE-155952 (IDA)

UCRL-CODE-237203 (IDAS)

LLNL-CODE-665877 (KINSOL)

Chapter 2

Mathematical Considerations

ARKode solves ODE initial value problems (IVP) in $\mathbb{R}^N$ posed in the form

$$M(t)\,\dot{y} = f(t,y), \qquad y(t_0) = y_0. \tag{2.1}$$

Here, t is the independent variable (e.g. time), and the dependent variables are given by $y \in \mathbb{R}^N$, where we use the notation $\dot{y}$ to denote dy/dt.

For each value of t, $M(t)$ is a user-specified linear operator from $\mathbb{R}^N \to \mathbb{R}^N$. This operator is assumed to be non-singular and independent of y. For standard systems of ordinary differential equations and for problems arising from the spatial semi-discretization of partial differential equations using finite difference, finite volume, or spectral finite element methods, M is typically the identity matrix, I. For PDEs using standard finite-element spatial semi-discretizations, M is typically a well-conditioned mass matrix that is fixed throughout a simulation (or at least fixed between spatial rediscretization events).

The ODE right-hand side is given by the function $f(t,y)$ – in general we make no assumption that the problem (2.1) is autonomous (i.e., $f = f(y)$) or linear ($f = Ay$). In general, the time integration methods within ARKode support additive splittings of this right-hand side function, as described in the subsections that follow. Through these splittings, the time-stepping methods currently supplied with ARKode are designed to solve stiff, nonstiff, mixed stiff/nonstiff, and multirate problems. As per Ascher and Petzold *[AP1998]*, a problem is "stiff" if the stepsize needed to maintain stability of the forward Euler method is much smaller than that required to represent the solution accurately.

In the sub-sections that follow, we elaborate on the numerical methods utilized in ARKode. We first discuss the "single-step" nature of the ARKode infrastructure, including its usage modes and approaches for interpolated solution output. We then discuss the current suite of time-stepping modules supplied with ARKode, including the ARKStep module for *additive Runge-Kutta methods*, the ERKStep module that is optimized for *explicit Runge-Kutta methods*, and the MRIStep module for *multirate infinitesimal step (MIS) based methods*. We then discuss the *adaptive temporal error controllers* shared by the time-stepping modules, including discussion of our choice of norms for measuring errors within various components of the solver.

We then discuss the nonlinear and linear solver strategies used by ARKode's time-stepping modules for solving implicit algebraic systems that arise in computing each stage and/or step: *nonlinear solvers*, *linear solvers*, *preconditioners*, *error control* within iterative nonlinear and linear solvers, algorithms for *initial predictors* for implicit stage solutions, and approaches for handling *non-identity mass-matrices*.

We conclude with a section describing ARKode's *rootfinding capabilities*, that may be used to stop integration of a problem prematurely based on traversal of roots in user-specified functions.

2.1 Adaptive single-step methods

The ARKode infrastructure is designed to support single-step, IVP integration methods, i.e.

$$y_n = \varphi(y_{n-1}, h_n)$$

where y_{n-1} is an approximation to the solution $y(t_{n-1})$, y_n is an approximation to the solution $y(t_n)$, $t_n = t_{n-1} + h_n$, and the approximation method is represented by the function φ.

The choice of step size h_n is determined by the time-stepping method (based on user-provided inputs, typically accuracy requirements). However, users may place minimum/maximum bounds on h_n if desired.

ARKode's time stepping modules may be run in a variety of "modes":

- **NORMAL** – The solver will take internal steps until it has just overtaken a user-specified output time, t_{out}, in the direction of integration, i.e. $t_{n-1} < t_{\text{out}} \le t_n$ for forward integration, or $t_n \le t_{\text{out}} < t_{n-1}$ for backward integration. It will then compute an approximation to the solution $y(t_{\text{out}})$ by interpolation (using one of the dense output routines described in the section *Interpolation*).
- **ONE-STEP** – The solver will only 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 the solver will again return an interpolated result; otherwise it will return a copy of the internal solution y_n.
- **NORMAL-TSTOP** – The solver will take internal steps until the next step will overtake t_{out}. It will then limit this next step so that $t_n = t_{n-1} + h_n = t_{\text{out}}$, and once the step completes it will return a copy of the internal solution y_n.
- **ONE-STEP-TSTOP** – The solver will check whether the next step will overtake t_{out} – if not then this mode is identical to "one-step" above; otherwise it will limit this next step so that $t_n = t_{n-1} + h_n = t_{\text{out}}$. In either case, once the step completes it will return a copy of the internal solution y_n.

We note that interpolated solutions may be slightly less accurate than the internal solutions produced by the solver. Hence, to ensure that the returned value has full method accuracy one of the "tstop" modes may be used.

2.2 Interpolation

As mentioned above, the time-stepping modules in ARKode support interpolation of solutions $y(t_{\text{out}})$ and derivatives $y^{(d)}(t_{\text{out}})$, where t_{out} occurs within a completed time step from $t_{n-1} \to t_n$. Additionally, this module supports extrapolation of solutions and derivatives for t outside this interval (e.g. to construct predictors for iterative nonlinear and linear solvers). To this end, ARKode currently supports construction of polynomial interpolants $p_q(t)$ of polynomial degree up to $q = 5$, although users may select interpolants of lower degree.

ARKode provides two complementary interpolation approaches, both of which are accessible from any of the time-stepping modules: "Hermite" and "Lagrange". The former approach has been included with ARKode since its inception, and is more suitable for non-stiff problems; the latter is a new approach that is designed to provide increased accuracy when integrating stiff problems. Both are described in detail below.

2.2.1 Hermite interpolation module

For non-stiff problems, polynomial interpolants of Hermite form are provided. Rewriting the IVP (2.1) in standard form,

$$\dot{y} = \hat{f}(t, y), \qquad y(t_0) = y_0.$$

we typically construct temporal interpolants using the data $\left\{y_{n-1}, \hat{f}_{n-1}, y_n, \hat{f}_n\right\}$, where here we use the simplified notation $\hat{f}_k$ to denote $\hat{f}(t_k, y_k)$. Defining a normalized "time" variable, τ, for the most-recently-computed solution interval $t_{n-1} \to t_n$ as

$$\tau(t) = \frac{t - t_n}{h_n},$$

we then construct the interpolants $p_q(t)$ as follows:

- $q = 0$: constant interpolant

$$p_0(\tau) = \frac{y_{n-1} + y_n}{2}.$$

- $q = 1$: linear Lagrange interpolant

$$p_1(\tau) = -\tau\, y_{n-1} + (1 + \tau)\, y_n.$$

- $q = 2$: quadratic Hermite interpolant

$$p_2(\tau) = \tau^2\, y_{n-1} + (1 - \tau^2)\, y_n + h_n(\tau + \tau^2)\, \hat{f}_n.$$

- $q = 3$: cubic Hermite interpolant

$$p_3(\tau) = (3\tau^2 + 2\tau^3)\, y_{n-1} + (1 - 3\tau^2 - 2\tau^3)\, y_n + h_n(\tau^2 + \tau^3)\, \hat{f}_{n-1} + h_n(\tau + 2\tau^2 + \tau^3)\, \hat{f}_n.$$

- $q = 4$: quartic Hermite interpolant

$$\begin{aligned} p_4(\tau) = {} & (-6\tau^2 - 16\tau^3 - 9\tau^4)\, y_{n-1} + (1 + 6\tau^2 + 16\tau^3 + 9\tau^4)\, y_n + \frac{h_n}{4}(-5\tau^2 - 14\tau^3 - 9\tau^4)\, \hat{f}_{n-1} \\ & + h_n(\tau + 2\tau^2 + \tau^3)\, \hat{f}_n + \frac{27h_n}{4}(-\tau^4 - 2\tau^3 - \tau^2)\, \hat{f}_a, \end{aligned}$$

  where $\hat{f}_a = \hat{f}\left(t_n - \frac{h_n}{3}, p_3\left(-\frac{1}{3}\right)\right)$. We point out that interpolation at this degree requires an additional evaluation of the full right-hand side function $\hat{f}(t, y)$, thereby increasing its cost in comparison with $p_3(t)$.

- $q = 5$: quintic Hermite interpolant

$$\begin{aligned} p_5(\tau) = {} & (54\tau^5 + 135\tau^4 + 110\tau^3 + 30\tau^2)\, y_{n-1} + (1 - 54\tau^5 - 135\tau^4 - 110\tau^3 - 30\tau^2)\, y_n \\ & + \frac{h_n}{4}(27\tau^5 + 63\tau^4 + 49\tau^3 + 13\tau^2)\, \hat{f}_{n-1} + \frac{h_n}{4}(27\tau^5 + 72\tau^4 + 67\tau^3 + 26\tau^2 + \tau)\, \hat{f}_n \\ & + \frac{h_n}{4}(81\tau^5 + 189\tau^4 + 135\tau^3 + 27\tau^2)\, \hat{f}_a + \frac{h_n}{4}(81\tau^5 + 216\tau^4 + 189\tau^3 + 54\tau^2)\, \hat{f}_b, \end{aligned}$$

  where $\hat{f}_a = \hat{f}\left(t_n - \frac{h_n}{3}, p_4\left(-\frac{1}{3}\right)\right)$ and $\hat{f}_b = \hat{f}\left(t_n - \frac{2h_n}{3}, p_4\left(-\frac{2}{3}\right)\right)$. We point out that interpolation at this degree requires four additional evaluations of the full right-hand side function $\hat{f}(t, y)$, thereby significantly increasing its cost over $p_4(t)$.

We note that although interpolants of order $q > 5$ are possible, these are not currently implemented due to their increased computing and storage costs.

2.2.2 Lagrange interpolation module

For stiff problems where $\hat{f}$ may have large Lipschitz constant, polynomial interpolants of Lagrange form are provided. These interpolants are constructed using the data $\{y_n, y_{n-1}, \ldots, y_{n-\nu}\}$ where $0 \le \nu \le 5$. These polynomials have the form

$$p(t) = \sum_{j=0}^{\nu} y_{n-j} p_j(t), \quad \text{where}$$
$$p_j(t) = \prod_{l=0, l\neq j}^{\nu} \left(\frac{t - t_l}{t_j - t_l}\right), \quad j = 0, \ldots, \nu.$$

Since we assume that the solutions y_{n-j} have length much larger than $\nu \le 5$ in ARKode-based simulations, we evaluate p at any desired $t \in \mathbb{R}$ by first evaluating the Lagrange polynomial basis functions at the input value for t, and then performing a simple linear combination of the vectors $\{y_k\}_{k=0}^{\nu}$. Derivatives $p^{(d)}(t)$ may be evaluated similarly as

$$p^{(d)}(t) = \sum_{j=0}^{\nu} y_{n-j}\, p_j^{(d)}(t),$$

however since the algorithmic complexity involved in evaluating derivatives of the Lagrange basis functions increases dramatically as the derivative order grows, our Lagrange interpolation module currently only provides derivatives up to $d = 3$.

We note that when using this interpolation module, during the first $(\nu - 1)$ steps of integration we do not have sufficient solution history to construct the full ν-degree interpolant. Therefore during these initial steps, we construct the highest-degree interpolants that are currently available at the moment, achieving the full ν-degree interpolant once these initial steps have completed.

2.3 ARKStep – Additive Runge-Kutta methods

The ARKStep time-stepping module in ARKode is designed for IVPs of the form

$$M(t)\,\dot{y} = f^E(t, y) + f^I(t, y), \qquad y(t_0) = y_0, \tag{2.2}$$

i.e. the right-hand side function is additively split into two components:

- $f^E(t, y)$ contains the "nonstiff" components of the system (this will be integrated using an explicit method);
- $f^I(t, y)$ contains the "stiff" components of the system (this will be integrated using an implicit method);

and the left-hand side may include a nonsingular, possibly time-dependent, matrix $M(t)$.

In solving the IVP (2.2), we first consider the corresponding problem in standard form,

$$\dot{y} = \hat{f}^E(t, y) + \hat{f}^I(t, y), \qquad y(t_0) = y_0, \tag{2.3}$$

where $\hat{f}^E(t, y) = M(t)^{-1}\, f^E(t, y)$ and $\hat{f}^I(t, y) = M(t)^{-1}\, f^I(t, y)$. ARKStep then utilizes variable-step, embedded, additive Runge-Kutta methods (ARK), corresponding to algorithms of the form

$$
\begin{aligned}
z_i &= y_{n-1} + h_n \sum_{j=1}^{i-1} A^E_{i,j} \hat{f}^E(t^E_{n,j}, z_j) + h_n \sum_{j=1}^{i} A^I_{i,j} \hat{f}^I(t^I_{n,j}, z_j), \quad i = 1, \ldots, s, \\
y_n &= y_{n-1} + h_n \sum_{i=1}^{s} \left(b^E_i \hat{f}^E(t^E_{n,i}, z_i) + b^I_i \hat{f}^I(t^I_{n,i}, z_i) \right), \\
\tilde{y}_n &= y_{n-1} + h_n \sum_{i=1}^{s} \left(\tilde{b}^E_i \hat{f}^E(t^E_{n,i}, z_i) + \tilde{b}^I_i \hat{f}^I(t^I_{n,i}, z_i) \right).
\end{aligned} \tag{2.4}
$$

Here $\tilde{y}_n$ are embedded solutions that approximate $y(t_n)$ and are used for error estimation; these typically have slightly lower accuracy than the computed solutions y_n. The internal stage times are abbreviated using the notation $t^E_{n,j} = t_{n-1} + c^E_j h_n$ and $t^I_{n,j} = t_{n-1} + c^I_j h_n$. The ARK method is primarily defined through the coefficients $A^E \in \mathbb{R}^{s\times s}$, $A^I \in \mathbb{R}^{s\times s}$, $b^E \in \mathbb{R}^s$, $b^I \in \mathbb{R}^s$, $c^E \in \mathbb{R}^s$ and $c^I \in \mathbb{R}^s$, that correspond with the explicit and implicit Butcher tables. Additional coefficients $\tilde{b}^E \in \mathbb{R}^s$ and $\tilde{b}^I \in \mathbb{R}^s$ are used to construct the embedding $\tilde{y}_n$. We note that ARKStep currently enforces the constraint that the explicit and implicit methods in an ARK pair must share the same number of stages, s. We note that when the problem has a time-independent mass matrix M, ARKStep allows the possibility for different explicit and implicit abcissae, i.e. c^E need not equal c^I.

The user of ARKStep must choose appropriately between one of three classes of methods: *ImEx*, *explicit*, and *implicit*. All of the built-in Butcher tables encoding the coefficients c^E, c^I, A^E, A^I, b^E, b^I, $\tilde{b}^E$ and $\tilde{b}^I$ are further described in the *Appendix: Butcher tables*.

For mixed stiff/nonstiff problems, a user should provide both of the functions f^E and f^I that define the IVP system. For such problems, ARKStep currently implements the ARK methods proposed in *[KC2003]*, allowing for methods having order of accuracy $q = \{3, 4, 5\}$; the tables for these methods are given in the section *Additive Butcher tables*. Additionally, user-defined ARK tables are supported.

For nonstiff problems, a user may specify that $f^I = 0$, i.e. the equation (2.2) reduces to the non-split IVP

$$
M(t)\,\dot{y} = f^E(t, y), \qquad y(t_0) = y_0. \tag{2.5}
$$

In this scenario, the coefficients $A^I = 0$, $c^I = 0$, $b^I = 0$ and $\tilde{b}^I = 0$ in (2.4), and the ARK methods reduce to classical explicit Runge-Kutta methods (ERK). For these classes of methods, ARKode provides coefficients with orders of accuracy $q = \{2, 3, 4, 5, 6, 8\}$, with embeddings of orders $p = \{1, 2, 3, 4, 5, 7\}$. These default to the *Heun-Euler-2-1-2*, *Bogacki-Shampine-4-2-3*, *Zonneveld-5-3-4*, *Cash-Karp-6-4-5*, *Verner-8-5-6* and *Fehlberg-13-7-8* methods, respectively. As with ARK methods, user-defined ERK tables are supported.

Alternately, for stiff problems the user may specify that $f^E = 0$, so the equation (2.2) reduces to the non-split IVP

$$
M(t)\,\dot{y} = f^I(t, y), \qquad y(t_0) = y_0. \tag{2.6}
$$

Similarly to ERK methods, in this scenario the coefficients $A^E = 0$, $c^E = 0$, $b^E = 0$ and $\tilde{b}^E = 0$ in (2.4), and the ARK methods reduce to classical diagonally-implicit Runge-Kutta methods (DIRK). For these classes of methods, ARKode provides tables with orders of accuracy $q = \{2, 3, 4, 5\}$, with embeddings of orders $p = \{1, 2, 3, 4\}$. These default to the *SDIRK-2-1-2*, *ARK-4-2-3 (implicit)*, *SDIRK-5-3-4* and *ARK-8-4-5 (implicit)* methods, respectively. Again, user-defined DIRK tables are supported.

2.4 ERKStep – Explicit Runge-Kutta methods

The ERKStep time-stepping module in ARKode is designed for IVP of the form

$$\dot{y} = f(t, y), \qquad y(t_0) = y_0, \tag{2.7}$$

i.e., unlike the more general problem form (2.2), ERKStep requires that problems have an identity mass matrix (i.e., $M(t) = I$) and that the right-hand side function is not split into separate components.

For such problems, ERKStep provides variable-step, embedded, explicit Runge-Kutta methods (ERK), corresponding to algorithms of the form

$$\begin{aligned} z_i &= y_{n-1} + h_n \sum_{j=1}^{i-1} A_{i,j} f(t_{n,j}, z_j), \quad i = 1, \ldots, s, \\ y_n &= y_{n-1} + h_n \sum_{i=1}^{s} b_i f(t_{n,i}, z_i), \\ \tilde{y}_n &= y_{n-1} + h_n \sum_{i=1}^{s} \tilde{b}_i f(t_{n,i}, z_i), \end{aligned} \tag{2.8}$$

where the variables have the same meanings as in the previous section.

Clearly, the problem (2.7) is fully encapsulated in the more general problem (2.5), and the algorithm (2.8) is similarly encapsulated in the more general algorithm (2.4). While it therefore follows that ARKStep can be used to solve every problem solvable by ERKStep, using the same set of methods, we include ERKStep as a distinct time-stepping module since this simplified form admits a more efficient and memory-friendly solution process than when considering the more general form (2.7).

2.5 MRIStep – Multirate infinitesimal step methods

The MRIStep time-stepping module in ARKode is designed for IVPs of the form

$$\dot{y} = f^S(t, y) + f^F(t, y), \qquad y(t_0) = y_0. \tag{2.9}$$

i.e. the right-hand side function is additively split into two components:

- $f^S(t, y)$ contains the "slow" components of the system (this will be integrated using a large time step h^S),
- $f^F(t, y)$ contains the "fast" components of the system (this will be integrated using small time steps $h^F \ll h^S$).

As with ERKStep, MRIStep currently requires that problems be posed with an identity mass matrix, $M(t) = I$.

For such problems, MRIStep provides fixed-step slow step multirate infinitesimal step and multirate infinitesimal GARK methods (see *[SKAW2009]*, *[SKAW2012a]*, *[SKAW2012b]*, and *[S2019]*) that combine two Runge-Kutta methods. The outer (slow) method derives from an s stage Runge-Kutta method where the stage values and the new solution are computed by solving an auxiliary ODE with an inner (fast) Runge-Kutta method. This corresponds to the following algorithm for a single step:

1. Set $z_1 = y_{n-1}$
2. For $i = 2, \ldots, s+1$
   (a) Let $v(0) = z_{i-1}$, $t^S_{n,i-1} = t_{n-1} + c^S_{i-1} h^S$, and $\Delta c^S_i = \left(c^S_i - c^S_{i-1}\right)$.
   (b) Let $r(\tau) = \sum_{j=1}^{i} \gamma_{i,j}\left(\tau/h^S\right) f^S(t^S_{n,j}, z_j)$

(c) For $\tau \in [0, h^S]$, solve $\dot{v}(\tau) = \Delta c_i^S f^F(t_{n,i-1}^S + \Delta c_i^S \tau, v) + r(\tau)$

(d) Set $z_i = v(h^S)$,

3. Set $y_n = z_{s+1}$.

where $c_{s+1}^S = 1$ and the coefficients $\gamma_{i,j}$ are polynomials in time that dictate the couplings from the slow to the fast time scale; these can be expressed as in *[S2019]*:

$$\gamma_{i,j}(\theta) = \sum_{k \geq 0} \gamma_{i,j}^{\{k\}} \theta^k, \tag{2.10}$$

and where the tables $\Gamma^{\{k\}} \in \mathbb{R}^{(s+1)\times(s+1)}$ define the slow-to-fast coupling. For traditional MIS methods (as in *[SKAW2009]*, *[SKAW2012a]*, and *[SKAW2012b]*), these coefficients are uniquely defined based on a slow Butcher table (A^S, b^S, c^S) having explicit first stage (i.e., $c_1^S = 0$ and $A_{1,j}^S = 0$ for $1 \leq j \leq s$), sorted abcissae (i.e., $c_i^S \geq c_{i-1}^S$ for $2 \leq i \leq s$), and final abcissa $c_s^S \leq 1$ as:

$$\gamma_{i,j}^{\{0\}} = \begin{cases} 0, & \text{if } i = 1, \\ A_{i,j}^S - A_{i-1,j}^S, & \text{if } 2 \leq i \leq s, \\ b_j^S - A_{s,j}^S, & \text{if } i = s+1. \end{cases} \tag{2.11}$$

For general slow tables (A^S, b^S, c^S) with at least second-order accuracy, the corresponding MIS method will be second order. However, if this slow table is at least third order and satisfies the additional condition

$$\sum_{i=2}^{s} \left(c_i^S - c_{i-1}^S\right) \left(\mathbf{e}_i + \mathbf{e}_{i-1}\right)^T A^S c^S + \left(1 - c_s^S\right) \left(\frac{1}{2} + \mathbf{e}_s^T A^S c^S\right) = \frac{1}{3}, \tag{2.12}$$

where $\mathbf{e}_j$ corresponds to the j-th column from the identity matrix, then the overall MIS method will be third order.

As with standard Runge–Kutta methods, implicitness at the slow time scale is characterized by nonzero values on or above the diagonal of the matrices $\Gamma^{\{k\}}$. Typically, MRI methods are at most diagonally-implicit (i.e., $\gamma_{i,j}^{\{k\}} = 0$ for all $j > i$). Additionally, an implicit stage i may be characterized as being "solve-decoupled," wherein $c_i^S - c_{i-1}^S = 0$ and thus the 'fast' IVP for v over $\tau \in [0, h^S]$ may be solved analytically,

$$\begin{aligned} & z_i = z_{i-1} + \int_0^{h^S} r(\tau)\, \mathrm{d}\tau \\ \Leftrightarrow \\ & z_i = z_{i-1} + h^S \sum_{j=1}^{i} \left(\sum_{k \geq 0} \frac{\gamma_{i,j}^{\{k\}}}{k+1} \right) f^S(t_{n,j}^S, z_j), \end{aligned} \tag{2.13}$$

corresponding to a standard diagonally-implicit Runge–Kutta stage. Alternately, an implicit MRI stage i is considered "solve-coupled" if both $c_i^S - c_{i-1}^S \neq 0$ and $\sum_{k \geq 0} \frac{\gamma_{i,j}^{\{k\}}}{k+1} \neq 0$, in which case the stage solution z_i is *both* an input to $r(\tau)$ and the result of time-evolution of the fast IVP, necessitating an implicit solve that is coupled to the 'fast' solver.

The default method supported by the MRIStep module is the explicit, third-order MIS method defined by the slow Butcher table (*Knoth-Wolke-3-3*); however, other slow Butcher tables (A^S, b^S, c^S) or coupling tables $\Gamma^{\{k\}} \in \mathbb{R}^{(s+1)\times(s+1)}$ may be provided. At present, only 'solve-decoupled' implicit MRI methods are supported.

At present, the inner ODEs for step 2c of the MRI algorithm must be solved using the ARKStep module. As such, this can be evolved using either an explicit, implicit, or IMEX method with adaptive or fixed time steps.

2.6 Error norms

In the process of controlling errors at various levels (time integration, nonlinear solution, linear solution), the methods in ARKode use a weighted root-mean-square norm, denoted $\|\cdot\|_{\text{WRMS}}$, for all error-like quantities,

$$\|v\|_{\text{WRMS}} = \left(\frac{1}{N}\sum_{i=1}^{N}\left(v_i\, w_i\right)^2\right)^{1/2}. \tag{2.14}$$

The utility of this norm arises in the specification of the weighting vector w, that combines the units of the problem with user-supplied values that specify an "acceptable" level of error. To this end, we construct an error weight vector using the most-recent step solution and user-supplied relative and absolute tolerances, namely

$$w_i = \left(RTOL \cdot |y_{n-1,i}| + ATOL_i\right)^{-1}. \tag{2.15}$$

Since $1/w_i$ represents a tolerance in the i-th component of the solution vector y, a vector whose WRMS norm is 1 is regarded as "small." For brevity, unless specified otherwise we will drop the subscript WRMS on norms in the remainder of this section.

Additionally, for problems involving a non-identity mass matrix, $M \neq I$, the units of equation (2.2) may differ from the units of the solution y. In this case, we may additionally construct a residual weight vector,

$$w_i = \left(RTOL \cdot \left|\left(M(t_{n-1})\, y_{n-1}\right)_i\right| + ATOL'_i\right)^{-1}, \tag{2.16}$$

where the user may specify a separate absolute residual tolerance value or array, $ATOL'$. The choice of weighting vector used in any given norm is determined by the quantity being measured: values having "solution" units use (2.15), whereas values having "equation" units use (2.16). Obviously, for problems with $M = I$, the solution and equation units are identical, so the solvers in ARKode will use (2.15) when computing all error norms.

2.7 Time step adaptivity

A critical component of IVP "solvers" (rather than just time-steppers) is their adaptive control of local truncation error (LTE). At every step, we estimate the local error, and ensure that it satisfies tolerance conditions. If this local error test fails, then the step is recomputed with a reduced step size. To this end, the Runge-Kutta methods packaged within both the ARKStep and ERKStep modules admit an embedded solution $\tilde{y}_n$, as shown in equations (2.4) and (2.8). Generally, these embedded solutions attain a slightly lower order of accuracy than the computed solution y_n. Denoting the order of accuracy for y_n as q and for $\tilde{y}_n$ as p, most of these embedded methods satisfy $p = q - 1$. These values of q and p correspond to the *global* orders of accuracy for the method and embedding, hence each admit local truncation errors satisfying *[HW1993]*

$$\begin{aligned} \|y_n - y(t_n)\| &= Ch_n^{q+1} + \mathcal{O}(h_n^{q+2}), \\ \|\tilde{y}_n - y(t_n)\| &= Dh_n^{p+1} + \mathcal{O}(h_n^{p+2}), \end{aligned} \tag{2.17}$$

where C and D are constants independent of h_n, and where we have assumed exact initial conditions for the step, i.e. $y_{n-1} = y(t_{n-1})$. Combining these estimates, we have

$$\|y_n - \tilde{y}_n\| = \|y_n - y(t_n) - \tilde{y}_n + y(t_n)\| \leq \|y_n - y(t_n)\| + \|\tilde{y}_n - y(t_n)\| \leq Dh_n^{p+1} + \mathcal{O}(h_n^{p+2}).$$

We therefore use the norm of the difference between y_n and $\tilde{y}_n$ as an estimate for the LTE at the step n

$$T_n = \beta\,(y_n - \tilde{y}_n) = \beta h_n \sum_{i=1}^{s} \left[\left(b_i^E - \tilde{b}_i^E\right) \hat{f}^E(t_{n,i}^E, z_i) + \left(b_i^I - \tilde{b}_i^I\right) \hat{f}^I(t_{n,i}^I, z_i) \right] \tag{2.18}$$

for ARK methods, and similarly for ERK methods. Here, $\beta > 0$ is an error *bias* to help account for the error constant D; the default value of this constant is $\beta = 1.5$, which may be modified by the user.

With this LTE estimate, the local error test is simply $\|T_n\| < 1$ since this norm includes the user-specified tolerances. If this error test passes, the step is considered successful, and the estimate is subsequently used to estimate the next step size, the algorithms used for this purpose are described below in the section *Asymptotic error control*. If the error test fails, the step is rejected and a new step size h' is then computed using the same error controller as for successful steps. A new attempt at the step is made, and the error test is repeated. If the error test fails twice, then h'/h is limited above to 0.3, and limited below to 0.1 after an additional step failure. After seven error test failures, control is returned to the user with a failure message. We note that all of the constants listed above are only the default values; each may be modified by the user.

We define the step size ratio between a prospective step h' and a completed step h as η, i.e. $\eta = h'/h$. This value is subsequently bounded from above by $\eta_{\max}$ to ensure that step size adjustments are not overly aggressive. This upper bound changes according to the step and history,

$$\eta_{\max} = \begin{cases} \text{etamx1}, & \text{on the first step (default is 10000),} \\ \text{growth}, & \text{on general steps (default is 20),} \\ 1, & \text{if the previous step had an error test failure.} \end{cases}$$

A flowchart detailing how the time steps are modified at each iteration to ensure solver convergence and successful steps is given in the figure below. Here, all norms correspond to the WRMS norm, and the error adaptivity function **arkAdapt** is supplied by one of the error control algorithms discussed in the subsections below.

For some problems it may be preferable to avoid small step size adjustments. This can be especially true for problems that construct a Newton Jacobian matrix or a preconditioner for a nonlinear or an iterative linear solve, where this construction is computationally expensive, and where convergence can be seriously hindered through use of an inaccurate matrix. To accommodate these scenarios, the step is left unchanged when $\eta \in [\eta_L, \eta_U]$. The default values for this interval are $\eta_L = 1$ and $\eta_U = 1.5$, and may be modified by the user.

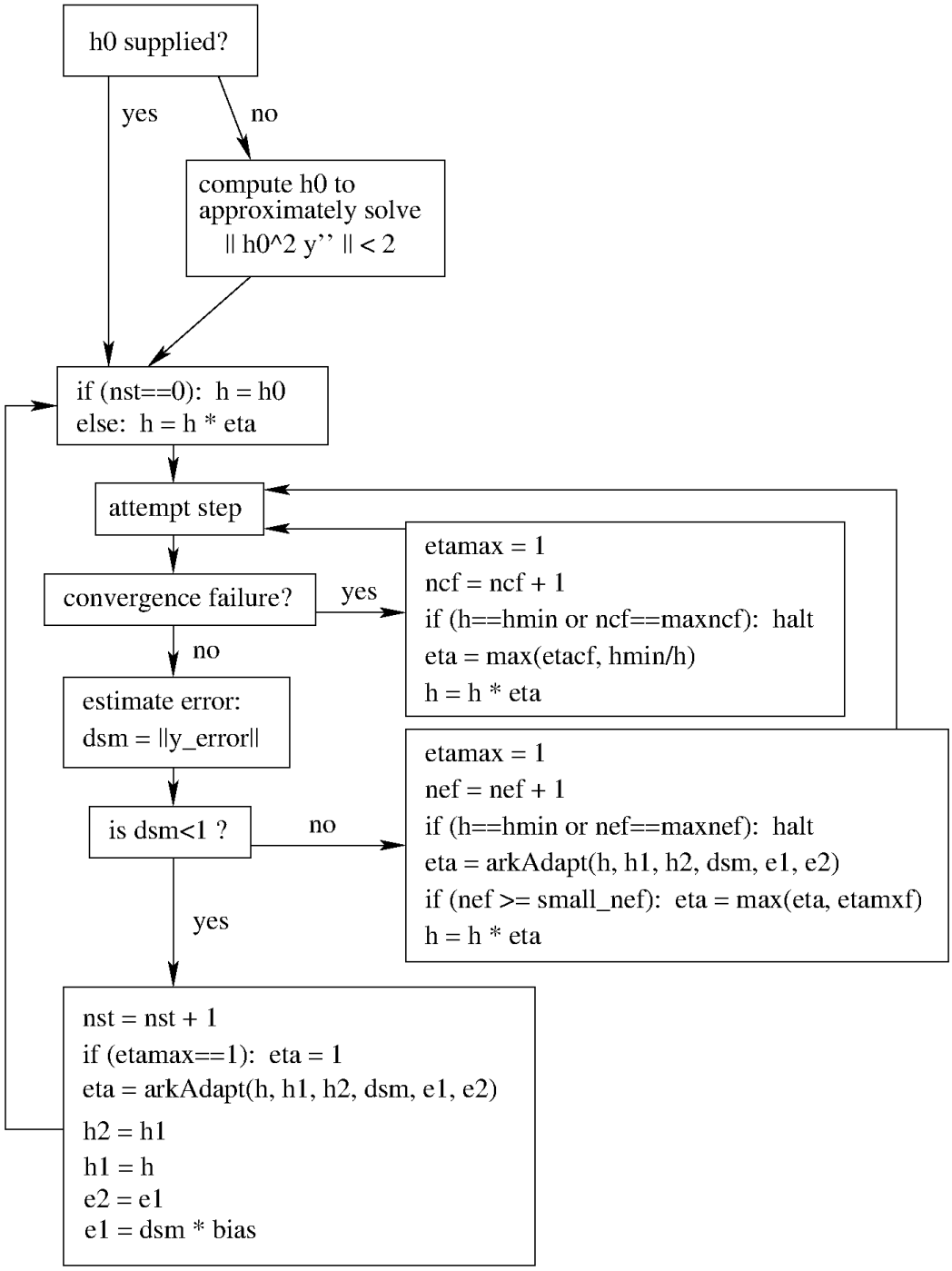
We note that any choices for η (or equivalently, h') are subsequently constrained by the optional user-supplied bounds $h_{\min}$ and $h_{\max}$. Additionally, the time-stepping algorithms in ARKode may similarly limit h' to adhere to a user-provided "TSTOP" stopping point, t_{stop}.

2.7.1 Asymptotic error control

As mentioned above, the time-stepping modules in ARKode adapt the step size in order to attain local errors within desired tolerances of the true solution. These adaptivity algorithms estimate the prospective step size h' based on the asymptotic local error estimates (2.17). We define the values ε_n, ε_{n-1} and ε_{n-2} as

$$\varepsilon_k \;\equiv\; \|T_k\| \;=\; \beta\|y_k - \tilde{y}_k\|,$$

corresponding to the local error estimates for three consecutive steps, $t_{n-3} \to t_{n-2} \to t_{n-1} \to t_n$. These local error history values are all initialized to 1 upon program initialization, to accommodate the few initial time steps of a calculation where some of these error estimates have not yet been computed. With these estimates, ARKode supports a variety of error control algorithms, as specified in the subsections below.


h0 supplied?
yes
no
compute h0 to approximately solve || h0^2 y'' || < 2
if (nst==0): h = h0
else: h = h * eta
attempt step
convergence failure?
yes
etamax = 1
ncf = ncf + 1
if (h==hmin or ncf==maxncf): halt
eta = max(etacf, hmin/h)
h = h * eta
no
estimate error:
dsm = ||y_error||
is dsm<1 ?
no
etamax = 1
nef = nef + 1
if (h==hmin or nef==maxnef): halt
eta = arkAdapt(h, h1, h2, dsm, e1, e2)
if (nef >= small_nef): eta = max(eta, etamxf)
h = h * eta
yes
nst = nst + 1
if (etamax==1): eta = 1
eta = arkAdapt(h, h1, h2, dsm, e1, e2)
h2 = h1
h1 = h
e2 = e1
e1 = dsm * bias

2.7.1.1 PID controller

This is the default time adaptivity controller used by the ARKStep and ERKStep modules. It derives from those found in *[KC2003]*, *[S1998]*, *[S2003]* and *[S2006]*, and uses all three of the local error estimates ε_n, ε_{n-1} and ε_{n-2} in determination of a prospective step size,

$$h' \;=\; h_n\, \varepsilon_n^{-k_1/p}\, \varepsilon_{n-1}^{k_2/p}\, \varepsilon_{n-2}^{-k_3/p},$$

where the constants k_1, k_2 and k_3 default to 0.58, 0.21 and 0.1, respectively, and may be modied by the user. In this estimate, a floor of $\varepsilon > 10^{-10}$ is enforced to avoid division-by-zero errors.

2.7.1.2 PI controller

Like with the previous method, the PI controller derives from those found in *[KC2003]*, *[S1998]*, *[S2003]* and *[S2006]*, but it differs in that it only uses the two most recent step sizes in its adaptivity algorithm,

$$h' \;=\; h_n\, \varepsilon_n^{-k_1/p}\, \varepsilon_{n-1}^{k_2/p}.$$

Here, the default values of k_1 and k_2 default to 0.8 and 0.31, respectively, though they may be changed by the user.

2.7.1.3 I controller

This is the standard time adaptivity control algorithm in use by most publicly-available ODE solver codes. It bases the prospective time step estimate entirely off of the current local error estimate,

$$h' \;=\; h_n\, \varepsilon_n^{-k_1/p}.$$

By default, $k_1 = 1$, but that may be modified by the user.

2.7.1.4 Explicit Gustafsson controller

This step adaptivity algorithm was proposed in *[G1991]*, and is primarily useful with explicit Runge-Kutta methods. In the notation of our earlier controllers, it has the form

$$h' \;=\; \begin{cases} h_1\, \varepsilon_1^{-1/p}, & \text{on the first step,} \\ h_n\, \varepsilon_n^{-k_1/p}\, \left(\varepsilon_n/\varepsilon_{n-1}\right)^{k_2/p}, & \text{on subsequent steps.} \end{cases} \tag{2.19}$$

The default values of k_1 and k_2 are 0.367 and 0.268, respectively, and may be modified by the user.

2.7.1.5 Implicit Gustafsson controller

A version of the above controller suitable for implicit Runge-Kutta methods was introduced in *[G1994]*, and has the form

$$h' = \begin{cases} h_1 \varepsilon_1^{-1/p}, & \text{on the first step,} \\ h_n\, \left(h_n/h_{n-1}\right) \varepsilon_n^{-k_1/p}\, \left(\varepsilon_n/\varepsilon_{n-1}\right)^{-k_2/p}, & \text{on subsequent steps.} \end{cases} \tag{2.20}$$

The algorithm parameters default to $k_1 = 0.98$ and $k_2 = 0.95$, but may be modified by the user.

2.7.1.6 ImEx Gustafsson controller

An ImEx version of these two preceding controllers is also available. This approach computes the estimates h'_1 arising from equation (2.19) and the estimate h'_2 arising from equation (2.20), and selects

$$h' = \frac{h}{|h|} \min\left\{|h'_1|, |h'_2|\right\}.$$

Here, equation (2.19) uses k_1 and k_2 with default values of 0.367 and 0.268, while equation (2.20) sets both parameters to the input k_3 that defaults to 0.95. All of these values may be modified by the user.

2.7.1.7 User-supplied controller

Finally, ARKode's time-stepping modules allow the user to define their own time step adaptivity function,

$$h' = H(y, t, h_n, h_{n-1}, h_{n-2}, \varepsilon_n, \varepsilon_{n-1}, \varepsilon_{n-2}, q, p),$$

to allow for problem-specific choices, or for continued experimentation with temporal error controllers.

2.8 Explicit stability

For problems that involve a nonzero explicit component, i.e. $f^E(t, y) \neq 0$ in ARKStep or for any problem in ERKStep, explicit and ImEx Runge-Kutta methods may benefit from additional user-supplied information regarding the explicit stability region. All ARKode adaptivity methods utilize estimates of the local error, and it is often the case that such local error control will be sufficient for method stability, since unstable steps will typically exceed the error control tolerances. However, for problems in which $f^E(t, y)$ includes even moderately stiff components, and especially for higher-order integration methods, it may occur that a significant number of attempted steps will exceed the error tolerances. While these steps will automatically be recomputed, such trial-and-error can result in an unreasonable number of failed steps, increasing the cost of the computation. In these scenarios, a stability-based time step controller may also be useful.

Since the maximum stable explicit step for any method depends on the problem under consideration, in that the value $(h_n\lambda)$ must reside within a bounded stability region, where λ are the eigenvalues of the linearized operator $\partial f^E/\partial y$, information on the maximum stable step size is not readily available to ARKode's time-stepping modules. However, for many problems such information may be easily obtained through analysis of the problem itself, e.g. in an advection-diffusion calculation f^I may contain the stiff diffusive components and f^E may contain the comparably nonstiff advection terms. In this scenario, an explicitly stable step h_{exp} would be predicted as one satisfying the Courant-Friedrichs-Lewy (CFL) stability condition for the advective portion of the problem,

$$|h_{\text{exp}}| < \frac{\Delta x}{|\lambda|}$$

where Δx is the spatial mesh size and λ is the fastest advective wave speed.

In these scenarios, a user may supply a routine to predict this maximum explicitly stable step size, $|h_{\text{exp}}|$. If a value for $|h_{\text{exp}}|$ is supplied, it is compared against the value resulting from the local error controller, $|h_{\text{acc}}|$, and the eventual time step used will be limited accordingly,

$$h' = \frac{h}{|h|} \min\{c\,|h_{\text{exp}}|,\, |h_{\text{acc}}|\}.$$

Here the explicit stability step factor $c > 0$ (often called the "CFL number") defaults to $1/2$ but may be modified by the user.

2.8.1 Fixed time stepping

While both the ARKStep and ERKStep time-stepping modules are designed for tolerance-based time step adaptivity, they additionally support a "fixed-step" mode (*note: fixed-step mode is currently required for the slow time scale in the MRIStep module*). This mode is typically used for debugging purposes, for verification against hand-coded Runge-Kutta methods, or for problems where the time steps should be chosen based on other problem-specific information. In this mode, all internal time step adaptivity is disabled:

- temporal error control is disabled,
- nonlinear or linear solver non-convergence will result in an error (instead of a step size adjustment),
- no check against an explicit stability condition is performed.

Additional information on this mode is provided in the sections *ARKStep Optional Inputs*, *ERKStep Optional Inputs*, and *MRIStep Optional Inputs*.

2.9 Algebraic solvers

When solving a problem involving either an implicit component (e.g., in ARKStep with $f^I(t,y) \neq 0$, or in MRIStep with a solve-decoupled implicit slow stage), or a non-identity mass matrix ($M(t) \neq I$ in ARKStep), systems of linear or nonlinear algebraic equations must be solved at each stage and/or step of the method. This section therefore focuses on the variety of mathematical methods provided in the ARKode infrastructure for such problems, including *nonlinear solvers*, *linear solvers*, *preconditioners*, *iterative solver error control*, *implicit predictors*, and techniques used for simplifying the above solves when using different classes of *mass-matrices*.

2.9.1 Nonlinear solver methods

For the DIRK and ARK methods corresponding to (2.2) and (2.6) in ARKStep, and the solve-decoupled implicit slow stages (2.13) in MRIStep, an implicit system

$$G(z_i) = 0 \tag{2.21}$$

must be solved for each stage $z_i, i = 1, \ldots, s$. In order to maximize solver efficiency, we define this root-finding problem differently based on the type of mass-matrix supplied by the user.

- In the case that $M = I$ within ARKStep, we define the residual as

$$G(z_i) \equiv z_i - h_n A^I_{i,i} f^I(t^I_{n,i}, z_i) - a_i, \tag{2.22}$$

  where we have the data

$$a_i \equiv y_{n-1} + h_n \sum_{j=1}^{i-1} \left[A^E_{i,j} f^E(t^E_{n,j}, z_j) + A^I_{i,j} f^I(t^I_{n,j}, z_j) \right].$$

- In the case of non-identity mass matrix $M \neq I$ within ARKStep, but where M is independent of t, we define the residual as

$$G(z_i) \equiv M z_i - h_n A^I_{i,i} f^I(t^I_{n,i}, z_i) - a_i, \tag{2.23}$$

where we have the data

$$a_i \equiv My_{n-1} + h_n \sum_{j=1}^{i-1} \left[A^E_{i,j} f^E(t^E_{n,j}, z_j) + A^I_{i,j} f^I(t^I_{n,j}, z_j) \right].$$

Note: This form of residual, as opposed to $G(z_i) = z_i - h_n A^I_{i,i} \hat{f}^I(t^I_{n,i}, z_i) - a_i$ (with a_i defined appropriately), removes the need to perform the nonlinear solve with right-hand side function $\hat{f}^I = M^{-1} f^I$, as that would require a linear solve with M at *every evaluation* of the implicit right-hand side routine.

- In the case of ARKStep with M dependent on t, we define the residual as

$$G(z_i) \equiv M(t^I_{n,i})(z_i - a_i) - h_n A^I_{i,i} f^I(t^I_{n,i}, z_i) \tag{2.24}$$

where we have the data

$$a_i \equiv y_{n-1} + h_n \sum_{j=1}^{i-1} \left[A^E_{i,j} \hat{f}^E(t^E_{n,j}, z_j) + A^I_{i,j} \hat{f}^I(t^I_{n,j}, z_j) \right].$$

Note: As above, this form of the residual is chosen to remove excessive mass-matrix solves from the nonlinear solve process.

- Similarly, in MRIStep (that always assumes $M = I$), we have the residual

$$G(z_i) \equiv z_i - h^S \left(\sum_{k \geq 0} \frac{\gamma^{\{k\}}_{i,i}}{k+1} \right) f^S(t^S_{n,i}, z_i) - a_i = 0 \tag{2.25}$$

where

$$a_i \equiv z_{i-1} + h^S \sum_{j=1}^{i-1} \left(\sum_{k \geq 0} \frac{\gamma^{\{k\}}_{i,j}}{k+1} \right) f^S(t^S_{n,j}, z_j).$$

In each of the above nonlinear residual functions, if $f^I(t, y)$ or $f^S(t, y)$ depends nonlinearly on y then (2.21) corresponds to a nonlinear system of equations; if instead $f^I(t, y)$ or $f^S(t, y)$ depends linearly on y then this is a linear system of equations.

To solve each of the above root-finding problems ARKode provides a choice of strategies, with the default being a variant of Newton's method,

$$z_i^{(m+1)} = z_i^{(m)} + \delta^{(m+1)}, \tag{2.26}$$

where m is the Newton iteration index, and the Newton update $\delta^{(m+1)}$ in turn requires the solution of the Newton linear system

$$\mathcal{A}\left(t^I_{n,i}, z_i^{(m)}\right) \delta^{(m+1)} = -G\left(z_i^{(m)}\right), \tag{2.27}$$

in which

$$\mathcal{A}(t,z) \approx M(t) - \gamma J(t,z), \quad J(t,z) = \frac{\partial f^I(t,z)}{\partial z}, \quad \text{and} \quad \gamma = h_n A^I_{i,i} \tag{2.28}$$

within ARKStep, or

$$\mathcal{A}(t,z) \approx I - \gamma J(t,z), \quad J(t,z) = \frac{\partial f^S(t,z)}{\partial z}, \quad \text{and} \quad \gamma = h^S \sum_{k\geq 0} \frac{\gamma^{\{k\}}_{i,i}}{k+1} \tag{2.29}$$

within MRIStep.

As an alternative to Newton's method, ARKode provides a fixed point iteration for solving the stages $z_i, i = 1,\ldots,s$,

$$z_i^{(m+1)} = \Phi\left(z_i^{(m)}\right) \equiv z_i^{(m)} - M(t^I_{n,i})^{-1}\, G\left(z_i^{(m)}\right), \quad m = 0,1,\ldots. \tag{2.30}$$

This iteration may additionally be improved using a technique called "Anderson acceleration" *[WN2011]*. Unlike with Newton's method, these methods *do not* require the solution of a linear system involving the Jacobian of f at each iteration, instead opting for solution of a low-dimensional least-squares solution to construct the nonlinear update.

Finally, if the user specifies that $f^I(t,y)$ or $f^S(t,y)$ depend linearly on y in ARKStep or MRIStep, respectively, and if the Newton-based nonlinear solver is chosen, then the problem (2.21) will be solved using only a single Newton iteration. In this case, an additional user-supplied argument indicates whether this Jacobian is time-dependent or not, signaling whether the Jacobian or preconditioner needs to be recomputed at each stage or time step, or if it can be reused throughout the full simulation.

The optimal choice of solver (Newton vs fixed-point) is highly problem dependent. Since fixed-point solvers do not require the solution of linear systems involving the Jacobian of f, each iteration may be significantly less costly than their Newton counterparts. However, this can come at the cost of slower convergence (or even divergence) in comparison with Newton-like methods. On the other hand, these fixed-point solvers do allow for user specification of the Anderson-accelerated subspace size, m_k. While the required amount of solver memory for acceleration grows proportionately to $m_k N$, larger values of m_k may result in faster convergence. In our experience, this improvement is most significant for relatively modest values, e.g. $1 \leq m_k \leq 5$, and that larger values of m_k may not result in improved convergence.

While a Newton-based iteration is the default solver in ARKode due to its increased robustness on very stiff problems, we strongly recommend that users also consider the fixed-point solver when attempting a new problem.

For either the Newton or fixed-point solvers, it is well-known that both the efficiency and robustness of the algorithm intimately depend on the choice of a good initial guess. The initial guess for these solvers is a prediction $z_i^{(0)}$ that is computed explicitly from previously-computed data (e.g. y_{n-2}, y_{n-1}, and z_j where $j < i$). Additional information on the specific predictor algorithms is provided in the following section, *Implicit predictors*.

2.9.2 Linear solver methods

When a Newton-based method is chosen for solving each nonlinear system, a linear system of equations must be solved at each nonlinear iteration. For this solve ARKode provides several choices, including the option of a user-supplied linear solver module. The linear solver modules distributed with SUNDIALS are organized into two families: a *direct* family comprising direct linear solvers for dense, banded or sparse matrices, and a *spils* family comprising scaled, preconditioned, iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal SUNDIALS implementation or a BLAS/LAPACK implementation (serial version only),
- band direct solvers, using either an internal SUNDIALS implementation or a BLAS/LAPACK implementation (serial version only),
- sparse direct solvers, using either the KLU sparse matrix library *[KLU]*, or the OpenMP or PThreads-enabled SuperLU_MT sparse matrix library *[SuperLUMT]* [Note that users will need to download and install the KLU or SuperLU_MT packages independent of ARKode],
- SPGMR, a scaled, preconditioned GMRES (Generalized Minimal Residual) solver,
- SPFGMR, a scaled, preconditioned FGMRES (Flexible Generalized Minimal Residual) solver,
- SPBCGS, a scaled, preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable) solver,
- SPTFQMR, a scaled, preconditioned TFQMR (Transpose-free Quasi-Minimal Residual) solver, or
- PCG, a preconditioned CG (Conjugate Gradient method) solver for symmetric linear systems.

For large stiff systems where direct methods are often infeasible, the combination of an implicit integrator and a preconditioned Krylov method can yield a powerful tool because it combines established methods for stiff integration, nonlinear solver iteration, and Krylov (linear) iteration with a problem-specific treatment of the dominant sources of stiffness, in the form of a user-supplied preconditioner matrix *[BH1989]*. We note that the direct linear solver modules currently provided by SUNDIALS are only designed to be used with the serial and threaded vector representations.

2.9.2.1 Matrix-based linear solvers

In the case that a matrix-based linear solver is used, a *modified Newton iteration* is utilized. In a modified newton iteration, the matrix $\mathcal{A}$ is held fixed for multiple Newton iterations. More precisely, each Newton iteration is computed from the modified equation

$$\tilde{\mathcal{A}}\left(\tilde{t},\tilde{z}\right)\,\delta^{(m+1)} = -G\left(z_i^{(m)}\right), \tag{2.31}$$

in which

$$\tilde{\mathcal{A}}(\tilde{t},\tilde{z}) \approx M(\tilde{t}) - \tilde{\gamma}J(\tilde{t},\tilde{z}), \quad \text{and} \quad \tilde{\gamma} = \tilde{h}A^I_{i,i} \quad \text{(ARKStep)} \tag{2.32}$$

or

$$\tilde{\mathcal{A}}(\tilde{t},\tilde{z}) \approx I - \tilde{\gamma}J(\tilde{t},\tilde{z}), \quad \text{and} \quad \tilde{\gamma} = \tilde{h}\sum_{k\geq 0}\frac{\gamma^{\{k\}}_{i,i}}{k+1} \quad \text{(MRIStep)}. \tag{2.33}$$

Here, the solution $\tilde{z}$, time $\tilde{t}$, and step size $\tilde{h}$ upon which the modified equation rely, are merely values of these quantities from a previous iteration. In other words, the matrix $\tilde{\mathcal{A}}$ is only computed rarely, and reused for repeated solves. The frequency at which $\tilde{\mathcal{A}}$ is recomputed defaults to 20 time steps, but may be modified by the user.

When using the dense and band SUNMatrix objects for the linear systems (2.31), the Jacobian J may be supplied by a user routine, or approximated internally by finite-differences. In the case of differencing, we use the standard approximation

$$J_{i,j}(t,z) \approx \frac{f_i^*(t,z+\sigma_j e_j) - f_i^*(t,z)}{\sigma_j},$$

where f^* is either f^I for ARKStep or f^S for MRIStep, e_j is the j-th unit vector, and the increments σ_j are given by

$$\sigma_j = \max\left\{\sqrt{U}\,|z_j|, \frac{\sigma_0}{w_j}\right\}.$$

Here U is the unit roundoff, σ_0 is a small dimensionless value, and w_j is the error weight defined in (2.15). In the dense case, this approach requires N evaluations of f^*, one for each column of J. In the band case, the columns of J are computed in groups, using the Curtis-Powell-Reid algorithm, with the number of f^* evaluations equal to the matrix bandwidth.

We note that with sparse and user-supplied SUNMatrix objects, the Jacobian *must* be supplied by a user routine.

2.9.2.2 Matrix-free iterative linear solvers

In the case that a matrix-free iterative linear solver is chosen, an *inexact Newton iteration* is utilized. Here, the matrix $\mathcal{A}$ is not itself constructed since the algorithms only require the product of this matrix with a given vector. Additionally, each Newton system (2.27) is not solved completely, since these linear solvers are iterative (hence the "inexact" in the name). As a result. for these linear solvers $\mathcal{A}$ is applied in a matrix-free manner,

$$\mathcal{A}(t,z)\,v = M(t)\,v - \gamma\,J(t,z)\,v.$$

The mass matrix-vector products Mv *must* be provided through a user-supplied routine; the Jacobian matrix-vector products Jv are obtained by either calling an optional user-supplied routine, or through a finite difference approximation to the directional derivative:

$$J(t,z)\,v \approx \frac{f^*(t,z+\sigma v) - f^*(t,z)}{\sigma},$$

where again f^* is either f^I for ARKStep or f^S for MRIStep, and we use the increment $\sigma = 1/\|v\|$ to ensure that $\|\sigma v\| = 1$.

As with the modified Newton method that reused $\mathcal{A}$ between solves, the inexact Newton iteration may also recompute the preconditioner P infrequently to balance the high costs of matrix construction and factorization against the reduced convergence rate that may result from a stale preconditioner.

2.9.2.3 Updating the linear solver

In cases where recomputation of the Newton matrix $\tilde{\mathcal{A}}$ or preconditioner P is lagged, these structures will be recomputed only in the following circumstances:

- when starting the problem,
- when more than 20 steps have been taken since the last update (this value may be modified by the user),
- when the value $\tilde{\gamma}$ of γ at the last update satisfies $|\gamma/\tilde{\gamma} - 1| > 0.2$ (this value may be modified by the user),
- when a non-fatal convergence failure just occurred,
- when an error test failure just occurred, or
- if the problem is linearly implicit and γ has changed by a factor larger than 100 times machine epsilon.

When an update is forced due to a convergence failure, an update of $\tilde{\mathcal{A}}$ or P may or may not involve a re-evaluation of J (in $\tilde{\mathcal{A}}$) or of Jacobian data (in P), depending on whether errors in the Jacobian were the likely cause of the failure. More generally, the decision is made to re-evaluate J (or instruct the user to update P) when:

- starting the problem,
- more than 50 steps have been taken since the last evaluation,

- a convergence failure occurred with an outdated matrix, and the value $\tilde{\gamma}$ of γ at the last update satisfies $|\gamma/\tilde{\gamma} - 1| > 0.2$,
- a convergence failure occurred that forced a step size reduction, or
- if the problem is linearly implicit and γ has changed by a factor larger than 100 times machine epsilon.

However, for linear solvers and preconditioners that do not rely on costly matrix construction and factorization operations (e.g. when using a geometric multigrid method as preconditioner), it may be more efficient to update these structures more frequently than the above heuristics specify, since the increased rate of linear/nonlinear solver convergence may more than account for the additional cost of Jacobian/preconditioner construction. To this end, a user may specify that the system matrix $\mathcal{A}$ and/or preconditioner P should be recomputed more frequently.

As will be further discussed in the section *Preconditioning*, in the case of most Krylov methods, preconditioning may be applied on the left, right, or on both sides of $\mathcal{A}$, with user-supplied routines for the preconditioner setup and solve operations.

2.9.3 Iteration Error Control

2.9.3.1 Nonlinear iteration error control

The stopping test for all of the nonlinear solver algorithms is related to the temporal local error test, with the goal of keeping the nonlinear iteration errors from interfering with local error control. Denoting the final computed value of each stage solution as $z_i^{(m)}$, and the true stage solution solving (2.21) as z_i, we want to ensure that the iteration error $z_i - z_i^{(m)}$ is "small" (recall that a norm less than 1 is already considered within an acceptable tolerance).

To this end, we first estimate the linear convergence rate R_i of the nonlinear iteration. We initialize $R_i = 1$, and reset it to this value whenever $\tilde{\mathcal{A}}$ or P are updated. After computing a nonlinear correction $\delta^{(m)} = z_i^{(m)} - z_i^{(m-1)}$, if $m > 0$ we update R_i as

$$R_i \leftarrow \max\left\{0.3R_i, \left\|\delta^{(m)}\right\| / \left\|\delta^{(m-1)}\right\|\right\}.$$

where the factor 0.3 is user-modifiable.

Let $y_n^{(m)}$ denote the time-evolved solution constructed using our approximate nonlinear stage solutions, $z_i^{(m)}$, and let $y_n^{(\infty)}$ denote the time-evolved solution constructed using *exact* nonlinear stage solutions. We then use the estimate

$$\left\|y_n^{(\infty)} - y_n^{(m)}\right\| \approx \max_i \left\|z_i^{(m+1)} - z_i^{(m)}\right\| \approx \max_i R_i \left\|z_i^{(m)} - z_i^{(m-1)}\right\| = \max_i R_i \left\|\delta^{(m)}\right\|.$$

Therefore our convergence (stopping) test for the nonlinear iteration for each stage is

$$R_i \left\|\delta^{(m)}\right\| < \epsilon, \tag{2.34}$$

where the factor ϵ has default value 0.1. We default to a maximum of 3 nonlinear iterations. We also declare the nonlinear iteration to be divergent if any of the ratios $\|\delta^{(m)}\|/\|\delta^{(m-1)}\| > 2.3$ with $m > 0$. If convergence fails in the fixed point iteration, or in the Newton iteration with J or $\mathcal{A}$ current, we reduce the step size h_n by a factor of 0.25. The integration will be halted after 10 convergence failures, or if a convergence failure occurs with $h_n = h_{\min}$. However, since the nonlinearity of (2.21) may vary significantly based on the problem under consideration, these default constants may all be modified by the user.

2.9.3.2 Linear iteration error control

When a Krylov method is used to solve the linear Newton systems (2.27), its errors must also be controlled. To this end, we approximate the linear iteration error in the solution vector $\delta^{(m)}$ using the preconditioned residual vector,

e.g. $r = P\mathcal{A}\delta^{(m)} + PG$ for the case of left preconditioning (the role of the preconditioner is further elaborated in the next section). In an attempt to ensure that the linear iteration errors do not interfere with the nonlinear solution error and local time integration error controls, we require that the norm of the preconditioned linear residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}. \tag{2.35}$$

Here ϵ is the same value as that is used above for the nonlinear error control. The factor of 10 is used to ensure that the linear solver error does not adversely affect the nonlinear solver convergence. Smaller values for the parameter ϵ_L are typically useful for strongly nonlinear or very stiff ODE systems, while easier ODE systems may benefit from a value closer to 1. The default value is $\epsilon_L = 0.05$, which may be modified by the user. We note that for linearly implicit problems the tolerance (2.35) is similarly used for the single Newton iteration.

2.9.4 Preconditioning

When using an inexact Newton method to solve the nonlinear system (2.21), an iterative method is used repeatedly to solve linear systems of the form $\mathcal{A}x = b$, where x is a correction vector and b is a residual vector. If this iterative method is one of the scaled preconditioned iterative linear solvers supplied with SUNDIALS, their efficiency may benefit tremendously from preconditioning. A system $\mathcal{A}x = b$ can be preconditioned using any one of:

$$\begin{aligned}
(P^{-1}\mathcal{A})x &= P^{-1}b \qquad \text{[left preconditioning]},\\
(\mathcal{A}P^{-1})Px &= b \qquad \text{[right preconditioning]},\\
(P_L^{-1}\mathcal{A}P_R^{-1})P_R x &= P_L^{-1}b \qquad \text{[left and right preconditioning]}.
\end{aligned}$$

These Krylov iterative methods are then applied to a system with the matrix $P^{-1}\mathcal{A}$, $\mathcal{A}P^{-1}$, or $P_L^{-1}\mathcal{A}P_R^{-1}$, instead of $\mathcal{A}$. In order to improve the convergence of the Krylov iteration, the preconditioner matrix P, or the product $P_L P_R$ in the third case, should in some sense approximate the system matrix $\mathcal{A}$. Simultaneously, in order to be cost-effective the matrix P (or matrices P_L and P_R) should be reasonably efficient to evaluate and solve. Finding an optimal point in this trade-off between rapid convergence and low cost can be quite challenging. Good choices are often problem-dependent (for example, see *[BH1989]* for an extensive study of preconditioners for reaction-transport systems).

Most of the iterative linear solvers supplied with SUNDIALS allow for all three types of preconditioning (left, right or both), although for non-symmetric matrices $\mathcal{A}$ we know of few situations where preconditioning on both sides is superior to preconditioning on one side only (with the product $P = P_L P_R$). Moreover, for a given preconditioner matrix, the merits of left vs. right preconditioning are unclear in general, so we recommend that the user experiment with both choices. Performance can differ between these since the inverse of the left preconditioner is included in the linear system residual whose norm is being tested in the Krylov algorithm. As a rule, however, if the preconditioner is the product of two matrices, we recommend that preconditioning be done either on the left only or the right only, rather than using one factor on each side. An exception to this rule is the PCG solver, that itself assumes a symmetric matrix $\mathcal{A}$, since the PCG algorithm in fact applies the single preconditioner matrix P in both left/right fashion as $P^{-1/2}\mathcal{A}P^{-1/2}$.

Typical preconditioners are based on approximations to the system Jacobian, $J = \partial f^I/\partial y$. Since the Newton iteration matrix involved is $\mathcal{A} = M - \gamma J$, any approximation $\bar{J}$ to J yields a matrix that is of potential use as a preconditioner, namely $P = M - \gamma\bar{J}$. Because the Krylov iteration occurs within a Newton iteration and further also within a time integration, and since each of these iterations has its own test for convergence, the preconditioner may use a very crude approximation, as long as it captures the dominant numerical features of the system. We have found that the combination of a preconditioner with the Newton-Krylov iteration, using even a relatively poor approximation to the Jacobian, can be surprisingly superior to using the same matrix without Krylov acceleration (i.e., a modified Newton iteration), as well as to using the Newton-Krylov method with no preconditioning.

2.9.5 Implicit predictors

For problems with implicit components, a prediction algorithm is employed for constructing the initial guesses for each implicit Runge-Kutta stage, $z_i^{(0)}$. As is well-known with nonlinear solvers, the selection of a good initial guess can have dramatic effects on both the speed and robustness of the solve, making the difference between rapid quadratic convergence versus divergence of the iteration. To this end, a variety of prediction algorithms are provided. In each case, the stage guesses $z_i^{(0)}$ are constructed explicitly using readily-available information, including the previous step solutions y_{n-1} and y_{n-2}, as well as any previous stage solutions $z_j, \quad j < i$. In most cases, prediction is performed by constructing an interpolating polynomial through existing data, which is then evaluated at the desired stage time to provide an inexpensive but (hopefully) reasonable prediction of the stage solution. Specifically, for most Runge-Kutta methods each stage solution satisfies

$$z_i \approx y(t_{n,i}^I),$$

(similarly for MRI methods $z_i \approx y(t_{n,i}^S)$), so by constructing an interpolating polynomial $p_q(t)$ through a set of existing data, the initial guess at stage solutions may be approximated as

$$z_i^{(0)} = p_q(t_{n,i}^I). \tag{2.36}$$

As the stage times for MRI stages and implicit ARK and DIRK stages usually have non-negative abcissae (i.e., $c_j^I > 0$), it is typically the case that $t_{n,j}^I$ (resp., $t_{n,j}^S$) is outside of the time interval containing the data used to construct $p_q(t)$, hence (2.36) will correspond to an extrapolant instead of an interpolant. The dangers of using a polynomial interpolant to extrapolate values outside the interpolation interval are well-known, with higher-order polynomials and predictions further outside the interval resulting in the greatest potential inaccuracies.

The prediction algorithms available in ARKode therefore construct a variety of interpolants $p_q(t)$, having different polynomial order and using different interpolation data, to support 'optimal' choices for different types of problems, as described below. We note that due to the structural similarities between implicit ARK and DIRK stages in ARKStep, and solve-decoupled implicit stages in MRIStep, we use the ARKStep notation throughout the remainder of this section, but each statement equally applies to MRIStep (unless otherwise noted).

2.9.5.1 Trivial predictor

The so-called "trivial predictor" is given by the formula

$$p_0(t) = y_{n-1}.$$

While this piecewise-constant interpolant is clearly not a highly accurate candidate for problems with time-varying solutions, it is often the most robust approach for highly stiff problems, or for problems with implicit constraints whose violation may cause illegal solution values (e.g. a negative density or temperature).

2.9.5.2 Maximum order predictor

At the opposite end of the spectrum, ARKode's *interpolation module* can be used to construct a higher-order polynomial interpolant, $p_q(t)$. The implicit stage predictor is computed through evaulating this interpolant at each stage time $t_{n,i}^I$.

2.9.5.3 Variable order predictor

This predictor attempts to use higher-order polynomials $p_q(t)$ for predicting earlier stages, and lower-order interpolants for later stages. It uses the same interpolation module as described above, but chooses the polynomial degree adaptively based on the stage index i, under the assumption that the stage times are increasing, i.e. $c_j^I < c_k^I$ for

$j < k$:

$$q_i = \max\{q_{\max} - i + 1,\ 1\}, \quad i = 1, \ldots, s.$$

2.9.5.4 Cutoff order predictor

This predictor follows a similar idea as the previous algorithm, but monitors the actual stage times to determine the polynomial interpolant to use for prediction. Denoting $\tau = c_i^I \dfrac{h_n}{h_{n-1}}$, the polynomial degree q_i is chosen as:

$$q_i = \begin{cases} q_{\max}, & \text{if} \quad \tau < \frac{1}{2}, \\ 1, & \text{otherwise.} \end{cases}$$

2.9.5.5 Bootstrap predictor ($M = I$ only)

This predictor does not use any information from the preceding step, instead using information only within the current step $[t_{n-1}, t_n]$. In addition to using the solution and ODE right-hand side function, y_{n-1} and $f(t_{n-1}, y_{n-1})$, this approach uses the right-hand side from a previously computed stage solution in the same step, $f(t_{n-1} + c_j^I h, z_j)$ to construct a quadratic Hermite interpolant for the prediction. If we define the constants $\tilde{h} = c_j^I h$ and $\tau = c_i^I h$, the predictor is given by

$$z_i^{(0)} = y_{n-1} + \left(\tau - \frac{\tau^2}{2\tilde{h}}\right) f(t_{n-1}, y_{n-1}) + \frac{\tau^2}{2\tilde{h}} f(t_{n-1} + \tilde{h}, z_j).$$

For stages without a nonzero preceding stage time, i.e. $c_j^I \neq 0$ for $j < i$, this method reduces to using the trivial predictor $z_i^{(0)} = y_{n-1}$. For stages having multiple preceding nonzero c_j^I, we choose the stage having largest c_j^I value, to minimize the level of extrapolation used in the prediction.

We note that in general, each stage solution z_j has significantly worse accuracy than the time step solutions y_{n-1}, due to the difference between the *stage order* and the *method order* in Runge-Kutta methods. As a result, the accuracy of this predictor will generally be rather limited, but it is provided for problems in which this increased stage error is better than the effects of extrapolation far outside of the previous time step interval $[t_{n-2}, t_{n-1}]$.

Although this approach could be used with non-identity mass matrix, support for that mode is not currently implemented, so selection of this predictor in the case of a non-identity mass matrix will result in use of the trivial predictor.

2.9.5.6 Minimum correction predictor (ARKStep, $M = I$ only)

The final predictor is not interpolation based; instead it utilizes all existing stage information from the current step to create a predictor containing all but the current stage solution. Specifically, as discussed in equations (2.4) and (2.21), each stage solves a nonlinear equation

$$z_i = y_{n-1} + h_n \sum_{j=1}^{i-1} A_{i,j}^E f^E(t_{n,j}^E, z_j) + h_n \sum_{j=1}^{i} A_{i,j}^I f^I(t_{n,j}^I, z_j),$$

$$\Leftrightarrow$$

$$G(z_i) \equiv z_i - h_n A_{i,i}^I f^I(t_{n,i}^I, z_i) - a_i = 0.$$

This prediction method merely computes the predictor z_i as

$$z_i = y_{n-1} + h_n \sum_{j=1}^{i-1} A^E_{i,j} f^E(t^E_{n,j}, z_j) + h_n \sum_{j=1}^{i-1} A^I_{i,j} f^I(t^I_{n,j}, z_j),$$

$$\Leftrightarrow$$

$$z_i = a_i.$$

Again, although this approach could be used with non-identity mass matrix, support for that mode is not currently implemented, so selection of this predictor in the case of a non-identity mass matrix will result in use of the trivial predictor.

2.9.6 Mass matrix solver (ARKStep only)

Within the ARKStep algorithms described above, there are multiple locations where a matrix-vector product

$$b = Mv \tag{2.37}$$

or a linear solve

$$x = M^{-1}b \tag{2.38}$$

is required.

Of course, for problems in which $M = I$ both of these operators are trivial. However for problems with non-identity mass matrix, these linear solves (2.38) may be handled using any valid linear solver module, in the same manner as described in the section *Linear solver methods* for solving the linear Newton systems.

For ERK methods involving non-identity mass matrix, even though calculation of individual stages does not require an algebraic solve, both of the above operations (matrix-vector product, and mass matrix solve) may be required within each time step. Therefore, for these users we recommend reading the rest of this section as it pertains to ARK methods, with the obvious simplification that since $f^E = f$ and $f^I = 0$ no Newton or fixed-point nonlinear solve, and no overall system linear solve, is involved in the solution process.

At present, for DIRK and ARK problems using a matrix-based solver for the Newton nonlinear iterations, the type of matrix (dense, band, sparse, or custom) for the Jacobian matrix J must match the type of mass matrix M, since these are combined to form the Newton system matrix $\tilde{A}$. When matrix-based methods are employed, the user must supply a routine to compute $M(t)$ in the appropriate form to match the structure of $\mathcal{A}$, with a user-supplied routine of type `ARKLsMassFn()`. This matrix structure is used internally to perform any requisite mass matrix-vector products (2.37).

When matrix-free methods are selected, a routine must be supplied to perform the mass-matrix-vector product, Mv. As with iterative solvers for the Newton systems, preconditioning may be applied to aid in solution of the mass matrix systems (2.38). When using an iterative mass matrix linear solver, we require that the norm of the preconditioned linear residual satisfies

$$\|r\| \leq \epsilon_L \epsilon, \tag{2.39}$$

where again, ϵ is the nonlinear solver tolerance parameter from (2.34). When using iterative system and mass matrix linear solvers, ϵ_L may be specified separately for both tolerances (2.35) and (2.39).

In the above algorithmic description there are five locations where a linear solve of the form (2.38) is required: (a) at each iteration of a fixed-point nonlinear solve, (b) in computing the Runge–Kutta right-hand side vectors $\hat{f}^E_i$ and

$\hat{f}_i^I$, (c) in constructing the time-evolved solution y_n, (d) in estimating the local temporal truncation error, and (e) in constructing predictors for the implicit solver iteration (see section *Maximum order predictor*). We note that different nonlinear solver approaches (i.e., Newton vs fixed-point) and different types of mass matrices (i.e., time-dependent versus fixed) result in different subsets of the above operations. We discuss each of these in the bullets below.

- When using a fixed-point nonlinear solver, at each fixed-point iteration we must solve

$$M(t_{n,i}^I)\, z_i^{(m+1)} = G\left(z_i^{(m)}\right), \quad m = 0, 1, \ldots$$

  for the new fixed-point iterate, $z_i^{(m+1)}$.

- In the case of a time-dependent mass matrix, to construct the Runge–Kutta right-hand side vectors we must solve

$$M(t_{n,i}^E)\hat{f}_i^E \;=\; f^E(t_{n,i}^E, z_i) \quad \text{and} \quad M(t_{n,i}^I)\hat{f}_j^I \;=\; f^I(t_{n,i}^I, z_i)$$

  for the vectors $\hat{f}_i^E$ and $\hat{f}_i^I$.

- For fixed mass matrices, we construct the time-evolved solution y_n from equation (2.4) by solving

$$\begin{aligned} My_n \;&=\; My_{n-1} + h_n \sum_{i=1}^{s} \left(b_i^E f^E(t_{n,i}^E, z_i) + b_i^I f^I(t_{n,i}^I, z_i)\right), \\ \Leftrightarrow \\ M(y_n - y_{n-1}) \;&=\; h_n \sum_{i=1}^{s} \left(b_i^E f^E(t_{n,i}^E, z_i) + b_i^I f^I(t_{n,i}^I, z_i)\right), \\ \Leftrightarrow \\ M\nu \;&=\; h_n \sum_{i=1}^{s} \left(b_i^E f^E(t_{n,i}^E, z_i) + b_i^I f^I(t_{n,i}^I, z_i)\right), \end{aligned}$$

  for the update $\nu = y_n - y_{n-1}$.

  Similarly, we compute the local temporal error estimate T_n from equation (2.18) by solving systems of the form

$$M\, T_n = h \sum_{i=1}^{s} \left[\left(b_i^E - \tilde{b}_i^E\right) f^E(t_{n,i}^E, z_i) + \left(b_i^I - \tilde{b}_i^I\right) f^I(t_{n,i}^I, z_i)\right]. \tag{2.40}$$

- For problems with either form of non-identity mass matrix, in constructing dense output and implicit predictors of order 2 or higher (see the section *Maximum order predictor* above), we compute the derivative information $\hat{f}_k$ from the equation

$$M(t_n)\hat{f}_n = f^E(t_n, y_n) + f^I(t_n, y_n).$$

In total, for problems with time-independent mass matrix, we require only two mass-matrix linear solves (2.38) per attempted time step, with one more upon completion of a time step that meets the solution accuracy requirements. When fixed time-stepping is used ($h_n = h$), the solve (2.40) is not performed at each attempted step.

Similarly, for problems with time-dependent mass matrix, we require $2s$ mass-matrix linear solves (2.38) per attempted step, where s is the number of stages in the ARK method (only half of these are required for purely explicit or purely implicit problems, (2.5) or (2.6)), with one more upon completion of a time step that meets the solution accuracy requirements.

In addition to the above totals, when using a fixed-point nonlinear solver (assumed to require m iterations), we will need an additional ms mass-matrix linear solves (2.38) per attempted time step (but zero linear solves with the system Jacobian).

2.10 Rootfinding

All of the time-stepping modules in ARKode also support a rootfinding feature. This means that, while integrating the IVP (2.1), these can also find the roots of a set of user-defined functions $g_i(t, y)$ that depend on t and the solution vector $y = y(t)$. The number of these root functions is arbitrary, and if more than one g_i is found to have a root in any given interval, the various root locations are found and reported in the order that they occur on the t axis, in the direction of integration.

Generally, this rootfinding feature finds only roots of odd multiplicity, corresponding to changes in sign of $g_i(t, y(t))$, denoted $g_i(t)$ for short. If a user root function has a root of even multiplicity (no sign change), it will almost certainly be missed due to the realities of floating-point arithmetic. If such a root is desired, the user should reformulate the root function so that it changes sign at the desired root.

The basic scheme used is to check for sign changes of any $g_i(t)$ over each time step taken, and then (when a sign change is found) to home in on the root (or roots) with a modified secant method *[HS1980]*. In addition, each time g is evaluated, ARKode checks to see if $g_i(t) = 0$ exactly, and if so it reports this as a root. However, if an exact zero of any g_i is found at a point t, ARKode computes $g(t + \delta)$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$ also, ARKode stops and reports an error. This way, each time ARKode takes a time step, it is guaranteed that the values of all g_i are nonzero at some past value of t, beyond which a search for roots is to be done.

At any given time in the course of the time-stepping, after suitable checking and adjusting has been done, ARKode has an interval $(t_{\text{lo}}, t_{\text{hi}}]$ in which roots of the $g_i(t)$ are to be sought, such that t_{hi} is further ahead in the direction of integration, and all $g_i(t_{\text{lo}}) \neq 0$. The endpoint t_{hi} is either t_n, the end of the time step last taken, or the next requested output time t_{out} if this comes sooner. The endpoint t_{lo} is either t_{n-1}, or the last output time t_{out} (if this occurred within the last step), or the last root location (if a root was just located within this step), possibly adjusted slightly toward t_n if an exact zero was found. The algorithm checks $g(t_{\text{hi}})$ for zeros, and it checks for sign changes in $(t_{\text{lo}}, t_{\text{hi}})$. If no sign changes are found, then either a root is reported (if some $g_i(t_{\text{hi}}) = 0$) or we proceed to the next time interval (starting at t_{hi}). If one or more sign changes were found, then a loop is entered to locate the root to within a rather tight tolerance, given by

$$\tau = 100\, U\, (|t_n| + |h|) \qquad (\text{where } U = \text{unit roundoff}).$$

Whenever sign changes are seen in two or more root functions, the one deemed most likely to have its root occur first is the one with the largest value of $|g_i(t_{\text{hi}})| \,/\, |g_i(t_{\text{hi}}) - g_i(t_{\text{lo}})|$, corresponding to the closest to t_{lo} of the secant method values. At each pass through the loop, a new value t_{mid} is set, strictly within the search interval, and the values of $g_i(t_{\text{mid}})$ are checked. Then either t_{lo} or t_{hi} is reset to t_{mid} according to which subinterval is found to have the sign change. If there is none in $(t_{\text{lo}}, t_{\text{mid}})$ but some $g_i(t_{\text{mid}}) = 0$, then that root is reported. The loop continues until $|t_{\text{hi}} - t_{\text{lo}}| < \tau$, and then the reported root location is t_{hi}. In the loop to locate the root of $g_i(t)$, the formula for t_{mid} is

$$t_{\text{mid}} = t_{\text{hi}} - \frac{g_i(t_{\text{hi}})(t_{\text{hi}} - t_{\text{lo}})}{g_i(t_{\text{hi}}) - \alpha g_i(t_{\text{lo}})},$$

where α is a weight parameter. On the first two passes through the loop, α is set to 1, making t_{mid} the secant method value. Thereafter, α is reset according to the side of the subinterval (low vs high, i.e. toward t_{lo} vs toward t_{hi}) in which the sign change was found in the previous two passes. If the two sides were opposite, α is set to 1. If the two sides were the same, α is halved (if on the low side) or doubled (if on the high side). The value of t_{mid} is closer to t_{lo} when $\alpha < 1$ and closer to t_{hi} when $\alpha > 1$. If the above value of t_{mid} is within $\tau/2$ of t_{lo} or t_{hi}, it is adjusted inward, such that its fractional distance from the endpoint (relative to the interval size) is between 0.1 and 0.5 (with 0.5 being the midpoint), and the actual distance from the endpoint is at least $\tau/2$.

Finally, we note that when running in parallel, ARKode's rootfinding module assumes that the entire set of root defining functions $g_i(t, y)$ is replicated on every MPI task. Since in these cases the vector y is distributed across tasks, it is the user's responsibility to perform any necessary inter-task communication to ensure that $g_i(t, y)$ is identical on each task.

2.11 Inequality Constraints

The ARKStep and ERKStep modules in ARKode permit the user to impose optional inequality constraints on individual components of the solution vector y. Any of the following four constraints can be imposed: $y_i > 0$, $y_i < 0$, $y_i \geq 0$, or $y_i \leq 0$. The constraint satisfaction is tested after a successful step and before the error test. If any constraint fails, the step size is reduced and a flag is set to update the Jacobian or preconditioner if applicable. Rather than cutting the step size by some arbitrary factor, ARKode estimates a new step size h' using a linear approximation of the components in y that failed the constraint test (including a safety factor of 0.9 to cover the strict inequality case). If a step fails to satisfy the constraints 10 times (a value which may be modified by the user) within a step attempt or fails with the minimum step size then the integration is halted and an error is returned. In this case the user may need to employ other strategies as discussed in *ARKStep tolerance specification functions* and *ERKStep tolerance specification functions* to satisfy the inequality constraints.

Chapter 3

Code Organization

The family of solvers referred to as SUNDIALS consists of the solvers CVODE and ARKode (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, SUNDIALS also includes variants of CVODE and IDA with sensitivity analysis capabilities (using either forward or adjoint methods), called CVODES and IDAS, respectively.

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see the following Figures *SUNDIALS organization* and *SUNDIALS tree*). The following is a list of the solver packages presently available, and the basic functionality of each:

- CVODE, a linear multistep solver for stiff and nonstiff ODE systems $\dot{y} = f(t, y)$ based on Adams and BDF methods;
- CVODES, a linear multistep solver for stiff and nonstiff ODEs with sensitivity analysis capabilities;
- ARKode, a Runge-Kutta based solver for stiff, nonstiff, mixed stiff-nonstiff, and multirate ODE systems;
- IDA, a linear multistep solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$ based on BDF methods;
- IDAS, a linear multistep solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

Note for modules that provide interfaces to third-party libraries (i.e., LAPACK, KLU, SuperLU_MT, SuperLU_DIST, *hypre*, PETSc, Trilinos, and RAJA users will need to download and compile those packages independently.

3.1 ARKode organization

The ARKode package is written in the ANSI C language. The following summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

The overall organization of the ARKode package is shown in Figure *ARKode organization*. The central integration modules, implemented in the files `arkode.h`, `arkode_impl.h`, `arkode_butcher.h`, `arkode.c`, `arkode_arkstep.c` , `arkode_erkstep.c`, `arkode_mristep.h`, and `arkode_butcher.c`, deal with the evaluation of integration stages, the nonlinear solvers, estimation of the local truncation error, selection of step size, and interpolation to user output points, among other issues. ARKode currently supports modified Newton, inexact Newton, and accelerated fixed-point solvers for these nonlinearly implicit problems. However, when using the Newton-based iterations, or when using a non-identity mass matrix $M \neq I$, ARKode has flexibility in the choice of method used to solve the linear sub-systems that arise. Therefore, for any user problem invoking the Newton solvers,

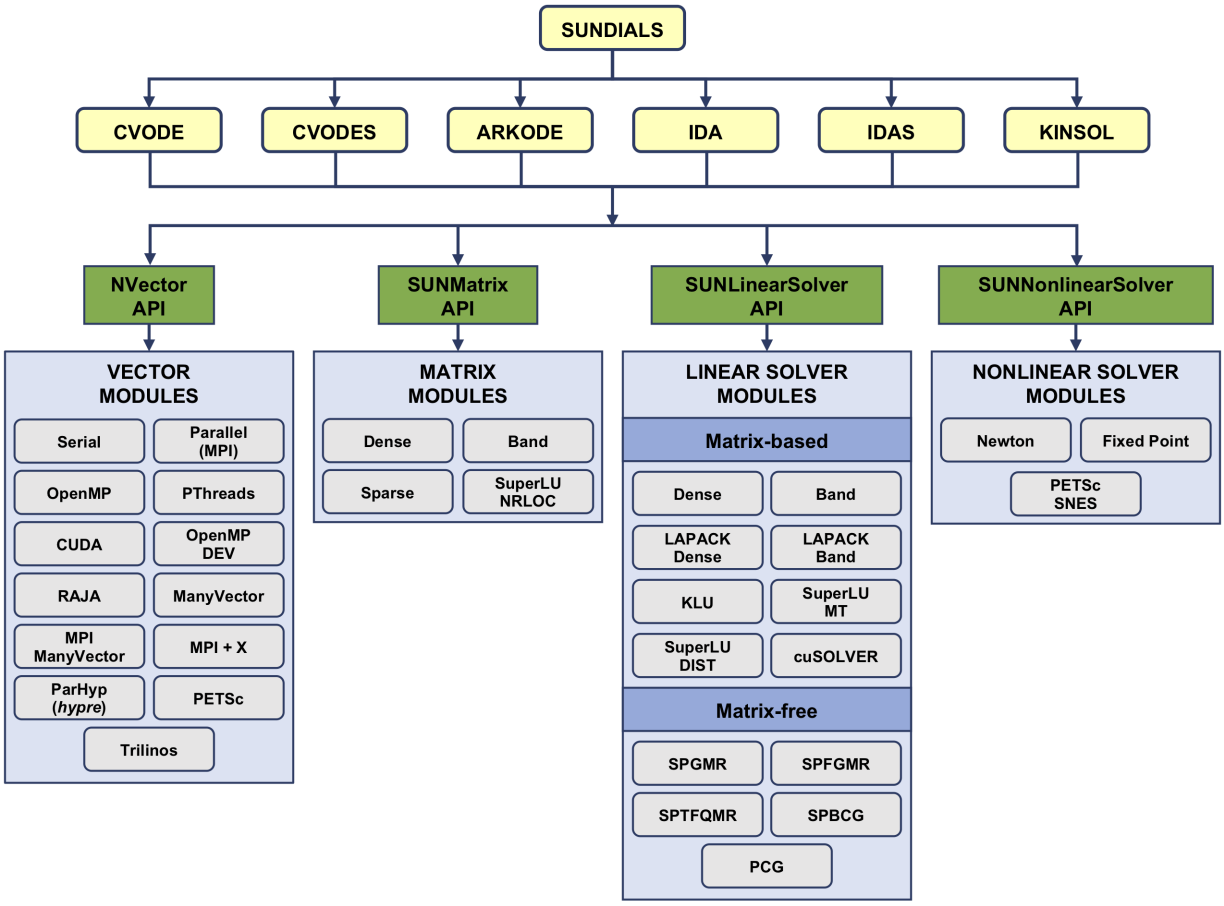


Fig. 3.1: *SUNDIALS organization*: High-level diagram of the SUNDIALS structure

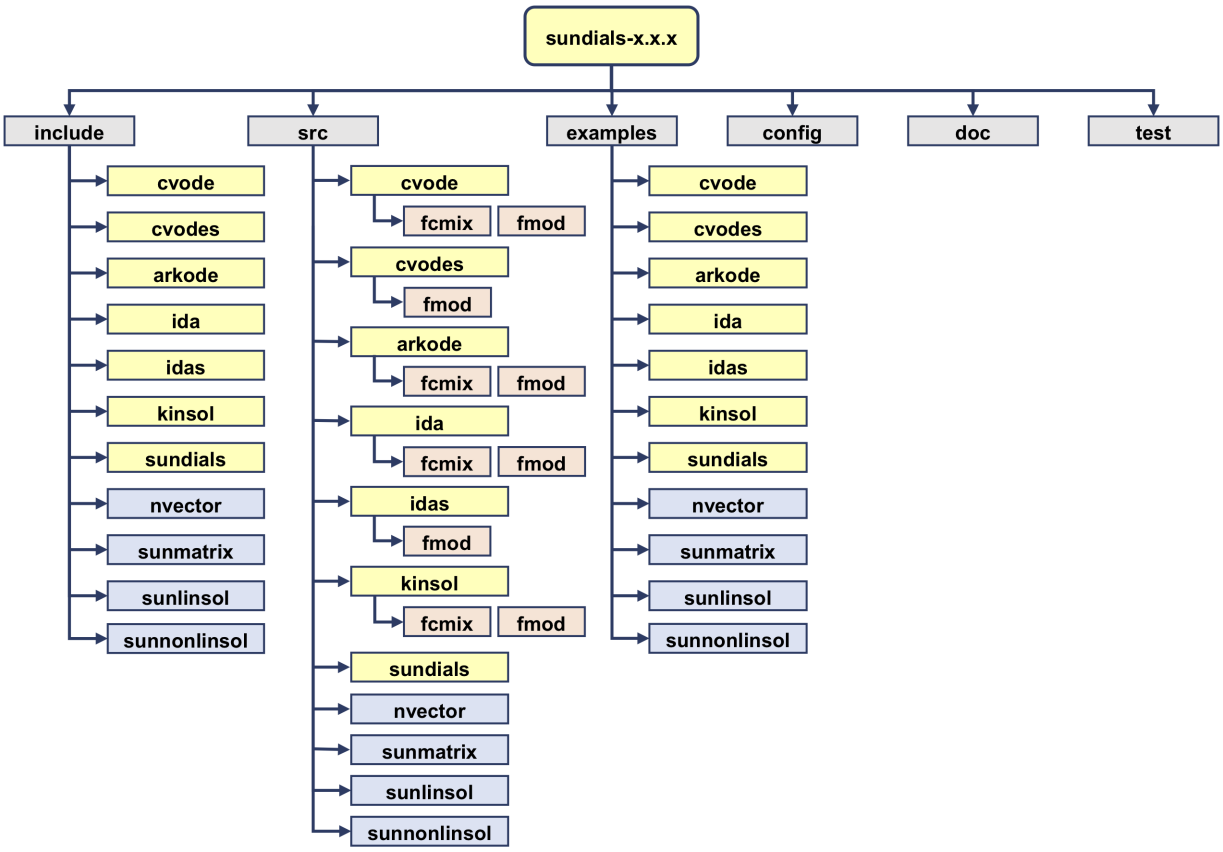


Fig. 3.2: *SUNDIALS tree*: Directory structure of the source tree.

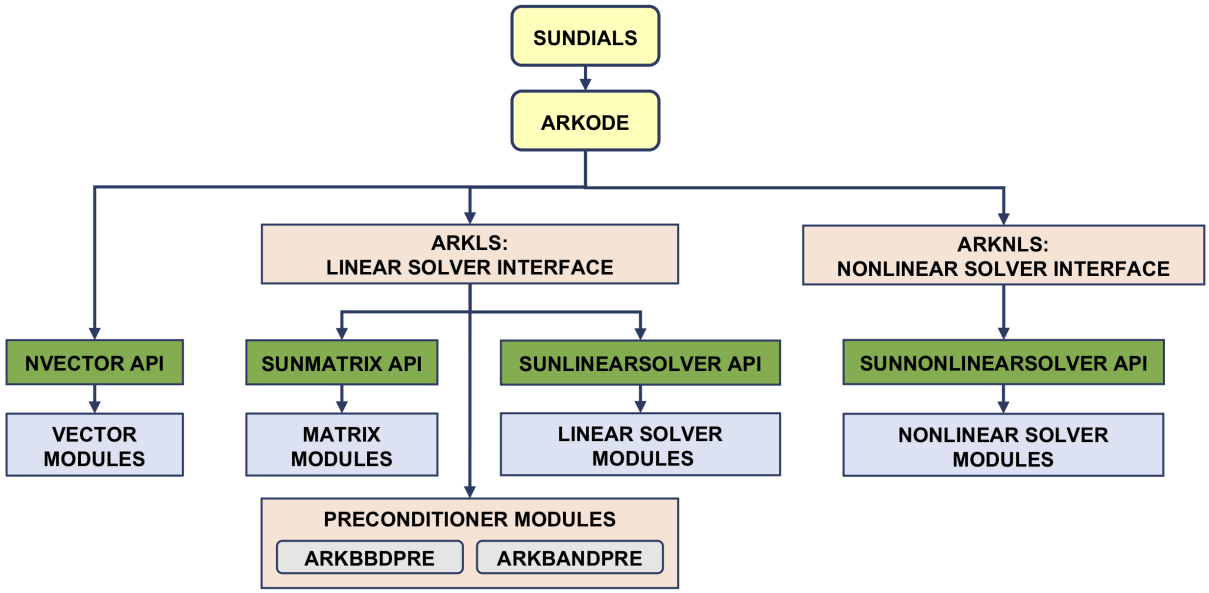


Fig. 3.3: *ARKode organization*: Overall structure of the ARKode package. Modules specific to ARKode are the timesteppers (ARKODE), linear solver interfaces (ARKLS), nonlinear solver interfaces (ARKNLS), and preconditioners (ARKBANDPRE and ARKBBDPRE); all other items correspond to generic SUNDIALS vector, matrix, and solver modules.

or any user problem with $M \neq I$, one (or more) of the linear system solver modules should be specified by the user, which is then invoked as needed during the integration process.

For solving these linear systems, ARKode's linear solver interface supports both direct and iterative linear solvers built using the generic SUNLINSOL API (see *Description of the SUNLinearSolver module*). These solvers may utilize a SUNMATRIX object for storing Jacobian information, or they may be matrix-free. Since ARKode can operate on any valid SUNLINSOL implementation, the set of linear solver modules available to ARKode will expand as new SUNLINSOL modules are developed.

For users employing dense or banded Jacobians, ARKode includes algorithms for their approximation through difference quotients, although the user also has the option of supplying a routine to compute the Jacobian (or an approximation to it) directly. This user-supplied routine is required when using sparse or user-supplied Jacobian matrices.

For users employing iterative linear solvers, ARKode includes an algorithm for the approximation by difference quotients of the product Av. Again, the user has the option of providing routines for this operation, in two phases: setup (preprocessing of Jacobian data) and multiplication.

When solve problems with non-identity mass matrices, corresponding user-supplied routines for computing either the mass matrix M or the product Mv are required. Additionally, the type of linear solver module (iterative, dense-direct, band-direct, sparse-direct) used for both the IVP system and mass matrix must match.

For preconditioned iterative methods for either the system or mass matrix solves, the preconditioning must be supplied by the user, again in two phases: setup and solve. While there is no default choice of preconditioner analogous to the difference-quotient approximation in the direct case, the references *[BH1989]* and *[B1992]*, together with the example and demonstration programs included with ARKode and CVODE, offer considerable assistance in building simple preconditioners.

ARKode's linear solver interface consists of four primary phases, devoted to

1. memory allocation and initialization,
2. setup of the matrix/preconditioner data involved,
3. solution of the system, and
4. freeing of memory.

The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the integration process, and only as required to achieve convergence.

ARKode also provides two rudimentary preconditioner modules, for use with any of the Krylov iterative linear solvers. The first, ARKBANDPRE is intended to be used with the serial or threaded vector data structures (NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS), and provides a banded difference-quotient approximation to the Jacobian as the preconditioner, with corresponding setup and solve routines. The second preconditioner module, ARKBBDPRE, is intended to work with the parallel vector data structure, NVECTOR_PARALLEL, and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix owned by a single processor.

All state information used by ARKode to solve a given problem is saved in a single opaque memory structure, and a pointer to that structure is returned to the user. For C and C++ applications there is no global data in the ARKode package, and so in this respect it is reentrant. State information specific to the linear solver interface is saved in a separate data structure, a pointer to which resides in the ARKode memory structure. State information specific to the linear solver implementation (and matrix implementation, if applicable) are stored in their own data structures, that are returned to the user upon construction, and subsequently provided to ARKode for use. We note that the ARKode Fortran interface, however, currently uses global variables, so at most one of each of these objects may be created per memory space (i.e. one per MPI task in distributed memory computations).

Chapter 4

Using ARKStep for C and C++ Applications

This chapter is concerned with the use of the ARKStep time-stepping module for the solution of initial value problems (IVPs) in a C or C++ language setting. The following sections discuss the header files and the layout of the user's main program, and provide descriptions of the ARKStep user-callable functions and user-supplied functions.

The example programs described in the companion document *[R2018]* may be helpful. Those codes may be used as templates for new codes and are included in the ARKode package `examples` subdirectory.

Users with applications written in Fortran should see the chapter *FARKODE, an Interface Module for FORTRAN Applications*, which describes the Fortran/C interface module for ARKStep, and may look to the Fortran example programs also described in the companion document *[R2018]*. These codes are also located in the ARKode package `examples` directory.

The user should be aware that not all SUNLINSOL, SUNMATRIX, and preconditioning modules are compatible with all NVECTOR implementations. Details on compatibility are given in the documentation for each SUNMATRIX (see *Matrix Data Structures*) and each SUNLINSOL module (see *Description of the SUNLinearSolver module*). For example, NVECTOR_PARALLEL is not compatible with the dense, banded, or sparse SUNMATRIX types, or with the corresponding dense, banded, or sparse SUNLINSOL modules. Please check the sections *Matrix Data Structures* and *Description of the SUNLinearSolver module* to verify compatibility between these modules. In addition to that documentation, we note that the ARKBANDPRE preconditioning module is only compatible with the NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS vector implementations, and the preconditioner module ARKBBDPRE can only be used with NVECTOR_PARALLEL.

ARKStep uses various input and output constants from the shared ARKode infrastructure. These are defined as needed in this chapter, but for convenience the full list is provided separately in the section *Appendix: ARKode Constants*.

The relevant information on using ARKStep's C and C++ interfaces is detailed in the following sub-sections.

4.1 Access to library and header files

At this point, it is assumed that the installation of ARKode, following the procedure described in the section *ARKode Installation Procedure*, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the appropriate locations for the library and header files required by ARKode. The relevant library files are

- `libdir/libsundials_arkode.lib`,
- `libdir/libsundials_nvec*.lib`,

where the file extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries. The relevant header files are located in the subdirectories

- `incdir/include/arkode`
- `incdir/include/sundials`
- `incdir/include/nvector`
- `incdir/include/sunmatrix`
- `incdir/include/sunlinsol`
- `incdir/include/sunnonlinsol`

The directories `libdir` and `incdir` are the installation library and include directories, respectively. For a default installation, these are `instdir/lib` and `instdir/include`, respectively, where `instdir` is the directory where SUNDIALS was installed (see the section *ARKode Installation Procedure* for further details).

4.2 Data Types

The `sundials_types.h` file contains the definition of the variable type `realtype`, which is used by the SUNDIALS solvers for all floating-point data, the definition of the integer type `sunindextype`, which is used for vector and matrix indices, and `booleantype`, which is used for certain logic operations within SUNDIALS.

4.2.1 Floating point types

The type "realtype" can be set to `float`, `double`, or `long double`, depending on how SUNDIALS was installed (with the default being `double`). The user can change the precision of the SUNDIALS solvers' floating-point arithmetic at the configuration stage (see the section *Configuration options (Unix/Linux)*).

Additionally, based on the current precision, `sundials_types.h` defines the values BIG_REAL to be the largest value representable as a `realtype`, SMALL_REAL to be the smallest positive value representable as a `realtype`, and UNIT_ROUNDOFF to be the smallest realtype number, ε, such that $1.0 + \varepsilon \neq 1.0$.

Within SUNDIALS, real constants may be set to have the appropriate precision by way of a macro called RCONST. It is this macro that needs the ability to branch on the definition `realtype`. In ANSI C, a floating-point constant with no suffix is stored as a `double`. Placing the suffix "F" at the end of a floating point constant makes it a `float`, whereas using the suffix "L" makes it a `long double`. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines `A` to be a `double` constant equal to 1.0, `B` to be a `float` constant equal to 1.0, and `C` to be a `long double` constant equal to 1.0. The macro call `RCONST(1.0)` automatically expands to 1.0 if `realtype` is `double`, to `1.0F` if `realtype` is `float`, or to `1.0L` if `realtype` is `long double`. SUNDIALS uses the `RCONST` macro internally to declare all of its floating-point constants.

Additionally, SUNDIALS defines several macros for common mathematical functions *e.g.*}, `fabs`, `sqrt`, `exp`, etc. in `sundials_math.h`. The macros are prefixed with `SUNR` and expand to the appropriate C function based on the `realtype`. For example, the macro `SUNRabs` expands to the C function `fabs` when `realtype` is `double`, `fabsf` when `realtype` is `float`, and `fabsl` when `realtype` is `long double`.

A user program which uses the type `realtype`, the `RCONST` macro, and the `SUNR` mathematical function macros is precision-independent except for any calls to precision-specific library functions. Our example programs use `realtype`, `RCONST`, and the `SUNR` macros. Users can, however, use the type `double`, `float`, or `long`

double in their code (assuming that this usage is consistent with the typedef for realtype) and call the appropriate math library functions directly. Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use realtype, RCONST, or the SUNR macros so long as the SUNDIALS libraries use the correct precision (for details see *ARKode Installation Procedure*).

4.2.2 Integer types used for vector and matrix indices

The type sunindextype can be either a 32- or 64-bit *signed* integer. The default is the portable int64_t type, and the user can change it to int32_t at the configuration stage. The configuration system will detect if the compiler does not support portable types, and will replace int32_t and int64_t with int, long int, or long long int as appropriate, to ensure use of the desired sizes on Linux, Mac OS X, and Windows platforms. SUNDIALS currently does not support *unsigned* integer types for vector and matrix indices, although these could be added in the future if there is sufficient demand.

A user program which uses sunindextype to handle vector and matrix indices will work with both index storage types except for any calls to index storage-specific external libraries. (Our C and C++ example programs use sunindextype.) Users can, however, use any one of int, long int, int32_t, int64_t or long long int in their code, assuming that this usage is consistent with the typedef for sunindextype on their architecture. Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use sunindextype, so long as the SUNDIALS libraries use the appropriate index storage type (for details see the section *ARKode Installation Procedure*).

4.3 Header Files

When using ARKStep, the calling program must include several header files so that various macros and data types can be used. The header file that is always required is:

- arkode/arkode_arkstep.h, the main header file for the ARKStep time-stepping module, which defines the several types and various constants, includes function prototypes, and includes the shared arkode/arkode.h and arkode/arkode_ls.h header files.

Note that arkode.h includes sundials_types.h directly, which defines the types realtype, sunindextype and booleantype and the constants SUNFALSE and SUNTRUE, so a user program does not need to include sundials_types.h directly.

Additionally, the calling program must also include a NVECTOR implementation header file, of the form nvector/nvector_***.h, corresponding to the user's preferred data layout and form of parallelism. See the section *Vector Data Structures* for details for the appropriate name. This file in turn includes the header file sundials_nvector.h which defines the abstract N_Vector data type.

If the user wishes to manually select between any of the pre-defined ERK or DIRK Butcher tables, these are defined through a set of constants that are enumerated in the header files arkode/arkode_butcher_erk.h and arkode/arkode_butcher_dirk.h, or if a user wishes to manually specify one or more Butcher tables, the corresponding ARKodeButcherTable structure is defined in arkode/arkode_butcher.h.

If the user includes a non-trivial implicit component to their ODE system, then each implicit stage will require a nonlinear solver for the resulting system of algebraic equations – the default for this is a modified or inexact Newton iteration, depending on the user's choice of linear solver. If using a non-default nonlinear solver module, or when interacting with a SUNNONLINSOL module directly, the calling program must also include a SUNNONLINSOL header file, of the form sunnonlinsol/sunnonlinsol_***.h where *** is the name of the nonlinear solver module (see the section *Description of the SUNNonlinearSolver Module* for more information). This file in turn includes the header file sundials_nonlinearsolver.h which defines the abstract SUNNonlinearSolver data type.

If using a nonlinear solver that requires the solution of a linear system of the form $\mathcal{A}x = b$ (e.g., the default Newton iteration), then a linear solver module header file will also be required. Similarly, if the ODE system involves a non-identity mass matrix $M \neq I$, then each time step will require a linear solver for systems of the form $Mx = b$. The header files corresponding to the SUNDIALS-provided linear solver modules available for use with ARKode are:

- Direct linear solvers:
  - `sunlinsol/sunlinsol_dense.h`, which is used with the dense linear solver module, SUNLINSOL_DENSE;
  - `sunlinsol/sunlinsol_band.h`, which is used with the banded linear solver module, SUNLINSOL_BAND;
  - `sunlinsol/sunlinsol_lapackdense.h`, which is used with the LAPACK dense linear solver module, SUNLINSOL_LAPACKDENSE;
  - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK banded linear solver module, SUNLINSOL_LAPACKBAND;
  - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver module, SUNLINSOL_KLU;
  - `sunlinsol/sunlinsol_superlumt.h`, which is used with the SuperLU_MT sparse linear solver module, SUNLINSOL_SUPERLUMT;
  - `sunlinsol/sunlinsol_superludist.h`, which is used with the SuperLU_DIST parallel sparse linear solver module, SUNLINSOL_SUPERLUDIST;
  - `sunlinsol/sunlinsol_cusolversp_batchqr.h`, which is used with the batched sparse QR factorization method provided by the NVDIA cuSOLVER library, SUNLINSOL_CUSOLVERSP_BATCHQR;
- Iterative linear solvers:
  - `sunlinsol/sunlinsol_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver module, SUNLINSOL_SPGMR;
  - `sunlinsol/sunlinsol_spfgmr.h`, which is used with the scaled, preconditioned FGMRES Krylov linear solver module, SUNLINSOL_SPFGMR;
  - `sunlinsol/sunlinsol_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, SUNLINSOL_SPBCGS;
  - `sunlinsol/sunlinsol_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov linear solver module, SUNLINSOL_SPTFQMR;
  - `sunlinsol/sunlinsol_pcg.h`, which is used with the scaled, preconditioned CG Krylov linear solver module, SUNLINSOL_PCG;

The header files for the SUNLINSOL_DENSE and SUNLINSOL_LAPACKDENSE linear solver modules include the file `sunmatrix/sunmatrix_dense.h`, which defines the SUNMATRIX_DENSE matrix module, as well as various functions and macros for acting on such matrices.

The header files for the SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND linear solver modules include the file `sunmatrix/sunmatrix_band.h`, which defines the SUNMATRIX_BAND matrix module, as well as various functions and macros for acting on such matrices.

The header files for the SUNLINSOL_KLU and SUNLINSOL_SUPERLUMT linear solver modules include the file `sunmatrix/sunmatrix_sparse.h`, which defines the SUNMATRIX_SPARSE matrix module, as well as various functions and macros for acting on such matrices.

The header file for the SUNLINSOL_CUSOLVERSP_BATCHQR linear solver module includes the file `sunmatrix/sunmatrix_cusparse.h`, which defines the SUNMATRIX_CUSPARSE matrix module, as well as various functions for acting on such matrices.

The header file for the SUNLINSOL_SUPERLUDIST linear solver module includes the file `sunmatrix/sunmatrix_slunrloc.h`, which defines the SUNMATRIX_SLUNRLOC matrix module, as well as various functions for acting on such matrices.

The header files for the Krylov iterative solvers include the file `sundials/sundials_iterative.h`, which enumerates the preconditioning type and (for the SPGMR and SPFGMR solvers) the choices for the Gram-Schmidt orthogonalization process.

Other headers may be needed, according to the choice of preconditioner, etc. For example, if preconditioning for an iterative linear solver were performed using the ARKBBDPRE module, the header `arkode/arkode_bbdpre.h` is needed to access the preconditioner initialization routines.

4.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP using the ARKStep module. Most of the steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL and SUNNONLINSOL implementations used. For the steps that are not, refer to the sections *Vector Data Structures*, *Matrix Data Structures*, *Description of the SUNLinearSolver module*, and *Description of the SUNNonlinearSolver Module* for the specific name of the function to be called or macro to be referenced.

1. Initialize parallel or multi-threaded environment, if appropriate.

   For example, call `MPI_Init` to initialize MPI if used, or set `num_threads`, the number of threads to use within the threaded vector functions, if used.

2. Set problem dimensions, etc.

   This generally includes the problem size, `N`, and may include the local vector length `Nlocal`.

   Note: The variables `N` and `Nlocal` should be of type `sunindextype`.

   ---

3. Set vector of initial values

   To set the vector `y0` of initial values, use the appropriate functions defined by the particular NVECTOR implementation.

   For native SUNDIALS vector implementations (except the CUDA and RAJA based ones), use a call of the form

   ```
   y0 = N_VMake_***(..., ydata);
   ```

   if the `realtype` array `ydata` containing the initial values of y already exists. Otherwise, create a new vector by making a call of the form

   ```
   y0 = N_VNew_***(...);
   ```

   and then set its elements by accessing the underlying data where it is located with a call of the form

   ```
   ydata = N_VGetArrayPointer_***(y0);
   ```

   See the sections *The NVECTOR_SERIAL Module* through *The NVECTOR_PTHREADS Module* for details.

For the HYPRE and PETSc vector wrappers, first create and initialize the underlying vector, and then create the NVECTOR wrapper with a call of the form

```
y0 = N_VMake_***(yvec);
```

where `yvec` is a HYPRE or PETSc vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer_***(...)` are not available for these vector wrappers. See the sections *The NVECTOR_PARHYP Module* and *The NVECTOR_PETSC Module* for details.

If using either the CUDA- or RAJA-based vector implementations use calls to the module-specific routines

```
y0 = N_VMake_***(...);
```

as applicable. See the sections *The NVECTOR_CUDA Module* and *The NVECTOR_RAJA Module* for details.

4. Create ARKStep object

   Call `arkode_mem = ARKStepCreate(...)` to create the ARKStep memory block. *ARKStepCreate()* returns a `void*` pointer to this memory structure. See the section *ARKStep initialization and deallocation functions* for details.

5. Specify integration tolerances

   Call *ARKStepSStolerances()* or *ARKStepSVtolerances()* to specify either a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. Alternatively, call *ARKStepWFtolerances()* to specify a function which sets directly the weights used in evaluating WRMS vector norms. See the section *ARKStep tolerance specification functions* for details.

   If a problem with non-identity mass matrix is used, and the solution units differ considerably from the equation units, absolute tolerances for the equation residuals (nonlinear and linear) may be specified separately through calls to *ARKStepResStolerance()*, *ARKStepResVtolerance()*, or *ARKStepResFtolerance()*.

6. Create matrix object

   If a nonlinear solver requiring a linear solver will be used (e.g., a Newton iteration) and the linear solver will be a matrix-based linear solver, then a template Jacobian matrix must be created by using the appropriate functions defined by the particular SUNMATRIX implementation.

   For the SUNDIALS-supplied SUNMATRIX implementations, the matrix object may be created using a call of the form

   ```
   SUNMatrix A = SUNBandMatrix(...);
   ```

   or

   ```
   SUNMatrix A = SUNDenseMatrix(...);
   ```

   or

   ```
   SUNMatrix A = SUNSparseMatrix(...);
   ```

   or similarly for the CUDA and SuperLU_DIST matrix modules (see the sections *The SUNMATRIX_CUSPARSE Module* or *The SUNMATRIX_SLUNRLOC Module* for further information).

   Similarly, if the problem involves a non-identity mass matrix, and the mass-matrix linear systems will be solved using a direct linear solver, then a template mass matrix must be created by using the appropriate functions defined by the particular SUNMATRIX implementation.

   NOTE: The dense, banded, and sparse matrix objects are usable only in a serial or threaded environment.

7. Create linear solver object

   If a nonlinear solver requiring a linear solver will be used (e.g., a Newton iteration), or if the problem involves a non-identity mass matrix, then the desired linear solver object(s) must be created by using the appropriate functions defined by the particular SUNLINSOL implementation.

   For any of the SUNDIALS-supplied SUNLINSOL implementations, the linear solver object may be created using a call of the form

   ```
   SUNLinearSolver LS = SUNLinSol_*(...);
   ```

   where * can be replaced with "Dense", "SPGMR", or other options, as discussed in the sections *Linear solver interface functions* and *Description of the SUNLinearSolver module*.

8. Set linear solver optional inputs

   Call `*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See the documentation for each SUNLINSOL module in the section *Description of the SUNLinearSolver module* for details.

9. Attach linear solver module

   If a linear solver was created above for implicit stage solves, initialize the ARKLS linear solver interface by attaching the linear solver object (and Jacobian matrix object, if applicable) with the call (for details see the section *Linear solver interface functions*):

   ```
   ier = ARKStepSetLinearSolver(...);
   ```

   Similarly, if the problem involves a non-identity mass matrix, initialize the ARKLS mass matrix linear solver interface by attaching the mass linear solver object (and mass matrix object, if applicable) with the call (for details see the section *Linear solver interface functions*):

   ```
   ier = ARKStepSetMassLinearSolver(...);
   ```

10. Create nonlinear solver object

    If the problem involves an implicit component, and if a non-default nonlinear solver object will be used for implicit stage solves (see the section *Nonlinear solver interface functions*), then the desired nonlinear solver object must be created by using the appropriate functions defined by the particular SUNNONLINSOL implementation (e.g., `NLS = SUNNonlinSol_***(...);` where `***` is the name of the nonlinear solver (see the section *Description of the SUNNonlinearSolver Module* for details).

    For the SUNDIALS-supplied SUNNONLINSOL implementations, the nonlinear solver object may be created using a call of the form

    ```
    SUNNonlinearSolver NLS = SUNNonlinSol_*(...);
    ```

    where * can be replaced with "Newton", "FixedPoint", or other options, as discussed in the sections *Nonlinear solver interface functions* and *Description of the SUNNonlinearSolver Module*.

11. Attach nonlinear solver module

    If a nonlinear solver object was created above, then it must be attached to ARKStep using the call (for details see the section *Nonlinear solver interface functions*):

    ```
    ier = ARKStepSetNonlinearSolver(...);
    ```

12. Set nonlinear solver optional inputs

    Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These *must* be called after attaching the nonlinear solver to ARKStep, otherwise the op-

tional inputs will be overridden by ARKStep defaults. See the section *Description of the SUNNonlinearSolver Module* for more information on optional inputs.

13. Set optional inputs

    Call `ARKStepSet*` functions to change any optional inputs that control the behavior of ARKStep from their default values. See the section *Optional input functions* for details.

14. Specify rootfinding problem

    Optionally, call *`ARKStepRootInit()`* to initialize a rootfinding problem to be solved during the integration of the ODE system. See the section *Rootfinding initialization function* for general details, and the section *Optional input functions* for relevant optional input calls.

15. Advance solution in time

    For each point at which output is desired, call

    ```
    ier = ARKStepEvolve(arkode_mem, tout, yout, &tret, itask);
    ```

    Here, `itask` specifies the return mode. The vector `yout` (which can be the same as the vector `y0` above) will contain $y(t_{\text{out}})$. See the section *ARKStep solver function* for details.

16. Get optional outputs

    Call `ARKStepGet*` functions to obtain optional output. See the section *Optional output functions* for details.

17. Deallocate memory for solution vector

    Upon completion of the integration, deallocate memory for the vector `y` (or `yout`) by calling the destructor function:

    ```
    N_VDestroy(y);
    ```

18. Free solver memory

    Call `ARKStepFree(&arkode_mem)` to free the memory allocated for the ARKStep module (and any nonlinear solver module).

19. Free linear solver and matrix memory

    Call *`SUNLinSolFree()`* and (possibly) *`SUNMatDestroy()`* to free any memory allocated for the linear solver and matrix objects created above.

20. Free nonlinear solver memory

    If a user-supplied `SUNNonlinearSolver` was provided to ARKStep, then call *`SUNNonlinSolFree()`* to free any memory allocated for the nonlinear solver object created above.

21. Finalize MPI, if used

    Call `MPI_Finalize` to terminate MPI.

SUNDIALS provides some linear solvers only as a means for users to get problems running and not as highly efficient solvers. For example, if solving a dense system, we suggest using the LAPACK solvers if the size of the linear system is $> 50,000$ (thanks to A. Nicolai for his testing and recommendation). The table below shows the linear solver interfaces available as `SUNLinearSolver` modules and the vector implementations required for use. As an example, one cannot use the dense direct solver interfaces with the MPI-based vector implementation. However, as discussed in section *Description of the SUNLinearSolver module* the SUNDIALS packages operate on generic `SUNLinearSolver` objects, allowing a user to develop their own solvers should they so desire.

4.4.1 SUNDIALS linear solver interfaces and vector implementations that can be used for each

Linear Solver Interface	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre* Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	X		X	X					X
Band	X		X	X					X
LapackDense	X		X	X					X
LapackBand	X		X	X					X
KLU	X		X	X					X
SuperLU_DIST	X	X	X	X	X	X			X
SuperLU_MT	X		X	X					X
SPGMR	X	X	X	X	X	X	X	X	X
SPFGMR	X	X	X	X	X	X	X	X	X
SPBCGS	X	X	X	X	X	X	X	X	X
SPTFQMR	X	X	X	X	X	X	X	X	X
PCG	X	X	X	X	X	X	X	X	X
User supplied	X	X	X	X	X	X	X	X	X

4.5 ARKStep User-callable functions

This section describes the functions that are called by the user to setup and then solve an IVP using the ARKStep time-stepping module. Some of these are required; however, starting with the section *Optional input functions*, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of ARKode's ARKStep module. In any case, refer to the preceding section, *A skeleton of the user's main program*, for the correct order of these calls.

On an error, each user-callable function returns a negative value (or `NULL` if the function returns a pointer) and sends an error message to the error handler routine, which prints the message to `stderr` by default. However, the user can set a file as error output or can provide her own error handler function (see the section *Optional input functions* for details).

4.5.1 ARKStep initialization and deallocation functions

void* **`ARKStepCreate`** (*ARKRhsFn fe*, *ARKRhsFn fi*, realtype *t0*, N_Vector *y0*)

This function creates an internal memory block for a problem to be solved using the ARKStep time-stepping module in ARKode.

Arguments:

- *fe* – the name of the C function (of type `ARKRhsFn()`) defining the explicit portion of the right-hand side function in $M\,\dot{y} = f^E(t,y) + f^I(t,y)$.
- *fi* – the name of the C function (of type `ARKRhsFn()`) defining the implicit portion of the right-hand side function in $M\,\dot{y} = f^E(t,y) + f^I(t,y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.

Return value: If successful, a pointer to initialized problem memory of type `void*`, to be passed to all user-facing ARKStep routines listed below. If unsuccessful, a `NULL` pointer will be returned, and an error message will be printed to `stderr`.

void **ARKStepFree**(void** *arkode_mem*)
This function frees the problem memory *arkode_mem* created by *ARKStepCreate()*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value: None

4.5.2 ARKStep tolerance specification functions

These functions specify the integration tolerances. One of them **should** be called before the first call to *ARKStepEvolve()*; otherwise default values of `reltol = 1e-4` and `abstol = 1e-9` will be used, which may be entirely incorrect for a specific problem.

The integration tolerances `reltol` and `abstol` define a vector of error weights, `ewt`. In the case of *ARKStepSStolerances()*, this vector has components

```
ewt[i] = 1.0/(reltol*abs(y[i]) + abstol);
```

whereas in the case of *ARKStepSVtolerances()* the vector components are given by

```
ewt[i] = 1.0/(reltol*abs(y[i]) + abstol[i]);
```

This vector is used in all error and convergence tests, which use a weighted RMS norm on all error-like vectors v:

$$\|v\|_{WRMS} = \left(\frac{1}{N} \sum_{i=1}^{N} (v_i \; ewt_i)^2 \right)^{1/2},$$

where N is the problem dimension.

Alternatively, the user may supply a custom function to supply the `ewt` vector, through a call to *ARKStepWFtolerances()*.

int **ARKStepSStolerances**(void* *arkode_mem*, realtype *reltol*, realtype *abstol*)
This function specifies scalar relative and absolute tolerances.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – scalar absolute tolerance.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKStepSVtolerances**(void* *arkode_mem*, realtype *reltol*, N_Vector *abstol*)
This function specifies a scalar relative tolerance and a vector absolute tolerance (a potentially different absolute tolerance for each vector component).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

- *reltol* – scalar relative tolerance.
- *abstol* – vector containing the absolute tolerances for each solution component.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKStepWFtolerances**(void* *arkode_mem*, *ARKEwtFn efun*)

This function specifies a user-supplied function *efun* to compute the error weight vector `ewt`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *efun* – the name of the function (of type *ARKEwtFn()*) that implements the error weight vector computation.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ARKStep memory was not allocated by the time-stepping module

Moreover, for problems involving a non-identity mass matrix $M \neq I$, the units of the solution vector y may differ from the units of the IVP, posed for the vector My. When this occurs, iterative solvers for the Newton linear systems and the mass matrix linear systems may require a different set of tolerances. Since the relative tolerance is dimensionless, but the absolute tolerance encodes a measure of what is "small" in the units of the respective quantity, a user may optionally define absolute tolerances in the equation units. In this case, ARKStep defines a vector of residual weights, `rwt` for measuring convergence of these iterative solvers. In the case of *ARKStepResStolerance()*, this vector has components

```
rwt[i] = 1.0/(reltol*abs(My[i]) + rabstol);
```

whereas in the case of *ARKStepResVtolerance()* the vector components are given by

```
rwt[i] = 1.0/(reltol*abs(My[i]) + rabstol[i]);
```

This residual weight vector is used in all iterative solver convergence tests, which similarly use a weighted RMS norm on all residual-like vectors v:

$$\|v\|_{WRMS} = \left(\frac{1}{N} \sum_{i=1}^{N} (v_i \; rwt_i)^2 \right)^{1/2},$$

where N is the problem dimension.

As with the error weight vector, the user may supply a custom function to supply the `rwt` vector, through a call to *ARKStepResFtolerance()*. Further information on all three of these functions is provided below.

int **ARKStepResStolerance**(void* *arkode_mem*, realtype *abstol*)

This function specifies a scalar absolute residual tolerance.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *rabstol* – scalar absolute residual tolerance.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKStepResVtolerance**(void* *arkode_mem*, N_Vector *rabstol*)
This function specifies a vector of absolute residual tolerances.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *rabstol* – vector containing the absolute residual tolerances for each solution component.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ARKStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKStepResFtolerance**(void* *arkode_mem*, *ARKRwtFn* *rfun*)
This function specifies a user-supplied function *rfun* to compute the residual weight vector `rwt`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *rfun* – the name of the function (of type *`ARKRwtFn()`*) that implements the residual weight vector computation.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ARKStep memory was not allocated by the time-stepping module

4.5.2.1 General advice on the choice of tolerances

For many users, the appropriate choices for tolerance values in `reltol`, `abstol`, and `rabstol` are a concern. The following pieces of advice are relevant.

1. The scalar relative tolerance `reltol` is to be set to control relative errors. So a value of 10^{-4} means that errors are controlled to .01%. We do not recommend using `reltol` larger than 10^{-3}. On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 10^{-15} for double-precision).

2. The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector y may be so small that pure relative error control is meaningless. For example, if y_i starts at some nonzero value, but in time decays to zero, then pure relative error control on y_i makes no sense (and is overly costly) after y_i is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. For example, see the example problem `ark_robertson.c`, and the discussion of it in the ARKode Examples Documentation *[R2018]*. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the `atols` vector therein. It is impossible to give any general

advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.

3. The residual absolute tolerances `rabstol` (whether scalar or vector) follow a similar explanation as for `abstol`, except that these should be set to the noise level of the equation components, i.e. the noise level of My. For problems in which $M = I$, it is recommended that `rabstol` be left unset, which will default to the already-supplied `abstol` values.

4. Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual step. The final (global) errors are an accumulation of those per-step errors, where that accumulation factor is problem-dependent. A general rule of thumb is to reduce the tolerances by a factor of 10 from the actual desired limits on errors. So if you want .01% relative accuracy (globally), a good choice for `reltol` is 10^{-5}. In any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

4.5.2.2 Advice on controlling nonphysical negative values

In many applications, some components in the true solution are always positive or non-negative, though at times very small. In the numerical solution, however, small negative (nonphysical) values can then occur. In most cases, these values are harmless, and simply need to be controlled, not eliminated, but in other cases any value that violates a constraint may cause a simulation to halt. For both of these scenarios the following pieces of advice are relevant.

1. The best way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.

2. If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in y returned by ARKStep, with magnitude comparable to `abstol` or less, is equivalent to zero as far as the computation is concerned.

3. The user's right-hand side routines f^E and f^I should never change a negative value in the solution vector y to a non-negative value in attempt to "fix" this problem, since this can lead to numerical instability. If the f^E or f^I routines cannot tolerate a zero or negative value (e.g. because there is a square root or log), then the offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input y vector) for the purposes of computing $f^E(t, y)$ or $f^I(t, y)$.

4. Positivity and non-negativity constraints on components can be enforced by use of the recoverable error return feature in the user-supplied right-hand side functions, f^E and f^I. When a recoverable error is encountered, ARKStep will retry the step with a smaller step size, which typically alleviates the problem. However, because this option involves some additional overhead cost, it should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

4.5.3 Linear solver interface functions

As previously explained, the Newton iterations used in solving implicit systems within ARKStep require the solution of linear systems of the form

$$\mathcal{A}\left(z_i^{(m)}\right)\delta^{(m+1)} = -G\left(z_i^{(m)}\right)$$

where

$$\mathcal{A} \approx M - \gamma J, \qquad J = \frac{\partial f^I}{\partial y}.$$

ARKode's ARKLS linear solver interface supports all valid `SUNLinearSolver` modules for this task.

Matrix-based SUNLinearSolver modules utilize SUNMatrix objects to store the approximate Jacobian matrix J, the Newton matrix $\mathcal{A}$, the mass matrix M, and, when using direct solvers, the factorizations used throughout the solution process.

Matrix-free SUNLinearSolver modules instead use iterative methods to solve the Newton systems of equations, and only require the *action* of the matrix on a vector, $\mathcal{A}v$. With most of these methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver portions of the sections *Optional input functions* and *User-supplied functions*.

If preconditioning is done, user-supplied functions should be used to define left and right preconditioner matrices P_1 and P_2 (either of which could be the identity matrix), such that the product $P_1 P_2$ approximates the Newton matrix $\mathcal{A} = M - \gamma J$.

To specify a generic linear solver for ARKStep to use for the Newton systems, after the call to *ARKStepCreate()* but before any calls to *ARKStepEvolve()*, the user's program must create the appropriate SUNLinearSolver object and call the function *ARKStepSetLinearSolver()*, as documented below. To create the SUNLinearSolver object, the user may call one of the SUNDIALS-packaged SUNLinSol module constructor routines via a call of the form

```
SUNLinearSolver LS = SUNLinSol_*(...);
```

The current list of such constructor routines includes *SUNLinSol_Dense()*, *SUNLinSol_Band()*, *SUNLinSol_LapackDense()*, *SUNLinSol_LapackBand()*, *SUNLinSol_KLU()*, *SUNLinSol_SuperLUMT()*, *SUNLinSol_SuperLUDIST()*, *SUNLinSol_cuSolverSp_batchQR()*, *SUNLinSol_SPGMR()*, *SUNLinSol_SPFGMR()*, *SUNLinSol_SPBCGS()*, *SUNLinSol_SPTFQMR()*, and *SUNLinSol_PCG()*.

Alternately, a user-supplied SUNLinearSolver module may be created and used instead. The use of each of the generic linear solvers involves certain constants, functions and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific SUNMatrix or SUNLinearSolver module in question, as described in the sections *Matrix Data Structures* and *Description of the SUNLinearSolver module*.

Once this solver object has been constructed, the user should attach it to ARKStep via a call to *ARKStepSetLinearSolver()*. The first argument passed to this function is the ARKStep memory pointer returned by *ARKStepCreate()*; the second argument is the SUNLinearSolver object created above. The third argument is an optional SUNMatrix object to accompany matrix-based SUNLinearSolver inputs (for matrix-free linear solvers, the third argument should be NULL). A call to this function initializes the ARKLS linear solver interface, linking it to the ARKStep integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

int **ARKStepSetLinearSolver**(void* *arkode_mem*, SUNLinearSolver *LS*, SUNMatrix *J*)

This function specifies the SUNLinearSolver object that ARKStep should use, as well as a template Jacobian SUNMatrix object (if applicable).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *LS* – the SUNLinearSolver object to use.
- *J* – the template Jacobian SUNMatrix object to use (or NULL if not applicable).

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was NULL

- *ARKLS_MEM_FAIL* if there was a memory allocation failure
- *ARKLS_ILL_INPUT* if ARKLS is incompatible with the provided *LS* or *J* input objects, or the current `N_Vector` module.

Notes: If *LS* is a matrix-free linear solver, then the *J* argument should be `NULL`.

If *LS* is a matrix-based linear solver, then the template Jacobian matrix *J* will be used in the solve process, so if additional storage is required within the `SUNMatrix` object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular SUNMATRIX type in the section *Matrix Data Structures* for further information).

When using sparse linear solvers, it is typically much more efficient to supply *J* so that it includes the full sparsity pattern of the Newton system matrices $\mathcal{A} = I - \gamma J$ (or $\mathcal{A} = M - \gamma J$ in the case of non-identity mass matrix), even if *J* itself has zeros in nonzero locations of I (or M). The reasoning for this is that $\mathcal{A}$ is constructed in-place, on top of the user-specified values of *J*, so if the sparsity pattern in *J* is insufficient to store $\mathcal{A}$ then it will need to be resized internally by ARKStep.

4.5.4 Mass matrix solver specification functions

As discussed in section *Mass matrix solver (ARKStep only)*, if the ODE system involves a non-identity mass matrix $M \neq I$, then ARKStep must solve linear systems of the form

$$Mx = b.$$

ARKode's ARKLS mass-matrix linear solver interface supports all valid `SUNLinearSolver` modules for this task. For iterative linear solvers, user-supplied preconditioning can be applied. For the specification of a preconditioner, see the iterative linear solver portions of the sections *Optional input functions* and *User-supplied functions*. If preconditioning is to be performed, user-supplied functions should be used to define left and right preconditioner matrices P_1 and P_2 (either of which could be the identity matrix), such that the product $P_1 P_2$ approximates the mass matrix M.

To specify a generic linear solver for ARKStep to use for mass matrix systems, after the call to `ARKStepCreate()` but before any calls to `ARKStepEvolve()`, the user's program must create the appropriate `SUNLinearSolver` object and call the function `ARKStepSetMassLinearSolver()`, as documented below. The first argument passed to this functions is the ARKStep memory pointer returned by `ARKStepCreate()`; the second argument is the desired `SUNLinearSolver` object to use for solving mass matrix systems. The third object is a template `SUNMatrix` to use with the provided `SUNLinearSolver` (if applicable). The fourth input is a flag to indicate whether the mass matrix is time-dependent, i.e. $M = M(t)$ or not. A call to this function initializes the ARKLS mass matrix linear solver interface, linking this to the main ARKStep integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

The use of each of the generic linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific `SUNMatrix` or `SUNLinearSolver` module in question, as described in the sections *Matrix Data Structures* and *Description of the SUNLinearSolver module*.

Note: if the user program includes linear solvers for *both* the Newton and mass matrix systems, these must have the same type:

- If both are matrix-based, then they must utilize the same `SUNMatrix` type, since these will be added when forming the Newton system matrices $\mathcal{A}$. In this case, both the Newton and mass matrix linear solver interfaces can use the same `SUNLinearSolver` object, although different solver objects (e.g. with different solver parameters) are also allowed.
- If both are matrix-free, then the Newton and mass matrix `SUNLinearSolver` objects must be different. These may even use different solver algorithms (SPGMR, SPBCGS, etc.), if desired. For example, if the mass

matrix is symmetric but the Jacobian is not, then PCG may be used for the mass matrix systems and SPGMR for the Newton systems.

As with the Newton system solvers, the mass matrix linear system solvers listed below are all built on top of generic SUNDIALS solver modules.

int **ARKStepSetMassLinearSolver**(void* *arkode_mem*, SUNLinearSolver *LS*, SUNMatrix *M*, booleantype *time_dep*)

This function specifies the `SUNLinearSolver` object that ARKStep should use for mass matrix systems, as well as a template `SUNMatrix` object.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *LS* – the `SUNLinearSolver` object to use.
- *M* – the template mass `SUNMatrix` object to use.
- *time_dep* – flag denoting whether the mass matrix depends on the independent variable ($M = M(t)$) or not ($M \neq M(t)$). `SUNTRUE` indicates time-dependence of the mass matrix.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_MEM_FAIL* if there was a memory allocation failure
- *ARKLS_ILL_INPUT* if ARKLS is incompatible with the provided *LS* or *M* input objects, or the current `N_Vector` module.

Notes: If *LS* is a matrix-free linear solver, then the *M* argument should be `NULL`.

If *LS* is a matrix-based linear solver, then the template mass matrix *M* will be used in the solve process, so if additional storage is required within the `SUNMatrix` object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size.

If called with *time_dep* set to `SUNFALSE`, then the mass matrix is only computed and factored once (or when either *ARKStepReInit()* or :c:func‘ARKStepResize()‘ are called), with the results reused throughout the entire ARKStep simulation.

Unlike the system Jacobian, the system mass matrix is not approximated using finite-differences of any functions provided to ARKStep. Hence, use of the a matrix-based *LS* requires the user to provide a mass-matrix constructor routine (see *ARKLsMassFn* and *ARKStepSetMassFn()*).

Similarly, the system mass matrix-vector-product is not approximated using finite-differences of any functions provided to ARKStep. Hence, use of a matrix-free *LS* requires the user to provide a mass-matrix-times-vector product routine (see *ARKLsMassTimesVecFn* and *ARKStepSetMassTimes()*).

4.5.5 Nonlinear solver interface functions

When changing the nonlinear solver in ARKStep, after the call to *ARKStepCreate()* but before any calls to *ARKStepEvolve()*, the user's program must create the appropriate SUNNonlinSol object and call *ARKStepSetNonlinearSolver()*, as documented below. If any calls to *ARKStepEvolve()* have been made, then ARKStep will need to be reinitialized by calling *ARKStepReInit()* to ensure that the nonlinear solver is initialized correctly before any subsequent calls to *ARKStepEvolve()*.

The first argument passed to the routine *ARKStepSetNonlinearSolver()* is the ARKStep memory pointer returned by *ARKStepCreate()*; the second argument passed to this function is the desired SUNNonlinSol object

to use for solving the nonlinear system for each implicit stage. A call to this function attaches the nonlinear solver to the main ARKStep integrator.

int **ARKStepSetNonlinearSolver**(void* *arkode_mem*, SUNNonlinearSolver *NLS*)

This function specifies the `SUNNonlinearSolver` object that ARKStep should use for implicit stage solves.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *NLS* – the `SUNNonlinearSolver` object to use.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_MEM_FAIL* if there was a memory allocation failure
- *ARK_ILL_INPUT* if ARKStep is incompatible with the provided *NLS* input object.

Notes: ARKStep will use the Newton SUNNonlinSol module by default; a call to this routine replaces that module with the supplied *NLS* object.

4.5.6 Rootfinding initialization function

As described in the section *Rootfinding*, while solving the IVP, ARKode's time-stepping modules have the capability to find the roots of a set of user-defined functions. To activate the root-finding algorithm, call the following function. This is normally called only once, prior to the first call to `ARKStepEvolve()`, but if the rootfinding problem is to be changed during the solution, `ARKStepRootInit()` can also be called prior to a continuation call to `ARKStepEvolve()`.

int **ARKStepRootInit**(void* *arkode_mem*, int *nrtfn*, *ARKRootFn* *g*)

Initializes a rootfinding problem to be solved during the integration of the ODE system. It must be called after `ARKStepCreate()`, and before `ARKStepEvolve()`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nrtfn* – number of functions g_i, an integer ≥ 0.
- *g* – name of user-supplied function, of type `ARKRootFn()`, defining the functions g_i whose roots are sought.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_MEM_FAIL* if there was a memory allocation failure
- *ARK_ILL_INPUT* if *nrtfn* is greater than zero but *g* = `NULL`.

Notes: To disable the rootfinding feature after it has already been initialized, or to free memory associated with ARKStep's rootfinding module, call *ARKStepRootInit* with *nrtfn = 0*.

Similarly, if a new IVP is to be solved with a call to `ARKStepReInit()`, where the new IVP has no rootfinding problem but the prior one did, then call *ARKStepRootInit* with *nrtfn = 0*.

4.5.7 ARKStep solver function

This is the central step in the solution process – the call to perform the integration of the IVP. The input argument *itask* specifies one of two modes as to where ARKStep is to return a solution. These modes are modified if the user has set a stop time (with a call to the optional input function `ARKStepSetStopTime()`) or has requested rootfinding.

int **`ARKStepEvolve`**(void* *arkode_mem*, realtype *tout*, N_Vector *yout*, realtype **tret*, int *itask*)

Integrates the ODE over an interval in t.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *tout* – the next time at which a computed solution is desired.
- *yout* – the computed solution vector.
- *tret* – the time corresponding to *yout* (output).
- *itask* – a flag indicating the job of the solver for the next user step.

  The *ARK_NORMAL* option causes the solver to take internal steps until it has just overtaken a user-specified output time, *tout*, in the direction of integration, i.e. $t_{n-1} <$ *tout* $\leq t_n$ for forward integration, or $t_n \leq$ *tout* $< t_{n-1}$ for backward integration. It will then compute an approximation to the solution $y(tout)$ by interpolation (using one of the dense output routines described in the section *Interpolation*).

  The *ARK_ONE_STEP* option tells the solver to only 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 *tout* then the solver will again return an interpolated result; otherwise it will return a copy of the internal solution y_n in the vector *yout*

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_ROOT_RETURN* if `ARKStepEvolve()` succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call `ARKStepGetRootInfo()` to see which g_i were found to have a root at (**tret*).
- *ARK_TSTOP_RETURN* if `ARKStepEvolve()` succeeded and returned at *tstop*.
- *ARK_MEM_NULL* if the *arkode_mem* argument was `NULL`.
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if one of the inputs to `ARKStepEvolve()` is illegal, or some other input to the solver was either illegal or missing. Details will be provided in the error message. Typical causes of this failure:

  1. A component of the error weight vector became zero during internal time-stepping.
  2. The linear solver initialization function (called by the user after calling `ARKStepCreate()`) failed to set the linear solver-specific *lsolve* field in *arkode_mem*.
  3. A root of one of the root functions was found both at a point t and also very near t.
  4. The initial condition violates the inequality constraints.

- *ARK_TOO_MUCH_WORK* if the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is *MXSTEP_DEFAULT = 500*.
- *ARK_TOO_MUCH_ACC* if the solver could not satisfy the accuracy demanded by the user for some internal step.

- *ARK_ERR_FAILURE* if error test failures occurred either too many times (*ark_maxnef*) during one internal time step or occurred with $|h| = h_{min}$.
- *ARK_CONV_FAILURE* if either convergence test failures occurred too many times (*ark_maxncf*) during one internal time step or occurred with $|h| = h_{min}$.
- *ARK_LINIT_FAIL* if the linear solver's initialization function failed.
- *ARK_LSETUP_FAIL* if the linear solver's setup routine failed in an unrecoverable manner.
- *ARK_LSOLVE_FAIL* if the linear solver's solve routine failed in an unrecoverable manner.
- *ARK_MASSINIT_FAIL* if the mass matrix solver's initialization function failed.
- *ARK_MASSSETUP_FAIL* if the mass matrix solver's setup routine failed.
- *ARK_MASSSOLVE_FAIL* if the mass matrix solver's solve routine failed.
- *ARK_VECTOROP_ERR* a vector operation error occured.

Notes: The input vector *yout* can use the same memory as the vector *y0* of initial conditions that was passed to `ARKStepCreate()`.

In *ARK_ONE_STEP* mode, *tout* is used only on the first call, and only to get the direction and a rough scale of the independent variable.

All failure return values are negative and so testing the return argument for negative values will trap all `ARKStepEvolve()` failures.

Since interpolation may reduce the accuracy in the reported solution, if full method accuracy is desired the user should issue a call to `ARKStepSetStopTime()` before the call to `ARKStepEvolve()` to specify a fixed stop time to end the time step and return to the user. Upon return from `ARKStepEvolve()`, a copy of the internal solution y_n will be returned in the vector *yout*. Once the integrator returns at a *tstop* time, any future testing for *tstop* is disabled (and can be re-enabled only though a new call to `ARKStepSetStopTime()`).

On any error return in which one or more internal steps were taken by `ARKStepEvolve()`, the returned values of *tret* and *yout* correspond to the farthest point reached in the integration. On all other error returns, *tret* and *yout* are left unchanged from those provided to the routine.

4.5.8 Optional input functions

There are numerous optional input parameters that control the behavior of the ARKStep solver, each of which may be modified from its default value through calling an appropriate input function. The following tables list all optional input functions, grouped by which aspect of ARKStep they control. Detailed information on the calling syntax and arguments for each function are then provided following each table.

The optional inputs are grouped into the following categories:

- General ARKStep options (*Optional inputs for ARKStep*),
- IVP method solver options (*Optional inputs for IVP method selection*),
- Step adaptivity solver options (*Optional inputs for time step adaptivity*),
- Implicit stage solver options (*Optional inputs for implicit stage solves*),
- Linear solver interface options (*Linear solver interface optional input functions*), and
- Rootfinding options (*Rootfinding optional input functions*).

For the most casual use of ARKStep, relying on the default set of solver parameters, the reader can skip to the following section, *User-supplied functions*.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so a test on the return arguments for negative values will catch all errors. Finally, a call to an `ARKStepSet***` function can generally be made from the user's calling program at any time and, if successful, takes effect immediately. `ARKStepSet***` functions that cannot be called at any time note this in the "**Notes**:" section of the function documentation.

4.5.8.1 Optional inputs for ARKStep

Optional input	Function name	Default
Return ARKStep parameters to their defaults	*ARKStepSetDefaults()*	internal
Set dense output interpolation type	*ARKStepSetInterpolantType()*	ARK_INTERP_HERMITE
Set dense output polynomial degree	*ARKStepSetInterpolantDegree()*	5
Supply a pointer to a diagnostics output file	*ARKStepSetDiagnostics()*	NULL
Supply a pointer to an error output file	*ARKStepSetErrFile()*	stderr
Supply a custom error handler function	*ARKStepSetErrHandlerFn()*	internal fn
Disable time step adaptivity (fixed-step mode)	*ARKStepSetFixedStep()*	disabled
Supply an initial step size to attempt	*ARKStepSetInitStep()*	estimated
Maximum no. of warnings for $t_n + h = t_n$	*ARKStepSetMaxHnilWarns()*	10
Maximum no. of internal steps before *tout*	*ARKStepSetMaxNumSteps()*	500
Maximum absolute step size	*ARKStepSetMaxStep()*	∞
Minimum absolute step size	*ARKStepSetMinStep()*	0.0
Set a value for t_{stop}	*ARKStepSetStopTime()*	∞
Supply a pointer for user data	*ARKStepSetUserData()*	NULL
Maximum no. of ARKStep error test failures	*ARKStepSetMaxErrTestFails()*	7
Set 'optimal' adaptivity params. for a method	*ARKStepSetOptimalParams()*	internal
Set inequality constraints on solution	*ARKStepSetConstraints()*	NULL
Set max number of constraint failures	*ARKStepSetMaxNumConstrFails()*	10

int **ARKStepSetDefaults**(void* *arkode_mem*)

Resets all optional input parameters to ARKStep's original default values.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Does not change the *user_data* pointer or any parameters within the specified time-stepping module.

Also leaves alone any data structures or options related to root-finding (those can be reset using *ARKStepRootInit()*).

int **ARKStepSetInterpolantType**(void* *arkode_mem*, int *itype*)

Specifies use of the Lagrange or Hermite interpolation modules (used for dense output – interpolation of solution output values and implicit method predictors).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *itype* – requested interpolant type (`ARK_INTERP_HERMITE` or `ARK_INTERP_LAGRANGE`)

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_MEM_FAIL* if the interpolation module cannot be allocated
- *ARK_ILL_INPUT* if the *itype* argument is not recognized or the interpolation module has already been initialized

Notes: The Hermite interpolation module is described in the Section *Hermite interpolation module*, and the Lagrange interpolation module is described in the Section *Lagrange interpolation module*.

This routine frees any previously-allocated interpolation module, and re-creates one according to the specified argument. Thus any previous calls to `ARKStepSetInterpolantDegree()` will be nullified.

This routine may only be called *after* the call to `ARKStepCreate()`. After the first call to `ARKStepEvolve()` the interpolation type may not be changed without first calling `ARKStepReInit()`.

If this routine is not called, the Hermite interpolation module will be used.

int **ARKStepSetInterpolantDegree**(void* *arkode_mem*, int *degree*)

Specifies the degree of the polynomial interpolant used for dense output (i.e. interpolation of solution output values and implicit method predictors).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *degree* – requested polynomial degree.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory or interpolation module are `NULL`
- *ARK_INTERP_FAIL* if this is called after `ARKStepEvolve()`
- *ARK_ILL_INPUT* if an argument has an illegal value or the interpolation module has already been initialized

Notes: Allowed values are between 0 and 5.

This routine should be called *after* `ARKStepCreate()` and *before* `ARKStepEvolve()`. After the first call to `ARKStepEvolve()` the interpolation degree may not be changed without first calling `ARKStepReInit()`.

If a user calls both this routine and `ARKStepSetInterpolantType()`, then `ARKStepSetInterpolantType()` must be called first.

Since the accuracy of any polynomial interpolant is limited by the accuracy of the time-step solutions on which it is based, the *actual* polynomial degree that is used by ARKStep will be the minimum of $q - 1$ and the input *degree*, where q is the order of accuracy for the time integration method.

int **ARKStepSetDenseOrder**(void* *arkode_mem*, int *dord*)

This function is deprecated, and will be removed in a future release. Users should transition to calling `ARKStepSetInterpolantDegree()` *instead.*

int **ARKStepSetDiagnostics**(void* *arkode_mem*, FILE* *diagfp*)
Specifies the file pointer for a diagnostics file where all ARKStep step adaptivity and solver information is written.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *diagfp* – pointer to the diagnostics output file.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This parameter can be `stdout` or `stderr`, although the suggested approach is to specify a pointer to a unique file opened by the user and returned by `fopen`. If not called, or if called with a `NULL` file pointer, all diagnostics output is disabled.

When run in parallel, only one process should set a non-NULL value for this pointer, since statistics from all processes would be identical.

int **ARKStepSetErrFile**(void* *arkode_mem*, FILE* *errfp*)
Specifies a pointer to the file where all ARKStep warning and error messages will be written if the default internal error handling function is used.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *errfp* – pointer to the output file.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value for *errfp* is `stderr`.

Passing a `NULL` value disables all future error message output (except for the case wherein the ARKStep memory pointer is `NULL`). This use of the function is strongly discouraged.

If used, this routine should be called before any other optional input functions, in order to take effect for subsequent error messages.

int **ARKStepSetErrHandlerFn**(void* *arkode_mem*, *ARKErrHandlerFn* *ehfun*, void* *eh_data*)
Specifies the optional user-defined function to be used in handling error messages.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ehfun* – name of user-supplied error handler function.
- *eh_data* – pointer to user data passed to *ehfun* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Error messages indicating that the ARKStep solver memory is `NULL` will always be directed to `stderr`.

int **ARKStepSetFixedStep**(void* *arkode_mem*, realtype *hfixed*)

Disabled time step adaptivity within ARKStep, and specifies the fixed time step size to use for all internal steps.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hfixed* – value of the fixed step size to use.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass 0.0 to return ARKStep to the default (adaptive-step) mode.

Use of this function is not recommended, since we it gives no assurance of the validity of the computed solutions. It is primarily provided for code-to-code verification testing purposes.

When using *ARKStepSetFixedStep()*, any values provided to the functions *ARKStepSetInitStep()*, *ARKStepSetAdaptivityFn()*, *ARKStepSetMaxErrTestFails()*, *ARKStepSetAdaptivityMethod()*, *ARKStepSetCFLFraction()*, *ARKStepSetErrorBias()*, *ARKStepSetFixedStepBounds()*, *ARKStepSetMaxCFailGrowth()*, *ARKStepSetMaxEFailGrowth()*, *ARKStepSetMaxFirstGrowth()*, *ARKStepSetMaxGrowth()*, *ARKStepSetMinReduction()*, *ARKStepSetSafetyFactor()*, *ARKStepSetSmallNumEFails()* and *ARKStepSetStabilityFn()* will be ignored, since temporal adaptivity is disabled.

If both *ARKStepSetFixedStep()* and *ARKStepSetStopTime()* are used, then the fixed step size will be used for all steps until the final step preceding the provided stop time (which may be shorter). To resume use of the previous fixed step size, another call to *ARKStepSetFixedStep()* must be made prior to calling *ARKStepEvolve()* to resume integration.

It is *not* recommended that *ARKStepSetFixedStep()* be used in concert with *ARKStepSetMaxStep()* or *ARKStepSetMinStep()*, since at best those latter two routines will provide no useful information to the solver, and at worst they may interfere with the desired fixed step size.

int **ARKStepSetInitStep**(void* *arkode_mem*, realtype *hin*)

Specifies the initial time step size ARKStep should use after initialization, re-initialization, or resetting.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hin* – value of the initial step to be attempted ($\neq 0$).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass 0.0 to use the default value.

By default, ARKStep estimates the initial step size to be the solution h of the equation $\left\| \frac{h^2 \ddot{y}}{2} \right\| = 1$, where $\ddot{y}$ is an estimated value of the second derivative of the solution at *t0*.

This routine will also reset the step size and error history.

int **ARKStepSetMaxHnilWarns**(void* *arkode_mem*, int *mxhnil*)
Specifies the maximum number of messages issued by the solver to warn that $t + h = t$ on the next internal step, before ARKStep will instead return with an error.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mxhnil* – maximum allowed number of warning messages (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 10; set *mxhnil* to zero to specify this default.

A negative value indicates that no warning messages should be issued.

int **ARKStepSetMaxNumSteps**(void* *arkode_mem*, long int *mxsteps*)
Specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time, before ARKStep will return with an error.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mxsteps* – maximum allowed number of internal steps.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Passing *mxsteps* = 0 results in ARKStep using the default value (500).

Passing *mxsteps* < 0 disables the test (not recommended).

int **ARKStepSetMaxStep**(void* *arkode_mem*, realtype *hmax*)
Specifies the upper bound on the magnitude of the time step size.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hmax* – maximum absolute value of the time step size (≥ 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass *hmax* ≤ 0.0 to set the default value of ∞.

int **ARKStepSetMinStep**(void* *arkode_mem*, realtype *hmin*)
Specifies the lower bound on the magnitude of the time step size.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hmin* – minimum absolute value of the time step size (≥ 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass *hmin* ≤ 0.0 to set the default value of 0.

int **ARKStepSetStopTime**(void* *arkode_mem*, realtype *tstop*)
Specifies the value of the independent variable t past which the solution is not to proceed.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *tstop* – stopping time for the integrator.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default is that no stop time is imposed.

int **ARKStepSetUserData**(void* *arkode_mem*, void* *user_data*)
Specifies the user data block *user_data* and attaches it to the main ARKStep memory block.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *user_data* – pointer to the user data.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: If specified, the pointer to *user_data* is passed to all user-supplied functions for which it is an argument; otherwise `NULL` is passed.

If *user_data* is needed in user linear solver or preconditioner functions, the call to this function must be made *before* the call to specify the linear solver.

int **ARKStepSetMaxErrTestFails**(void* *arkode_mem*, int *maxnef*)
Specifies the maximum number of error test failures permitted in attempting one step, before returning with an error.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

- *maxnef* – maximum allowed number of error test failures (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 7; set *maxnef* ≤ 0 to specify this default.

int **ARKStepSetOptimalParams**(void* *arkode_mem*)

Sets all adaptivity and solver parameters to our 'best guess' values, for a given integration method (ERK, DIRK, ARK) and a given method order.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Should only be called after the method order and integration method have been set. These values resulted from repeated testing of ARKStep's solvers on a variety of training problems. However, all problems are different, so these values may not be optimal for all users.

int **ARKStepSetConstraints**(void* *arkode_mem*, N_Vector *constraints*)

Specifies a vector defining inequality constraints for each component of the solution vector y.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *constraints* – vector of constraint flags. If `constraints[i]` is
  - 0.0 then no constraint is imposed on y_i
  - 1.0 then y_i will be constrained to be $y_i \geq 0.0$
  - -1.0 then y_i will be constrained to be $y_i \leq 0.0$
  - 2.0 then y_i will be constrained to be $y_i > 0.0$
  - -2.0 then y_i will be constrained to be $y_i < 0.0$

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if the constraints vector contains illegal values

Notes: The presence of a non-`NULL` constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of `constraints` will result in an illegal input return. A `NULL` constraints vector will disable constraint checking.

After a call to *ARKStepResize()* inequality constraint checking will be disabled and a call to *ARKStepSetConstraints()* is required to re-enable constraint checking.

Since constraint-handling is performed through cutting time steps that would violate the constraints, it is possible that this feature will cause some problems to fail due to an inability to enforce constraints

even at the minimum time step size. Additionally, the features `ARKStepSetConstraints()` and `ARKStepSetFixedStep()` are incompatible, and should not be used simultaneously.

int **ARKStepSetMaxNumConstrFails**(void* *arkode_mem*, int *maxfails*)
Specifies the maximum number of constraint failures in a step before ARKStep will return with an error.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *maxfails* – maximum allowed number of constrain failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

Notes: Passing *maxfails* <= 0 results in ARKStep using the default value (10).

4.5.8.2 Optional inputs for IVP method selection

Optional input	Function name	Default
Set integrator method order	`ARKStepSetOrder()`	4
Specify implicit/explicit problem	`ARKStepSetImEx()`	`SUNTRUE`
Specify explicit problem	`ARKStepSetExplicit()`	`SUNFALSE`
Specify implicit problem	`ARKStepSetImplicit()`	`SUNFALSE`
Set additive RK tables	`ARKStepSetTables()`	internal
Specify additive RK table numbers	`ARKStepSetTableNum()`	internal

int **ARKStepSetOrder**(void* *arkode_mem*, int *ord*)
Specifies the order of accuracy for the ARK/DIRK/ERK integration method.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ord* – requested order of accuracy.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: For explicit methods, the allowed values are $2 \leq ord \leq 8$. For implicit methods, the allowed values are $2 \leq ord \leq 5$, and for ImEx methods the allowed values are $3 \leq ord \leq 5$. Any illegal input will result in the default value of 4.

Since *ord* affects the memory requirements for the internal ARKStep memory block, it cannot be changed after the first call to `ARKStepEvolve()`, unless `ARKStepReInit()` is called.

int **ARKStepSetImEx**(void* *arkode_mem*)
Specifies that both the implicit and explicit portions of problem are enabled, and to use an additive Runge Kutta method.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This is automatically deduced when neither of the function pointers *fe* or *fi* passed to *ARKStepCreate()* are NULL, but may be set directly by the user if desired.

int **ARKStepSetExplicit**(void* *arkode_mem*)

Specifies that the implicit portion of problem is disabled, and to use an explicit RK method.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This is automatically deduced when the function pointer *fi* passed to *ARKStepCreate()* is NULL, but may be set directly by the user if desired.

If the problem is posed in explicit form, i.e. $\dot{y} = f(t, y)$, then we recommend that the ERKStep time-stepper module be used instead.

int **ARKStepSetImplicit**(void* *arkode_mem*)

Specifies that the explicit portion of problem is disabled, and to use a diagonally implicit RK method.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This is automatically deduced when the function pointer *fe* passed to *ARKStepCreate()* is NULL, but may be set directly by the user if desired.

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

Specifies a customized Butcher table (or pair) for the ERK, DIRK, or ARK method.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *q* – global order of accuracy for the ARK method.
- *p* – global order of accuracy for the embedded ARK method.
- *Bi* – the Butcher table for the implicit RK method.
- *Be* – the Butcher table for the explicit RK method.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is NULL

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes:

For a description of the `ARKodeButcherTable` type and related functions for creating Butcher tables see *Butcher Table Data Structure*.

To set an explicit table, *Bi* must be `NULL`. This automatically calls `ARKStepSetExplicit()`. However, if the problem is posed in explicit form, i.e. $\dot{y} = f(t, y)$, then we recommend that the ERKStep time-stepper module be used instead of ARKStep.

To set an implicit table, *Be* must be `NULL`. This automatically calls `ARKStepSetImplicit()`.

If both *Bi* and *Be* are provided, this routine automatically calls `ARKStepSetImEx()`.

When only one table is provided (i.e., *Bi* or *Be* is `NULL`) then the input values of *q* and *p* are ignored and the global order of the method and embedding (if applicable) are obtained from the Butcher table structures. If both *Bi* and *Be* are non-NULL (e.g, an IMEX method is provided) then the input values of *q* and *p* are used as the order of the ARK method may be less than the orders of the individual tables. No error checking is performed to ensure that either *p* or *q* correctly describe the coefficients that were input.

Error checking is performed on *Bi* and *Be* (if non-NULL) to ensure that they specify DIRK and ERK methods, respectively.

If the inputs *Bi* or *Be* do not contain an embedding (when the corresponding explicit or implicit table is non-NULL), the user *must* call `ARKStepSetFixedStep()` to enable fixed-step mode and set the desired time step size.

int **ARKStepSetTableNum**(void* *arkode_mem*, int *itable*, int *etable*)

Indicates to use specific built-in Butcher tables for the ERK, DIRK or ARK method.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *itable* – index of the DIRK Butcher table.
- *etable* – index of the ERK Butcher table.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes:

The allowable values for both the *itable* and *etable* arguments corresponding to built-in tables may be found *Appendix: Butcher tables*.

To choose an explicit table, set *itable* to a negative value. This automatically calls `ARKStepSetExplicit()`. However, if the problem is posed in explicit form, i.e. $\dot{y} = f(t, y)$, then we recommend that the ERKStep time-stepper module be used instead of ARKStep.

To select an implicit table, set *etable* to a negative value. This automatically calls `ARKStepSetImplicit()`.

If both *itable* and *etable* are non-negative, then these should match an existing implicit/explicit pair, listed in the section *Additive Butcher tables*. This automatically calls `ARKStepSetImEx()`.

In all cases, error-checking is performed to ensure that the tables exist.

4.5.8.3 Optional inputs for time step adaptivity

The mathematical explanation of ARKode's time step adaptivity algorithm, including how each of the parameters below is used within the code, is provided in the section *Time step adaptivity*.

Optional input	Function name	Default
Set a custom time step adaptivity function	`ARKStepSetAdaptivityFn()`	internal
Choose an existing time step adaptivity method	`ARKStepSetAdaptivityMethod()`	0
Explicit stability safety factor	`ARKStepSetCFLFraction()`	0.5
Time step error bias factor	`ARKStepSetErrorBias()`	1.5
Bounds determining no change in step size	`ARKStepSetFixedStepBounds()`	1.0 1.5
Maximum step growth factor on convergence fail	`ARKStepSetMaxCFailGrowth()`	0.25
Maximum step growth factor on error test fail	`ARKStepSetMaxEFailGrowth()`	0.3
Maximum first step growth factor	`ARKStepSetMaxFirstGrowth()`	10000.0
Maximum allowed general step growth factor	`ARKStepSetMaxGrowth()`	20.0
Minimum allowed step reduction factor on error test fail	`ARKStepSetMinReduction()`	0.1
Time step safety factor	`ARKStepSetSafetyFactor()`	0.96
Error fails before MaxEFailGrowth takes effect	`ARKStepSetSmallNumEFails()`	2
Explicit stability function	`ARKStepSetStabilityFn()`	none

int **ARKStepSetAdaptivityFn**(void* *arkode_mem*, *ARKAdaptFn* *hfun*, void* *h_data*)

Sets a user-supplied time-step adaptivity function.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hfun* – name of user-supplied adaptivity function.
- *h_data* – pointer to user data passed to *hfun* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This function should focus on accuracy-based time step estimation; for stability based time steps the function `ARKStepSetStabilityFn()` should be used instead.

int **ARKStepSetAdaptivityMethod**(void* *arkode_mem*, int *imethod*, int *idefault*, int *pq*, realtype* *adapt_params*)

Specifies the method (and associated parameters) used for time step adaptivity.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *imethod* – accuracy-based adaptivity method choice ($0 \leq$ *imethod* ≤ 5): 0 is PID, 1 is PI, 2 is I, 3 is explicit Gustafsson, 4 is implicit Gustafsson, and 5 is the ImEx Gustafsson.
- *idefault* – flag denoting whether to use default adaptivity parameters (1), or that they will be supplied in the *adapt_params* argument (0).
- *pq* – flag denoting whether to use the embedding order of accuracy p (0) or the method order of accuracy q (1) within the adaptivity algorithm. p is the default.
- *adapt_params[0]* – k_1 parameter within accuracy-based adaptivity algorithms.
- *adapt_params[1]* – k_2 parameter within accuracy-based adaptivity algorithms.

- *adapt_params[2]* – k_3 parameter within accuracy-based adaptivity algorithms.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: If custom parameters are supplied, they will be checked for validity against published stability intervals. If other parameter values are desired, it is recommended to instead provide a custom function through a call to `ARKStepSetAdaptivityFn()`.

int **`ARKStepSetCFLFraction`**(void* *arkode_mem*, realtype *cfl_frac*)
Specifies the fraction of the estimated explicitly stable step to use.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *cfl_frac* – maximum allowed fraction of explicitly stable step (default is 0.5).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **`ARKStepSetErrorBias`**(void* *arkode_mem*, realtype *bias*)
Specifies the bias to be applied to the error estimates within accuracy-based adaptivity strategies.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *bias* – bias applied to error in accuracy-based time step estimation (default is 1.5).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value below 1.0 will imply a reset to the default value.

int **`ARKStepSetFixedStepBounds`**(void* *arkode_mem*, realtype *lb*, realtype *ub*)
Specifies the step growth interval in which the step size will remain unchanged.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lb* – lower bound on window to leave step size fixed (default is 1.0).
- *ub* – upper bound on window to leave step size fixed (default is 1.5).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any interval *not* containing 1.0 will imply a reset to the default values.

int **ARKStepSetMaxCFailGrowth**(void* *arkode_mem*, realtype *etacf*)
Specifies the maximum step size growth factor upon an algebraic solver convergence failure on a stage solve within a step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *etacf* – time step reduction factor on a nonlinear solver convergence failure (default is 0.25).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value outside the interval $(0, 1]$ will imply a reset to the default value.

int **ARKStepSetMaxEFailGrowth**(void* *arkode_mem*, realtype *etamxf*)
Specifies the maximum step size growth factor upon multiple successive accuracy-based error failures in the solver.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *etamxf* – time step reduction factor on multiple error fails (default is 0.3).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value outside the interval $(0, 1]$ will imply a reset to the default value.

int **ARKStepSetMaxFirstGrowth**(void* *arkode_mem*, realtype *etamx1*)
Specifies the maximum allowed growth factor in step size following the very first integration step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *etamx1* – maximum allowed growth factor after the first time step (default is 10000.0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value ≤ 1.0 will imply a reset to the default value.

int **ARKStepSetMaxGrowth**(void* *arkode_mem*, realtype *mx_growth*)
Specifies the maximum allowed growth factor in step size between consecutive steps in the integration process.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mx_growth* – maximum allowed growth factor between consecutive time steps (default is 20.0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value ≤ 1.0 will imply a reset to the default value.

int **ARKStepSetMinReduction**(void* *arkode_mem*, realtype *eta_min*)
Specifies the minimum allowed reduction factor in step size between step attempts, resulting from a temporal error failure in the integration process.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *eta_min* – minimum allowed reduction factor time step after an error test failure (default is 0.1).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value ≥ 1.0 or ≤ 0.0 will imply a reset to the default value.

int **ARKStepSetSafetyFactor**(void* *arkode_mem*, realtype *safety*)
Specifies the safety factor to be applied to the accuracy-based estimated step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *safety* – safety factor applied to accuracy-based time step (default is 0.96).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKStepSetSmallNumEFails**(void* *arkode_mem*, int *small_nef*)
Specifies the threshold for "multiple" successive error failures before the *etamxf* parameter from *ARKStepSetMaxEFailGrowth()* is applied.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *small_nef* – bound to determine 'multiple' for *etamxf* (default is 2).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKStepSetStabilityFn**(void* *arkode_mem*, *ARKExpStabFn EStab*, void* *estab_data*)
Sets the problem-dependent function to estimate a stable time step size for the explicit portion of the ODE system.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *EStab* – name of user-supplied stability function.
- *estab_data* – pointer to user data passed to *EStab* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This function should return an estimate of the absolute value of the maximum stable time step for the explicit portion of the ODE system. It is not required, since accuracy-based adaptivity may be sufficient for retaining stability, but this can be quite useful for problems where the explicit right-hand side function $f^E(t,y)$ may contain stiff terms.

4.5.8.4 Optional inputs for implicit stage solves

The mathematical explanation for the nonlinear solver strategies used by ARKStep, including how each of the parameters below is used within the code, is provided in the section *Nonlinear solver methods*.

Optional input	Function name	Default
Specify linearly implicit f^I	`ARKStepSetLinear()`	`SUNFALSE`
Specify nonlinearly implicit f^I	`ARKStepSetNonlinear()`	`SUNTRUE`
Implicit predictor method	`ARKStepSetPredictorMethod()`	0
Maximum number of nonlinear iterations	`ARKStepSetMaxNonlinIters()`	3
Coefficient in the nonlinear convergence test	`ARKStepSetNonlinConvCoef()`	0.1
Nonlinear convergence rate constant	`ARKStepSetNonlinCRDown()`	0.3
Nonlinear residual divergence ratio	`ARKStepSetNonlinRDiv()`	2.3
Maximum number of convergence failures	`ARKStepSetMaxConvFails()`	10
User-provided implicit stage predictor	`ARKStepSetStagePredictFn()`	`NULL`

int **ARKStepSetLinear**(void* *arkode_mem*, int *timedepend*)
Specifies that the implicit portion of the problem is linear.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *timedepend* – flag denoting whether the Jacobian of $f^I(t,y)$ is time-dependent (1) or not (0). Alternately, when using a matrix-free iterative linear solver this flag denotes time dependence of the preconditioner.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Tightens the linear solver tolerances and takes only a single Newton iteration. Calls *ARKStepSetDeltaGammaMax()* to enforce Jacobian recomputation when the step size ratio changes by more than 100 times the unit roundoff (since nonlinear convergence is not tested). Only applicable when used in combination with the modified or inexact Newton iteration (not the fixed-point solver).

The only SUNDIALS-provided SUNNonlinearSolver module that is compatible with the *MRIStepSetLinear()* option is the Newton solver.

int **ARKStepSetNonlinear**(void* *arkode_mem*)

Specifies that the implicit portion of the problem is nonlinear.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This is the default behavior of ARKStep, so the function is primarily useful to undo a previous call to *ARKStepSetLinear()*. Calls *ARKStepSetDeltaGammaMax()* to reset the step size ratio threshold to the default value.

int **ARKStepSetPredictorMethod**(void* *arkode_mem*, int *method*)

Specifies the method to use for predicting implicit solutions.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *method* – method choice ($0 \leq$ *method* ≤ 4):
  - 0 is the trivial predictor,
  - 1 is the maximum order (dense output) predictor,
  - 2 is the variable order predictor, that decreases the polynomial degree for more distant RK stages,
  - 3 is the cutoff order predictor, that uses the maximum order for early RK stages, and a first-order predictor for distant RK stages,
  - 4 is the bootstrap predictor, that uses a second-order predictor based on only information within the current step.
  - 5 is the minimum correction predictor, that uses all preceding stage information within the current step for prediction.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 0. If *method* is set to an undefined value, this default predictor will be used.

Options 4 and 5 are currently not supported when solving a problem involving a non-identity mass matrix. In that case, selection of *method* as 4 or 5 will instead default to the trivial predictor (*method* 0).

int **ARKStepSetMaxNonlinIters**(void* *arkode_mem*, int *maxcor*)

Specifies the maximum number of nonlinear solver iterations permitted per RK stage within each time step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *maxcor* – maximum allowed solver iterations per stage (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value or if the SUNNONLINSOL module is `NULL`
- *ARK_NLS_OP_ERR* if the SUNNONLINSOL object returned a failure flag

Notes: The default value is 3; set *maxcor* ≤ 0 to specify this default.

int **ARKStepSetNonlinConvCoef**(void* *arkode_mem*, realtype *nlscoef*)
Specifies the safety factor used within the nonlinear solver convergence test.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nlscoef* – coefficient in nonlinear solver convergence test (> 0.0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 0.1; set *nlscoef* ≤ 0 to specify this default.

int **ARKStepSetNonlinCRDown**(void* *arkode_mem*, realtype *crdown*)
Specifies the constant used in estimating the nonlinear solver convergence rate.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *crdown* – nonlinear convergence rate estimation constant (default is 0.3).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKStepSetNonlinRDiv**(void* *arkode_mem*, realtype *rdiv*)
Specifies the nonlinear correction threshold beyond which the iteration will be declared divergent.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *rdiv* – tolerance on nonlinear correction size ratio to declare divergence (default is 2.3).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ARKStepSetMaxConvFails**(void* *arkode_mem*, int *maxncf*)
Specifies the maximum number of nonlinear solver convergence failures permitted during one step, before ARKStep will return with an error.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *maxncf* – maximum allowed nonlinear solver convergence failures per step (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 10; set *maxncf* ≤ 0 to specify this default.

Upon each convergence failure, ARKStep will first call the Jacobian setup routine and try again (if a Newton method is used). If a convergence failure still occurs, the time step size is reduced by the factor *etacf* (set within `ARKStepSetMaxCFailGrowth()`).

int **ARKStepSetStagePredictFn**(void* *arkode_mem*, *ARKStagePredictFn PredictStage*)
Sets the user-supplied function to update the implicit stage predictor prior to execution of the nonlinear or linear solver algorithms that compute the implicit stage solution.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *PredictStage* – name of user-supplied predictor function. If `NULL`, then any previously-provided stage prediction function will be disabled.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

Notes: See the section *Implicit stage prediction function* for more information on this user-supplied routine.

4.5.8.5 Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to ARKStep is provided in the section *Linear solver methods*. We group the user-callable routines into four categories: general routines concerning the update frequency for matrices and/or preconditioners, optional inputs for matrix-based linear solvers, optional inputs for matrix-free linear solvers, and optional inputs for iterative linear solvers. We note that the matrix-based and matrix-free groups are mutually exclusive, whereas the "iterative" tag can apply to either case.

Optional inputs for the ARKLS linear solver interface

As discussed in the section *Updating the linear solver*, ARKode strives to reuse matrix and preconditioner data for as many solves as possible to amortize the high costs of matrix construction and factorization. To that end, ARKStep provides user-callable routines to modify this behavior. Recall that the Newton system matrices that arise within an implicit stage solve are $\mathcal{A}(t,z) \approx M - \gamma J(t,z)$, where the implicit right-hand side function has Jacobian matrix $J(t,z) = \frac{\partial f^I(t,z)}{\partial z}$.

The matrix or preconditioner for $\mathcal{A}$ can only be updated within a call to the linear solver 'setup' routine. In general, the frequency with which the linear solver setup routine is called may be controlled with the *msbp* argument to `ARKStepSetLSetupFrequency()`. When this occurs, the validity of $\mathcal{A}$ for successive time steps intimately depends on whether the corresponding γ and J inputs remain valid.

At each call to the linear solver setup routine the decision to update $\mathcal{A}$ with a new value of γ, and to reuse or reevaluate Jacobian information, depends on several factors including:

- the success or failure of previous solve attempts,
- the success or failure of the previous time step attempts,
- the change in γ from the value used when constructing $\mathcal{A}$, and
- the number of steps since Jacobian information was last evaluated.

The frequency with which to update Jacobian information can be controlled with the *msbj* argument to `ARKStepSetJacEvalFrequency()`. We note that this is only checked *within* calls to the linear solver setup routine, so values *msbj* $<$ *msbp* do not make sense. For linear-solvers with user-supplied preconditioning the above factors are used to determine whether to recommend updating the Jacobian information in the preconditioner (i.e., whether to set *jok* to `SUNFALSE` in calling the user-supplied `ARKLsPrecSetupFn()`). For matrix-based linear solvers these factors determine whether the matrix $J(t,y) = \frac{\partial f^I(t,y)}{\partial y}$ should be updated (either with an internal finite difference approximation or a call to the user-supplied `ARKLsJacFn`); if not then the previous value is reused and the system matrix $\mathcal{A}(t,y) \approx M - \gamma J(t,y)$ is recomputed using the current γ value.

Optional input	Function name	Default
Max change in step signaling new J	`ARKStepSetDeltaGammaMax()`	0.2
Linear solver setup frequency	`ARKStepSetLSetupFrequency()`	20
Jacobian / preconditioner update frequency	`ARKStepSetJacEvalFrequency()`	51

int **`ARKStepSetDeltaGammaMax`**(void* *arkode_mem*, realtype *dgmax*)

Specifies a scaled step size ratio tolerance, beyond which the linear solver setup routine will be signaled.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *dgmax* – tolerance on step size ratio change before calling linear solver setup routine (default is 0.2).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **`ARKStepSetLSetupFrequency`**(void* *arkode_mem*, int *msbp*)

Specifies the frequency of calls to the linear solver setup routine.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *msbp* – the linear solver setup frequency.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

Notes: Positive values of **msbp** specify the linear solver setup frequency. For example, an input of 1 means the setup function will be called every time step while an input of 2 means it will be called called every other time step. If **msbp** is 0, the default value of 20 will be used. A negative value forces a linear solver step at each implicit stage.

int **ARKStepSetJacEvalFrequency**(void* *arkode_mem*, long int *msbj*)

Specifies the frequency for recomputing the Jacobian or recommending a preconditioner update.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *msbj* – the Jacobian re-computation or preconditioner update frequency.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`.
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.

Notes: The Jacobian update frequency is only checked *within* calls to the linear solver setup routine, as such values of *msbj* $<$ *msbp* will result in recomputing the Jacobian every *msbp* steps. See `ARKStepSetLSetupFrequency()` for setting the linear solver steup frequency *msbp*.

Passing a value *msbj* ≤ 0 indicates to use the default value of 51.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `ARKStepSetLinearSolver()`.

Optional inputs for matrix-based **SUNLinearSolver** modules

Optional input	Function name	Default
Jacobian function	`ARKStepSetJacFn()`	`DQ`
Linear system function	`ARKStepSetLinSysFn()`	internal
Mass matrix function	`ARKStepSetMassFn()`	none
Enable or disable linear solution scaling	`ARKStepSetLinearSolutionScaling()`	on

When using matrix-based linear solver modules, the ARKLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t,y)$ or the linear system $\mathcal{A}(\sqcup,\dagger) = M(t) - \gamma J(t,y)$.

For $J(t,y)$, the ARKLS interface is packaged with a routine that can approximate J if the user has selected either dense or banded linear algebra. Alternatively, the user can supply a custom Jacobian function of type `ARKLsJacFn()` – this is *required* when the user selects other matrix formats. To specify a user-supplied Jacobian function, ARKStep provides the function `ARKStepSetJacFn()`.

Alternatively, a function of type `ARKLsLinSysFn()` can be provided to evaluate the matrix $\mathcal{A}(\sqcup,\dagger)$. By default, ARKLS uses an internal linear system function leveraging the SUNMATRIX API to form the matrix $\mathcal{A}(\sqcup,\dagger)$ by combining the matrices $M(t)$ and $J(t,y)$. To specify a user-supplied linear system function instead, ARKStep provides the function `ARKStepSetLinSysFn()`.

If the ODE system involves a non-identity mass matrix, $M \neq I$, matrix-based linear solver modules require a function to compute an approximation to the mass matrix $M(t)$. There is no default difference quotient approximation (for any matrix type), so this routine must be supplied by the user. This function must be of type `ARKLsMassFn()`, and should be set using the function `ARKStepSetMassFn()`.

In either case ($J(t,y)$ versus $\mathcal{A}(\square,\dagger)$ is supplied) the matrix information will be updated infrequently to reduce matrix construction and, with direct solvers, factorization costs. As a result the value of γ may not be current and a scaling factor is applied to the solution of the linear system to account for the lagged value of γ. See *Lagged matrix information* for more details. The function `ARKStepSetLinearSolutionScaling()` can be used to disable this scaling when necessary, e.g., when providing a custom linear solver that updates the matrix using the current γ as part of the solve.

The ARKLS interface passes the user data pointer to the Jacobian, linear system, and mass matrix functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian, linear system or mass matrix functions, without using global data in the program. The user data pointer may be specified through `ARKStepSetUserData()`.

int **ARKStepSetJacFn**(void* *arkode_mem*, *ARKLsJacFn jac*)

Specifies the Jacobian approximation routine to be used for the matrix-based solver with the ARKLS interface.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *jac* – name of user-supplied Jacobian approximation function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This routine must be called after the ARKLS linear solver interface has been initialized through a call to `ARKStepSetLinearSolver()`.

By default, ARKLS uses an internal difference quotient function for dense and band matrices. If `NULL` is passed in for *jac*, this default is used. An error will occur if no *jac* is supplied when using other matrix types.

The function type `ARKLsJacFn()` is described in the section *User-supplied functions*.

int **ARKStepSetLinSysFn**(void* *arkode_mem*, *ARKLsLinSysFn linsys*)

Specifies the linear system approximation routine to be used for the matrix-based solver with the ARKLS interface.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *linsys* – name of user-supplied linear system approximation function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This routine must be called after the ARKLS linear solver interface has been initialized through a call to `ARKStepSetLinearSolver()`.

By default, ARKLS uses an internal linear system function that leverages the SUNMATRIX API to form the system $M - \gamma J$. If `NULL` is passed in for *linsys*, this default is used.

The function type `ARKLsLinSysFn()` is described in the section *User-supplied functions*.

int **ARKStepSetMassFn**(void* *arkode_mem*, *ARKLsMassFn mass*)

Specifies the mass matrix approximation routine to be used for the matrix-based solver with the ARKLS interface.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mass* – name of user-supplied mass matrix approximation function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`
- *ARKLS_ILL_INPUT* if an argument has an illegal value

Notes: This routine must be called after the ARKLS mass matrix solver interface has been initialized through a call to `ARKStepSetMassLinearSolver()`.

Since there is no default difference quotient function for mass matrices, *mass* must be non-`NULL`.

The function type `ARKLsMassFn()` is described in the section *User-supplied functions*.

int **`ARKStepSetLinearSolutionScaling`**(void* *arkode_mem*, booleantype *onoff*)
Enables or disables scaling the linear system solution to account for a change in γ in the linear system. For more details see *Lagged matrix information*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *onoff* – flag to enable (`SUNTRUE`) or disable (`SUNFALSE`) scaling

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_ILL_INPUT* if the attached linear solver is not matrix-based

Notes: Linear solution scaling is enabled by default when a matrix-based linear solver is attached.

Optional inputs for matrix-free `SUNLinearSolver` modules

Optional input	Function name	Default
Jv functions (*jtimes* and *jtsetup*)	`ARKStepSetJacTimes()`	DQ, none
Jv DQ rhs function (*jtimesRhsFn*)	`ARKStepSetJacTimesRhsFn()`	fi
Mv functions (*mtimes* and *mtsetup*)	`ARKStepSetMassTimes()`	none, none

As described in the section *Linear solver methods*, when solving the Newton linear systems with matrix-free methods, the ARKLS interface requires a *jtimes* function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply a custom Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the ARKLS interface.

A user-defined Jacobian-vector function must be of type `ARKLsJacTimesVecFn` and can be specified through a call to `ARKStepSetJacTimes()` (see the section *User-supplied functions* for specification details). As with the user-supplied preconditioner functions, the evaluation and processing of any Jacobian-related data needed by the user's Jacobian-times-vector function is done in the optional user-supplied function of type `ARKLsJacTimesSetupFn` (see the section *User-supplied functions* for specification details). As with the preconditioner functions, a pointer to the user-defined data structure, *user_data*, specified through

ARKStepSetUserData() (or a NULL pointer otherwise) is passed to the Jacobian-times-vector setup and product functions each time they are called.

int **ARKStepSetJacTimes**(void* *arkode_mem*, *ARKLsJacTimesSetupFn* *jtsetup*, *ARKLsJacTimesVecFn* *jtimes*)

Specifies the Jacobian-times-vector setup and product functions.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *jtsetup* – user-defined Jacobian-vector setup function. Pass NULL if no setup is necessary.
- *jtimes* – user-defined Jacobian-vector product function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was NULL.
- *ARKLS_LMEM_NULL* if the linear solver memory was NULL.
- *ARKLS_ILL_INPUT* if an input has an illegal value.
- *ARKLS_SUNLS_FAIL* if an error occurred when setting up the Jacobian-vector product in the SUNLinearSolver object used by the ARKLS interface.

Notes: The default is to use an internal finite difference quotient for *jtimes* and to leave out *jtsetup*. If NULL is passed to *jtimes*, these defaults are used. A user may specify non-NULL *jtimes* and NULL *jtsetup* inputs.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to *ARKStepSetLinearSolver()*.

The function types *ARKLsJacTimesSetupFn* and *ARKLsJacTimesVecFn* are described in the section *User-supplied functions*.

When using the internal difference quotient the user may optionally supply an alternative implicit right-hand side function for use in the Jacobian-vector product approximation by calling *ARKStepSetJacTimesRhsFn()*. The alternative implicit right-hand side function should compute a suitable (and differentiable) approximation to the f^I function provided to *ARKStepCreate()*. For example, as done in *[DFWBT2010]*, the alternative function may use lagged values when evaluating a nonlinearity in f^I to avoid differencing a potentially non-differentiable factor.

int **ARKStepSetJacTimesRhsFn**(void* *arkode_mem*, *ARKRhsFn* *jtimesRhsFn*)

Specifies an alternative implicit right-hand side function for use in the internal Jacobian-vector product difference quotient approximation.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *jtimesRhsFn* – the name of the C function (of type *ARKRhsFn()*) defining the alternative right-hand side function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was NULL.
- *ARKLS_LMEM_NULL* if the linear solver memory was NULL.
- *ARKLS_ILL_INPUT* if an input has an illegal value.

Notes: The default is to use the implicit right-hand side function provided to *ARKStepCreate()* in the internal difference quotient. If the input implicit right-hand side function is NULL, the default is used.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `ARKStepSetLinearSolver()`.

Similarly, if a problem involves a non-identity mass matrix, $M \neq I$, then matrix-free solvers require a *mtimes* function to compute an approximation to the product between the mass matrix $M(t)$ and a vector v. This function must be user-supplied, since there is no default value. *mtimes* must be of type `ARKLsMassTimesVecFn()`, and can be specified through a call to the `ARKStepSetMassTimes()` routine. Similarly to the user-supplied preconditioner functions, any evaluation and processing of any mass matrix-related data needed by the user's mass-matrix-times-vector function may be done in an optional user-supplied function of type `ARKLsMassTimesSetupFn` (see the section *User-supplied functions* for specification details).

int **ARKStepSetMassTimes**(void* *arkode_mem*, *ARKLsMassTimesSetupFn* *mtsetup*, *ARKLsMassTimesVecFn* *mtimes*, void* *mtimes_data*)

Specifies the mass matrix-times-vector setup and product functions.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mtsetup* – user-defined mass matrix-vector setup function. Pass `NULL` if no setup is necessary.
- *mtimes* – user-defined mass matrix-vector product function.
- *mtimes_data* – a pointer to user data, that will be supplied to both the *mtsetup* and *mtimes* functions.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`.
- *ARKLS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.
- *ARKLS_SUNLS_FAIL* if an error occurred when setting up the mass-matrix-vector product in the `SUNLinearSolver` object used by the ARKLS interface.

Notes: There is no default finite difference quotient for *mtimes*, so if using the ARKLS mass matrix solver interface with NULL-valued M, and this routine is called with NULL-valued *mtimes*, an error will occur. A user may specify `NULL` for *mtsetup*.

This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to `ARKStepSetMassLinearSolver()`.

The function types `ARKLsMassTimesSetupFn` and `ARKLsMassTimesVecFn` are described in the section *User-supplied functions*.

Optional inputs for iterative `SUNLinearSolver` modules

Optional input	Function name	Default
Newton preconditioning functions	*ARKStepSetPreconditioner()*	NULL, NULL
Mass matrix preconditioning functions	*ARKStepSetMassPreconditioner()*	NULL, NULL
Newton linear and nonlinear tolerance ratio	*ARKStepSetEpsLin()*	0.05
Mass matrix linear and nonlinear tolerance ratio	*ARKStepSetMassEpsLin()*	0.05
Newton linear solve tolerance conversion factor	*ARKStepSetLSNormFactor()*	vector length
Mass matrix linear solve tolerance conversion factor	*ARKStepSetMassLSNormFactor()*	vector length

As described in the section *Linear solver methods*, when using an iterative linear solver the user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, *psetup* and *psolve*, that are supplied to ARKStep using either the function *ARKStepSetPreconditioner()* (for preconditioning the Newton system), or the function *ARKStepSetMassPreconditioner()* (for preconditioning the mass matrix system). The *psetup* function supplied to these routines should handle evaluation and preprocessing of any Jacobian or mass-matrix data needed by the user's preconditioner solve function, *psolve*. The user data pointer received through *ARKStepSetUserData()* (or a pointer to NULL if user data was not specified) is passed to the *psetup* and *psolve* functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program. If preconditioning is supplied for both the Newton and mass matrix linear systems, it is expected that the user will supply different *psetup* and *psolve* function for each.

Also, as described in the section *Linear iteration error control*, the ARKLS interface requires that iterative linear solvers stop when the norm of the preconditioned residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}$$

where the default $\epsilon_L = 0.05$, which may be modified by the user through the *ARKStepSetEpsLin()* function.

int **ARKStepSetPreconditioner**(void* *arkode_mem*, *ARKLsPrecSetupFn psetup*, *ARKLsPrecSolveFn psolve*)

Specifies the user-supplied preconditioner setup and solve functions.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *psetup* – user defined preconditioner setup function. Pass NULL if no setup is needed.
- *psolve* – user-defined preconditioner solve function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was NULL.
- *ARKLS_LMEM_NULL* if the linear solver memory was NULL.
- *ARKLS_ILL_INPUT* if an input has an illegal value.
- *ARKLS_SUNLS_FAIL* if an error occurred when setting up preconditioning in the SUNLinearSolver object used by the ARKLS interface.

Notes: The default is `NULL` for both arguments (i.e., no preconditioning).

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `ARKStepSetLinearSolver()`.

Both of the function types `ARKLsPrecSetupFn()` and `ARKLsPrecSolveFn()` are described in the section *User-supplied functions*.

int **ARKStepSetMassPreconditioner**(void* *arkode_mem*, *ARKLsMassPrecSetupFn psetup*, *ARKLsMassPrecSolveFn psolve*)

Specifies the mass matrix preconditioner setup and solve functions.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *psetup* – user defined preconditioner setup function. Pass `NULL` if no setup is to be done.
- *psolve* – user-defined preconditioner solve function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`.
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.
- *ARKLS_SUNLS_FAIL* if an error occurred when setting up preconditioning in the `SUNLinearSolver` object used by the ARKLS interface.

Notes: This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to `ARKStepSetMassLinearSolver()`.

The default is `NULL` for both arguments (i.e. no preconditioning).

Both of the function types `ARKLsMassPrecSetupFn()` and `ARKLsMassPrecSolveFn()` are described in the section *User-supplied functions*.

int **ARKStepSetEpsLin**(void* *arkode_mem*, realtype *eplifac*)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *eplifac* – linear convergence safety factor.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`.
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.

Notes: Passing a value *eplifac* ≤ 0 indicates to use the default value of 0.05.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `ARKStepSetLinearSolver()`.

int **ARKStepSetMassEpsLin**(void* *arkode_mem*, realtype *eplifac*)

Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the mass matrix linear iteration.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *eplifac* – linear convergence safety factor.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`.
- *ARKLS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.

Notes: This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to *ARKStepSetMassLinearSolver()*.

Passing a value *eplifac* ≤ 0 indicates to use the default value of 0.05.

int **ARKStepSetLSNormFactor**(void* *arkode_mem*, realtype *nrmfac*)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves e.g., `tol_L2 = fac * tol_WRMS`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nrmfac* – the norm conversion factor. If *nrmfac* is:

  > 0 then the provided value is used.

  $= 0$ then the conversion factor is computed using the vector length i.e., `nrmfac = sqrt(N_VGetLength(y))` (*default*).

  < 0 then the conversion factor is computed using the vector dot product i.e., `nrmfac = sqrt(N_VDotProd(v,v))` where all the entries of `v` are one.

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`.

Notes: This function must be called *after* the ARKLS system solver interface has been initialized through a call to *ARKStepSetLinearSolver()*.

Prior to the introduction of *N_VGetLength()* in SUNDIALS v5.0.0 the value of `nrmfac` was computed using the vector dot product i.e., the `nrmfac < 0` case.

int **ARKStepSetMassLSNormFactor**(void* *arkode_mem*, realtype *nrmfac*)

Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for mass matrix linear system solves e.g., `tol_L2 = fac * tol_WRMS`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nrmfac* – the norm conversion factor. If *nrmfac* is:

  > 0 then the provided value is used.

$= 0$ then the conversion factor is computed using the vector length i.e., `nrmfac = sqrt(N_VGetLength(y))` (*default*).

< 0 then the conversion factor is computed using the vector dot product i.e., `nrmfac = sqrt(N_VDotProd(v,v))` where all the entries of `v` are one.

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`.

Notes: This function must be called *after* the ARKLS mass matrix solver interface has been initialized through a call to *ARKStepSetMassLinearSolver()*.

Prior to the introduction of *N_VGetLength()* in SUNDIALS v5.0.0 (ARKODE v4.0.0) the value of `nrmfac` was computed using the vector dot product i.e., the `nrmfac < 0` case.

4.5.8.6 Rootfinding optional input functions

The following functions can be called to set optional inputs to control the rootfinding algorithm, the mathematics of which are described in the section *Rootfinding*.

Optional input	Function name	Default
Direction of zero-crossings to monitor	*ARKStepSetRootDirection()*	both
Disable inactive root warnings	*ARKStepSetNoInactiveRootWarn()*	enabled

int **ARKStepSetRootDirection**(void* *arkode_mem*, int* *rootdir*)
Specifies the direction of zero-crossings to be located and returned.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *rootdir* – state array of length *nrtfn*, the number of root functions g_i (the value of *nrtfn* was supplied in the call to *ARKStepRootInit()*). If `rootdir[i] == 0` then crossing in either direction for g_i should be reported. A value of +1 or -1 indicates that the solver should report only zero-crossings where g_i is increasing or decreasing, respectively.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default behavior is to monitor for both zero-crossing directions.

int **ARKStepSetNoInactiveRootWarn**(void* *arkode_mem*)
Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

Notes: ARKStep will not report the initial conditions as a possible zero-crossing (assuming that one or more components g_i are zero at the initial time). However, if it appears that some g_i is identically zero at the initial time (i.e., g_i is zero at the initial time *and* after the first step), ARKStep will issue a warning which can be disabled with this optional input function.

4.5.9 Interpolated output function

An optional function `ARKStepGetDky()` is available to obtain additional values of solution-related quantities. This function should only be called after a successful return from `ARKStepEvolve()`, as it provides interpolated values either of y or of its derivatives (up to the 5th derivative) interpolated to any value of t in the last internal step taken by `ARKStepEvolve()`. Internally, this *dense output* algorithm is identical to the algorithm used for the maximum order implicit predictors, described in the section *Maximum order predictor*, except that derivatives of the polynomial model may be evaluated upon request.

int **ARKStepGetDky**(void* *arkode_mem*, realtype *t*, int *k*, N_Vector *dky*)

Computes the *k*-th derivative of the function y at the time *t*, i.e. $\frac{d^{(k)}}{dt^{(k)}}y(t)$, for values of the independent variable satisfying $t_n - h_n \le t \le t_n$, with t_n as current internal time reached, and h_n is the last internal step size successfully used by the solver. This routine uses an interpolating polynomial of degree *min(degree, 5)*, where *degree* is the argument provided to `ARKStepSetInterpolantDegree()`. The user may request *k* in the range {0,...,*min(degree, kmax)*} where *kmax* depends on the choice of interpolation module. For Hermite interpolants *kmax* = *5* and for Lagrange interpolants *kmax* = *3*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *t* – the value of the independent variable at which the derivative is to be evaluated.
- *k* – the derivative order requested.
- *dky* – output vector (must be allocated by the user).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_BAD_K* if *k* is not in the range {0,...,*min(degree, kmax)*}.
- *ARK_BAD_T* if *t* is not in the interval $[t_n - h_n, t_n]$
- *ARK_BAD_DKY* if the *dky* vector was `NULL`
- *ARK_MEM_NULL* if the ARKStep memory is `NULL`

Notes: It is only legal to call this function after a successful return from `ARKStepEvolve()`.

A user may access the values t_n and h_n via the functions `ARKStepGetCurrentTime()` and `ARKStepGetLastStep()`, respectively.

4.5.10 Optional output functions

ARKStep provides an extensive set of functions that can be used to obtain solver performance information. We organize these into groups:

1. SUNDIALS version information accessor routines are in the subsection *SUNDIALS version information*,
2. General ARKStep output routines are in the subsection *Main solver optional output functions*,
3. ARKStep implicit solver output routines are in the subsection *Implicit solver optional output functions*,
4. Output routines regarding root-finding results are in the subsection *Rootfinding optional output functions*,

5. Linear solver output routines are in the subsection *Linear solver interface optional output functions* and
6. General usability routines (e.g. to print the current ARKStep parameters, or output the current Butcher table(s)) are in the subsection *General usability functions*.

Following each table, we elaborate on each function.

Some of the optional outputs, especially the various counters, can be very useful in determining the efficiency of various methods inside ARKStep. For example:

- The counters *nsteps*, *nfe_evals* and *nfi_evals* provide a rough measure of the overall cost of a given run, and can be compared between runs with different solver options to suggest which set of options is the most efficient.
- The ratio *nniters/nsteps* measures the performance of the nonlinear iteration in solving the nonlinear systems at each stage, providing a measure of the degree of nonlinearity in the problem. Typical values of this for a Newton solver on a general problem range from 1.1 to 1.8.
- When using a Newton nonlinear solver, the ratio *njevals/nniters* (in the case of a direct linear solver), and the ratio *npevals/nniters* (in the case of an iterative linear solver) can measure the overall degree of nonlinearity in the problem, since these are updated infrequently, unless the Newton method convergence slows.
- When using a Newton nonlinear solver, the ratio *njevals/nniters* (when using a direct linear solver), and the ratio *nliters/nniters* (when using an iterative linear solver) can indicate the quality of the approximate Jacobian or preconditioner being used. For example, if this ratio is larger for a user-supplied Jacobian or Jacobian-vector product routine than for the difference-quotient routine, it can indicate that the user-supplied Jacobian is inaccurate.
- The ratio *expsteps/accsteps* can measure the quality of the ImEx splitting used, since a higher-quality splitting will be dominated by accuracy-limited steps.
- The ratio *nsteps/step_attempts* can measure the quality of the time step adaptivity algorithm, since a poor algorithm will result in more failed steps, and hence a lower ratio.

It is therefore recommended that users retrieve and output these statistics following each run, and take some time to investigate alternate solver options that will be more optimal for their particular problem of interest.

4.5.10.1 SUNDIALS version information

The following functions provide a way to get SUNDIALS version information at runtime.

int **SUNDIALSGetVersion**(char **version*, int *len*)
This routine fills a string with SUNDIALS version information.

Arguments:

- *version* – character array to hold the SUNDIALS version information.
- *len* – allocated length of the *version* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS version

Notes: An array of 25 characters should be sufficient to hold the version information.

int **SUNDIALSGetVersionNumber**(int **major*, int **minor*, int **patch*, char **label*, int *len*)
This routine sets integers for the SUNDIALS major, minor, and patch release numbers and fills a string with the release label if applicable.

Arguments:

- *major* – SUNDIALS release major version number.
- *minor* – SUNDIALS release minor version number.
- *patch* – SUNDIALS release patch version number.
- *label* – string to hold the SUNDIALS release label.
- *len* – allocated length of the *label* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS label

Notes: An array of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to *label*.

4.5.10.2 Main solver optional output functions

Optional output	Function name
Size of ARKStep real and integer workspaces	`ARKStepGetWorkSpace()`
Cumulative number of internal steps	`ARKStepGetNumSteps()`
Actual initial time step size used	`ARKStepGetActualInitStep()`
Step size used for the last successful step	`ARKStepGetLastStep()`
Step size to be attempted on the next step	`ARKStepGetCurrentStep()`
Current internal time reached by the solver	`ARKStepGetCurrentTime()`
Current internal solution reached by the solver	`ARKStepGetCurrentState()`
Current γ value used by the solver	`ARKStepGetCurrentGamma()`
Suggested factor for tolerance scaling	`ARKStepGetTolScaleFactor()`
Error weight vector for state variables	`ARKStepGetErrWeights()`
Residual weight vector	`ARKStepGetResWeights()`
Single accessor to many statistics at once	`ARKStepGetStepStats()`
Name of constant associated with a return flag	`ARKStepGetReturnFlagName()`
No. of explicit stability-limited steps	`ARKStepGetNumExpSteps()`
No. of accuracy-limited steps	`ARKStepGetNumAccSteps()`
No. of attempted steps	`ARKStepGetNumStepAttempts()`
No. of calls to *fe* and *fi* functions	`ARKStepGetNumRhsEvals()`
No. of local error test failures that have occurred	`ARKStepGetNumErrTestFails()`
Current ERK and DIRK Butcher tables	`ARKStepGetCurrentButcherTables()`
Estimated local truncation error vector	`ARKStepGetEstLocalErrors()`
Single accessor to many statistics at once	`ARKStepGetTimestepperStats()`
Number of constraint test failures	`ARKStepGetNumConstrFails()`

int **`ARKStepGetWorkSpace`**(void* *arkode_mem*, long int* *lenrw*, long int* *leniw*)
Returns the ARKStep real and integer workspace sizes.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lenrw* – the number of `realtype` values in the ARKStep workspace.
- *leniw* – the number of integer values in the ARKStep workspace.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetNumSteps**(void* *arkode_mem*, long int* *nsteps*)
Returns the cumulative number of internal steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nsteps* – number of steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetActualInitStep**(void* *arkode_mem*, realtype* *hinused*)
Returns the value of the integration step size used on the first step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hinused* – actual value of initial step size.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: Even if the value of the initial integration step was specified by the user through a call to `ARKStepSetInitStep()`, this value may have been changed by ARKStep to ensure that the step size fell within the prescribed bounds ($h_{min} \leq h_0 \leq h_{max}$), or to satisfy the local error test condition, or to ensure convergence of the nonlinear solver.

int **ARKStepGetLastStep**(void* *arkode_mem*, realtype* *hlast*)
Returns the integration step size taken on the last successful internal step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hlast* – step size taken on the last internal step.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetCurrentStep**(void* *arkode_mem*, realtype* *hcur*)
Returns the integration step size to be attempted on the next internal step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *hcur* – step size to be attempted on the next internal step.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetCurrentTime**(void* *arkode_mem*, realtype* *tcur*)
Returns the current internal time reached by the solver.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *tcur* – current internal time reached.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetCurrentState**(void **arkode_mem*, N_Vector **ycur*)
Returns the current internal solution reached by the solver.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ycur* – current internal solution.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: Users should exercise extreme caution when using this function, as altering values of *ycur* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

int **ARKStepGetCurrentGamma**(void **arkode_mem*, realtype **gamma*)
Returns the current internal value of γ used in the implicit solver Newton matrix (see equation (2.28)).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *gamma* – current step size scaling factor in the Newton system.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetTolScaleFactor**(void* *arkode_mem*, realtype* *tolsfac*)
Returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *tolsfac* – suggested scaling factor for user-supplied tolerances.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetErrWeights**(void* *arkode_mem*, N_Vector *eweight*)
Returns the current error weight vector.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *eweight* – solution error weights at the current time.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The user must allocate space for *eweight*, that will be filled in by this function.

int **ARKStepGetResWeights**(void* *arkode_mem*, N_Vector *rweight*)
Returns the current residual weight vector.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *rweight* – residual error weights at the current time.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The user must allocate space for *rweight*, that will be filled in by this function.

int **ARKStepGetStepStats**(void* *arkode_mem*, long int* *nsteps*, realtype* *hinused*, realtype* *hlast*, realtype* *hcur*, realtype* *tcur*)
Returns many of the most useful optional outputs in a single call.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nsteps* – number of steps taken in the solver.
- *hinused* – actual value of initial step size.
- *hlast* – step size taken on the last internal step.
- *hcur* – step size to be attempted on the next internal step.
- *tcur* – current internal time reached.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

char ***ARKStepGetReturnFlagName**(long int *flag*)
Returns the name of the ARKStep constant corresponding to *flag*.

Arguments:

- *flag* – a return flag from an ARKStep function.

Return value: The return value is a string containing the name of the corresponding constant.

int **ARKStepGetNumExpSteps**(void* *arkode_mem*, long int* *expsteps*)
Returns the cumulative number of stability-limited steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetNumAccSteps**(void* *arkode_mem*, long int* *accsteps*)
Returns the cumulative number of accuracy-limited steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *accsteps* – number of accuracy-limited steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetNumStepAttempts**(void* *arkode_mem*, long int* *step_attempts*)
Returns the cumulative number of steps attempted by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *step_attempts* – number of steps attempted by solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetNumRhsEvals**(void* *arkode_mem*, long int* *nfe_evals*, long int* *nfi_evals*)
Returns the number of calls to the user's right-hand side functions, f^E and f^I (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nfe_evals* – number of calls to the user's $f^E(t, y)$ function.
- *nfi_evals* – number of calls to the user's $f^I(t, y)$ function.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The *nfi_evals* value does not account for calls made to f^I by a linear solver or preconditioner module.

int **ARKStepGetNumErrTestFails**(void* *arkode_mem*, long int* *netfails*)
Returns the number of local error test failures that have occurred (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *netfails* – number of error test failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetCurrentButcherTables**(void* *arkode_mem*, *ARKodeButcherTable* *Bi*, ARKodeButcherTable *Be*)
Returns the explicit and implicit Butcher tables currently in use by the solver.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *Bi* – pointer to implicit Butcher table structure.
- *Be* – pointer to explicit Butcher table structure.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The *ARKodeButcherTable* data structure is defined as a pointer to the following C structure:

```
typedef struct ARKStepButcherTableMem {

  int q;           /* method order of accuracy       */
  int p;           /* embedding order of accuracy    */
  int stages;      /* number of stages               */
  realtype **A;    /* Butcher table coefficients     */
  realtype *c;     /* canopy node coefficients       */
  realtype *b;     /* root node coefficients         */
  realtype *d;     /* embedding coefficients         */

} *ARKStepButcherTable;
```

For more details see *Butcher Table Data Structure*.

int **ARKStepGetEstLocalErrors**(void* *arkode_mem*, N_Vector *ele*)
Returns the vector of estimated local truncation errors for the current step.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ele* – vector of estimated local truncation errors.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The user must allocate space for *ele*, that will be filled in by this function.

The values returned in *ele* are valid only after a successful call to *ARKStepEvolve()* (i.e. it returned a non-negative value).

The *ele* vector, together with the *eweight* vector from *ARKStepGetErrWeights()*, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the WRMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

int **ARKStepGetTimestepperStats**(void* *arkode_mem*, long int* *expsteps*, long int* *accsteps*, long int* *step_attempts*, long int* *nfe_evals*, long int* *nfi_evals*, long int* *nlinsetups*, long int* *netfails*)
Returns many of the most useful time-stepper statistics in a single call.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.

- *accsteps* – number of accuracy-limited steps taken in the solver.
- *step_attempts* – number of steps attempted by the solver.
- *nfe_evals* – number of calls to the user's $f^E(t,y)$ function.
- *nfi_evals* – number of calls to the user's $f^I(t,y)$ function.
- *nlinsetups* – number of linear solver setup calls made.
- *netfails* – number of error test failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetNumConstrFails**(void* *arkode_mem*, long int* *nconstrfails*)
Returns the cumulative number of constraint test failures (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nconstrfails* – number of constraint test failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

4.5.10.3 Implicit solver optional output functions

Optional output	Function name
No. of calls to linear solver setup function	*ARKStepGetNumLinSolvSetups()*
No. of nonlinear solver iterations	*ARKStepGetNumNonlinSolvIters()*
No. of nonlinear solver convergence failures	*ARKStepGetNumNonlinSolvConvFails()*
Single accessor to all nonlinear solver statistics	*ARKStepGetNonlinSolvStats()*

int **ARKStepGetNumLinSolvSetups**(void* *arkode_mem*, long int* *nlinsetups*)
Returns the number of calls made to the linear solver's setup routine (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nlinsetups* – number of linear solver setup calls made.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: This is only accumulated for the 'life' of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumNonlinSolvIters**(void* *arkode_mem*, long int* *nniters*)
Returns the number of nonlinear solver iterations performed (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

- *nniters* – number of nonlinear iterations performed.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NLS_OP_ERR* if the SUNNONLINSOL object returned a failure flag

Notes: This is only accumulated for the 'life' of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumNonlinSolvConvFails**(void* *arkode_mem*, long int* *nncfails*)
Returns the number of nonlinear solver convergence failures that have occurred (so far).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nncfails* – number of nonlinear convergence failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: This is only accumulated for the 'life' of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNonlinSolvStats**(void* *arkode_mem*, long int* *nniters*, long int* *nncfails*)
Returns all of the nonlinear solver statistics in a single call.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nniters* – number of nonlinear iterations performed.
- *nncfails* – number of nonlinear convergence failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NLS_OP_ERR* if the SUNNONLINSOL object returned a failure flag

Notes: These are only accumulated for the 'life' of the nonlinear solver object; the counters are reset whenever a new nonlinear solver module is 'attached' to ARKStep, or when ARKStep is resized.

4.5.10.4 Rootfinding optional output functions

Optional output	Function name
Array showing roots found	*ARKStepGetRootInfo()*
No. of calls to user root function	*ARKStepGetNumGEvals()*

int **ARKStepGetRootInfo**(void* *arkode_mem*, int* *rootsfound*)
Returns an array showing which functions were found to have a root.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.

- *rootsfound* – array of length *nrtfn* with the indices of the user functions g_i found to have a root (the value of *nrtfn* was supplied in the call to `ARKStepRootInit()`). For $i = 0 \ldots$ *nrtfn*-1, `rootsfound[i]` is nonzero if g_i has a root, and 0 if not.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The user must allocate space for *rootsfound* prior to calling this function.

For the components of g_i for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of +1 indicates that g_i is increasing, while a value of -1 indicates a decreasing g_i.

int **ARKStepGetNumGEvals**(void* *arkode_mem*, long int* *ngevals*)
Returns the cumulative number of calls made to the user's root function g.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ngevals* – number of calls made to g so far.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

4.5.10.5 Linear solver interface optional output functions

The following optional outputs are available from the ARKLS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the mass matrix routine, number of calls to the implicit right-hand side routine for finite-difference Jacobian approximation or Jacobian-vector product approximation, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector setup and product routines, number of calls to the mass-matrix-vector setup and product routines, and last return value from an ARKLS function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) or MLS (for Mass Linear Solver) has been added here (e.g. *lenrwLS*).

Optional output	Function name
Size of real and integer workspaces	`ARKStepGetLinWorkSpace()`
No. of Jacobian evaluations	`ARKStepGetNumJacEvals()`
No. of preconditioner evaluations	`ARKStepGetNumPrecEvals()`
No. of preconditioner solves	`ARKStepGetNumPrecSolves()`
No. of linear iterations	`ARKStepGetNumLinIters()`
No. of linear convergence failures	`ARKStepGetNumLinConvFails()`
No. of Jacobian-vector setup evaluations	`ARKStepGetNumJTSetupEvals()`
No. of Jacobian-vector product evaluations	`ARKStepGetNumJtimesEvals()`
No. of *fi* calls for finite diff. J or Jv evals.	`ARKStepGetNumLinRhsEvals()`
Last return from a linear solver function	`ARKStepGetLastLinFlag()`
Name of constant associated with a return flag	`ARKStepGetLinReturnFlagName()`
Size of real and integer mass matrix solver workspaces	`ARKStepGetMassWorkSpace()`
No. of mass matrix solver setups (incl. M evals.)	`ARKStepGetNumMassSetups()`
No. of mass matrix multiply setups	`ARKStepGetNumMassMultSetups()`
No. of mass matrix multiplies	`ARKStepGetNumMassMult()`
No. of mass matrix solves	`ARKStepGetNumMassSolves()`
No. of mass matrix preconditioner evaluations	`ARKStepGetNumMassPrecEvals()`
No. of mass matrix preconditioner solves	`ARKStepGetNumMassPrecSolves()`
No. of mass matrix linear iterations	`ARKStepGetNumMassIters()`
No. of mass matrix solver convergence failures	`ARKStepGetNumMassConvFails()`
No. of mass-matrix-vector setup evaluations	`ARKStepGetNumMTSetups()`
Last return from a mass matrix solver function	`ARKStepGetLastMassFlag()`

int **`ARKStepGetLinWorkSpace`**(void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)
Returns the real and integer workspace used by the ARKLS linear solver interface.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lenrwLS* – the number of `realtype` values in the ARKLS workspace.
- *leniwLS* – the number of integer values in the ARKLS workspace.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template Jacobian matrix allocated by the user outside of ARKLS is not included in this report.

In a parallel setting, the above values are global (i.e. summed over all processors).

int **`ARKStepGetNumJacEvals`**(void* *arkode_mem*, long int* *njevals*)
Returns the number of Jacobian evaluations.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *njevals* – number of Jacobian evaluations.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumPrecEvals**(void* *arkode_mem*, long int* *npevals*)
Returns the total number of preconditioner evaluations, i.e. the number of calls made to *psetup* with `jok` = `SUNFALSE` and that returned `*jcurPtr` = `SUNTRUE`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *npevals* – the current number of calls to *psetup*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumPrecSolves**(void* *arkode_mem*, long int* *npsolves*)
Returns the number of calls made to the preconditioner solve function, *psolve*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *npsolves* – the number of calls to *psolve*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumLinIters**(void* *arkode_mem*, long int* *nliters*)
Returns the cumulative number of linear iterations.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nliters* – the current number of linear iterations.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumLinConvFails**(void* *arkode_mem*, long int* *nlcfails*)
Returns the cumulative number of linear convergence failures.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nlcfails* – the current number of linear convergence failures.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumJTSetupEvals**(void* *arkode_mem*, long int* *njtsetup*)
Returns the cumulative number of calls made to the user-supplied Jacobian-vector setup function, *jtsetup*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *njtsetup* – the current number of calls to *jtsetup*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumJtimesEvals**(void* *arkode_mem*, long int* *njvevals*)
Returns the cumulative number of calls made to the Jacobian-vector product function, *jtimes*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *njvevals* – the current number of calls to *jtimes*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumLinRhsEvals**(void* *arkode_mem*, long int* *nfevalsLS*)
Returns the number of calls to the user-supplied implicit right-hand side function f^I for finite difference Jacobian or Jacobian-vector product approximation.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nfevalsLS* – the number of calls to the user implicit right-hand side function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The value *nfevalsLS* is incremented only if the default internal difference quotient function is used.

This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetLastLinFlag**(void* *arkode_mem*, long int* *lsflag*)
Returns the last return value from an ARKLS routine.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lsflag* – the value of the last return flag from an ARKLS function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: If the ARKLS setup function failed when using the `SUNLINSOL_DENSE` or `SUNLINSOL_BAND` modules, then the value of *lsflag* is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix. For all other failures, *lsflag* is negative.

Otherwise, if the ARKLS setup function failed (*ARKStepEvolve()* returned *ARK_LSETUP_FAIL*), then *lsflag* will be *SUNLS_PSET_FAIL_UNREC*, *SUNLS_ASET_FAIL_UNREC* or *SUNLS_PACKAGE_FAIL_UNREC*.

If the ARKLS solve function failed (*ARKStepEvolve()* returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from the `SUNLinearSolver` object, which will be one of: *SUNLS_MEM_NULL*, indicating that the `SUNLinearSolver` memory is `NULL`; *SUNLS_ATIMES_NULL*, indicating that a matrix-free iterative solver was provided, but is missing a routine for the matrix-vector product approximation, *SUNLS_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the Jv function; *SUNLS_PSOLVE_NULL*, indicating that an iterative linear solver was configured to use preconditioning, but no preconditioner solve routine was provided, *SUNLS_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function failed unrecoverably; *SUNLS_GS_FAIL*, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); *SUNLS_QRSOL_FAIL*, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or *SUNLS_PACKAGE_FAIL_UNREC*, indicating an unrecoverable failure in an external iterative linear solver package.

char ***ARKStepGetLinReturnFlagName**(long int *lsflag*)
Returns the name of the ARKLS constant corresponding to *lsflag*.

Arguments:

- *lsflag* – a return flag from an ARKLS function.

Return value: The return value is a string containing the name of the corresponding constant. If using the `SUNLINSOL_DENSE` or `SUNLINSOL_BAND` modules, then if $1 \le$ *lsflag* $\le n$ (LU factorization failed), this routine returns "NONE".

int **ARKStepGetMassWorkSpace**(void* *arkode_mem*, long int* *lenrwMLS*, long int* *leniwMLS*)
Returns the real and integer workspace used by the ARKLS mass matrix linear solver interface.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lenrwMLS* – the number of `realtype` values in the ARKLS mass solver workspace.
- *leniwMLS* – the number of integer values in the ARKLS mass solver workspace.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template mass matrix allocated by the user outside of ARKLS is not included in this report.

In a parallel setting, the above values are global (i.e. summed over all processors).

int **`ARKStepGetNumMassSetups`**(void* *arkode_mem*, long int* *nmsetups*)
Returns the number of calls made to the ARKLS mass matrix solver 'setup' routine; these include all calls to the user-supplied mass-matrix constructor function.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmsetups* – number of calls to the mass matrix solver setup routine.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **`ARKStepGetNumMassMultSetups`**(void* *arkode_mem*, long int* *nmvsetups*)
Returns the number of calls made to the ARKLS mass matrix 'matvec setup' (matrix-based solvers) routine.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmvsetups* – number of calls to the mass matrix matrix-times-vector setup routine.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **`ARKStepGetNumMassMult`**(void* *arkode_mem*, long int* *nmmults*)
Returns the number of calls made to the ARKLS mass matrix 'matvec' routine (matrix-based solvers) or the user-supplied *mtimes* routine (matris-free solvers).

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmmults* – number of calls to the mass matrix solver matrix-times-vector routine.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumMassSolves**(void* *arkode_mem*, long int* *nmsolves*)
Returns the number of calls made to the ARKLS mass matrix solver 'solve' routine.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmsolves* – number of calls to the mass matrix solver solve routine.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumMassPrecEvals**(void* *arkode_mem*, long int* *nmpevals*)
Returns the total number of mass matrix preconditioner evaluations, i.e. the number of calls made to *psetup*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmpevals* – the current number of calls to *psetup*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumMassPrecSolves**(void* *arkode_mem*, long int* *nmpsolves*)
Returns the number of calls made to the mass matrix preconditioner solve function, *psolve*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmpsolves* – the number of calls to *psolve*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`

- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumMassIters**(void* *arkode_mem*, long int* *nmiters*)
Returns the cumulative number of mass matrix solver iterations.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmiters* – the current number of mass matrix solver linear iterations.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumMassConvFails**(void* *arkode_mem*, long int* *nmcfails*)
Returns the cumulative number of mass matrix solver convergence failures.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmcfails* – the current number of mass matrix solver convergence failures.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetNumMTSetups**(void* *arkode_mem*, long int* *nmtsetup*)
Returns the cumulative number of calls made to the user-supplied mass-matrix-vector product setup function, *mtsetup*.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nmtsetup* – the current number of calls to *mtsetup*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new mass-matrix linear solver module is 'attached' to ARKStep, or when ARKStep is resized.

int **ARKStepGetLastMassFlag**(void* *arkode_mem*, long int* *mlsflag*)
Returns the last return value from an ARKLS mass matrix interface routine.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mlsflag* – the value of the last return flag from an ARKLS mass matrix solver interface function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the ARKStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The values of *msflag* for each of the various solvers will match those described above for the function `ARKStepGetLastLSFlag()`.

4.5.10.6 General usability functions

The following optional routines may be called by a user to inquire about existing solver parameters or write the current Butcher table(s). While neither of these would typically be called during the course of solving an initial value problem, they may be useful for users wishing to better understand ARKStep and/or specific Runge-Kutta methods.

Optional routine	Function name
Output all ARKStep solver parameters	*ARKStepWriteParameters()*
Output the current Butcher table(s)	*ARKStepWriteButcher()*

int **ARKStepWriteParameters**(void* *arkode_mem*, FILE **fp*)
Outputs all ARKStep solver parameters to the provided file pointer.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *fp* – pointer to use for printing the solver parameters.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The *fp* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

When run in parallel, only one process should set a non-NULL value for this pointer, since parameters for all processes would be identical.

int **ARKStepWriteButcher**(void* *arkode_mem*, FILE **fp*)
Outputs the current Butcher table(s) to the provided file pointer.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *fp* – pointer to use for printing the Butcher table(s).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Notes: The *fp* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

If ARKStep is currently configured to run in purely explicit or purely implicit mode, this will output a single Butcher table; if configured to run an ImEx method then both tables will be output.

When run in parallel, only one process should set a non-NULL value for this pointer, since tables for all processes would be identical.

4.5.11 ARKStep re-initialization function

To reinitialize the ARKStep module for the solution of a new problem, where a prior call to *ARKStepCreate()* has been made, the user must call the function *ARKStepReInit()*. The new problem must have the same size as the previous one. This routine retains the current settings for all ARKstep module options and performs the same input checking and initializations that are done in *ARKStepCreate()*, but it performs no memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration. Following a successful call to *ARKStepReInit()*, call *ARKStepEvolve()* again for the solution of the new problem.

The use of *ARKStepReInit()* requires that the number of Runge Kutta stages, denoted by *s*, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the method order *q* and the problem type (explicit, implicit, ImEx) are left unchanged.

When using the ARKStep time-stepping module, if there are changes to the linear solver specifications, the user should make the appropriate calls to either the linear solver objects themselves, or to the ARKLS interface routines, as described in the section *Linear solver interface functions*. Otherwise, all solver inputs set previously remain in effect.

One important use of the *ARKStepReInit()* function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to *ARKStepReInit()*. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ARKStepReInit**(void* *arkode_mem*, *ARKRhsFn fe*, *ARKRhsFn fi*, realtype *t0*, N_Vector *y0*)
Provides required problem specifications and re-initializes the ARKStep time-stepper module.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *fe* – the name of the C function (of type *ARKRhsFn()*) defining the explicit portion of the right-hand side function in $M\,\dot{y} = f^E(t,y) + f^I(t,y)$.
- *fi* – the name of the C function (of type *ARKRhsFn()*) defining the implicit portion of the right-hand side function in $M\,\dot{y} = f^E(t,y) + f^I(t,y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: All previously set options are retained but may be updated by calling the appropriate "Set" functions.

If an error occurred, `ARKStepReInit()` also sends an error message to the error handler function.

4.5.12 ARKStep reset function

To reset the ARKStep module to a particular independent variable value and dependent variable vector for the continued solution of a problem, where a prior call to `ARKStepCreate()` has been made, the user must call the function `ARKStepReset()`. Like `ARKStepReInit()` this routine retains the current settings for all ARKStep module options and performs no memory allocations but, unlike `ARKStepReInit()`, this routine performs only a *subset* of the input checking and initializations that are done in `ARKStepCreate()`. In particular this routine retains all internal counter values and the step size/error history and does not reinitialize the linear and/or nonlinear solver but it does indicate that a linear solver setup is necessary in the next step. Following a successful call to `ARKStepReset()`, call `ARKStepEvolve()` again to continue solving the problem. By default the next call to `ARKStepEvolve()` will use the step size computed by ARKStep prior to calling `ARKStepReset()`. To set a different step size or have ARKStep estimate a new step size use `ARKStepSetInitStep()`.

One important use of the `ARKStepReset()` function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to `ARKStepReset()`. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ARKStepReset**(void* *arkode_mem*, realtype *tR*, N_Vector *yR*)

Resets the current ARKStep time-stepper module state to the provided independent variable value and dependent variable vector.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *tR* – the value of the independent variable t.
- *yR* – the value of the dependent variable vector $y(t_R)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: By default the next call to `ARKStepEvolve()` will use the step size computed by ARKStep prior to calling `ARKStepReset()`. To set a different step size or have ARKStep estimate a new step size use `ARKStepSetInitStep()`.

All previously set options are retained but may be updated by calling the appropriate "Set" functions.

If an error occurred, `ARKStepReset()` also sends an error message to the error handler function.

4.5.13 ARKStep system resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatially-adaptive PDE simulations under a method-of-lines approach), the ARKStep integrator may be "resized" between integration steps, through calls to the `ARKStepResize()` function. This function modifies ARKStep's internal memory structures to use the new problem size, without destruction of the temporal adaptivity heuristics. It is assumed that the dynamical time scales before and after the vector resize will be comparable, so that all time-stepping heuristics prior to calling `ARKStepResize()` remain valid after the call. If instead the dynamics should be recomputed from scratch, the ARKStep memory structure should be deleted with a call to `ARKStepFree()`, and recreated with a calls to `ARKStepCreate()`.

To aid in the vector resize operation, the user can supply a vector resize function that will take as input a vector with the previous size, and transform it in-place to return a corresponding vector of the new size. If this function (of type `ARKVecResizeFn()`) is not supplied (i.e. is set to `NULL`), then all existing vectors internal to ARKStep will be destroyed and re-cloned from the new input vector.

In the case that the dynamical time scale should be modified slightly from the previous time scale, an input *hscale* is allowed, that will rescale the upcoming time step by the specified factor. If a value *hscale* $\leq$ 0 is specified, the default of 1.0 will be used.

int **ARKStepResize**(void* *arkode_mem*, N_Vector *ynew*, realtype *hscale*, realtype *t0*, *ARKVecResizeFn resize*, void* *resize_data*)

Re-initializes ARKStep with a different state vector but with comparable dynamical time scale.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ynew* – the newly-sized solution vector, holding the current dependent variable values $y(t_0)$.
- *hscale* – the desired scaling factor for the dynamical time scale (i.e. the next step will be of size *h*hscale*).
- *t0* – the current value of the independent variable t_0 (this must be consistent with *ynew*).
- *resize* – the user-supplied vector resize function (of type `ARKVecResizeFn()`.
- *resize_data* – the user-supplied data structure to be passed to *resize* when modifying internal ARKStep vectors.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: If an error occurred, `ARKStepResize()` also sends an error message to the error handler function.

If inequality constraint checking is enabled a call to `ARKStepResize()` will disable constraint checking. A call to `ARKStepSetConstraints()` is required to re-enable constraint checking.

4.5.13.1 Resizing the linear solver

When using any of the SUNDIALS-provided linear solver modules, the linear solver memory structures must also be resized. At present, none of these include a solver-specific 'resize' function, so the linear solver memory must be destroyed and re-allocated **following** each call to `ARKStepResize()`. Moreover, the existing ARKLS interface should then be deleted and recreated by attaching the updated `SUNLinearSolver` (and possibly `SUNMatrix`) object(s) through calls to `ARKStepSetLinearSolver()`, and `ARKStepSetMassLinearSolver()`.

If any user-supplied routines are provided to aid the linear solver (e.g. Jacobian construction, Jacobian-vector product, mass-matrix-vector product, preconditioning), then the corresponding "set" routines must be called again **following** the solver re-specification.

4.5.13.2 Resizing the absolute tolerance array

If using array-valued absolute tolerances, the absolute tolerance vector will be invalid after the call to *ARKStepResize()*, so the new absolute tolerance vector should be re-set **following** each call to *ARKStepResize()* through a new call to *ARKStepSVtolerances()* (and similarly to *ARKStepResVtolerance()* if that was used for the original problem).

If scalar-valued tolerances or a tolerance function was specified through either *ARKStepSStolerances()* or *ARKStepWFtolerances()*, then these will remain valid and no further action is necessary.

Note: For an example of *ARKStepResize()* usage, see the supplied serial C example problem, `ark_heat1D_adapt.c`.

4.6 User-supplied functions

The user-supplied functions for ARKStep consist of:

- at least one function defining the ODE (required),
- a function that handles error and warning messages (optional),
- a function that provides the error weight vector (optional),
- a function that provides the residual weight vector (optional),
- a function that handles adaptive time step error control (optional),
- a function that handles explicit time step stability (optional),
- a function that updates the implicit stage prediction (optional),
- a function that defines the root-finding problem(s) to solve (optional),
- one or two functions that provide Jacobian-related information for the linear solver, if a Newton-based nonlinear iteration is chosen (optional),
- one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms, if a Newton-based nonlinear iteration and iterative linear solver are chosen (optional), and
- if the problem involves a non-identity mass matrix $M \neq I$:
  - one or two functions that provide mass-matrix-related information for the linear and mass matrix solvers (required),
  - one or two functions that define the mass matrix preconditioner for use in an iterative mass matrix solver is chosen (optional), and
- a function that handles vector resizing operations, if the underlying vector structure supports resizing (as opposed to deletion/recreation), and if the user plans to call *ARKStepResize()* (optional).

4.6.1 ODE right-hand side

The user must supply at least one function of type *ARKRhsFn* to specify the explicit and/or implicit portions of the ODE system:

typedef int **(*ARKRhsFn)** (realtype *t*, N_Vector *y*, N_Vector *ydot*, void* *user_data*)

These functions compute the ODE right-hand side for a given value of the independent variable t and state vector y.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *ydot* – the output vector that forms a portion of the ODE RHS $f^E(t,y) + f^I(t,y)$.
- *user_data* – the *user_data* pointer that was passed to *ARKStepSetUserData()*.

Return value: An *ARKRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKStep will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *ARK_RHSFUNC_FAIL* is returned).

Notes: Allocation of memory for *ydot* is handled within the ARKStep module.

The vector *ydot* may be uninitialized on input; it is the user's responsibility to fill this entire vector with meaningful values.

A recoverable failure error return from the *ARKRhsFn* is typically used to flag a value of the dependent variable y that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, ARKStep will attempt to recover (possibly repeating the nonlinear iteration, or reducing the step size) in order to avoid this recoverable error return. There are some situations in which recovery is not possible even if the right-hand side function returns a recoverable error flag. One is when this occurs at the very first call to the *ARKRhsFn* (in which case ARKStep returns *ARK_FIRST_RHSFUNC_ERR*). Another is when a recoverable error is reported by *ARKRhsFn* after the integrator completes a successful stage, in which case ARKStep returns *ARK_UNREC_RHSFUNC_ERR*).

4.6.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by *errfp* (see *ARKStepSetErrFile()*), the user may provide a function of type *ARKErrHandlerFn* to process any such messages.

typedef void **(*ARKErrHandlerFn)** (int *error_code*, const char* *module*, const char* *function*, char* *msg*, void* *user_data*)

This function processes error and warning messages from ARKStep and its sub-modules.

Arguments:

- *error_code* – the error code.
- *module* – the name of the ARKStep module reporting the error.
- *function* – the name of the function in which the error occurred.
- *msg* – the error message.
- *user_data* – a pointer to user data, the same as the *eh_data* parameter that was passed to *ARKStepSetErrHandlerFn()*.

Return value: An *ARKErrHandlerFn* function has no return value.

Notes: *error_code* is negative for errors and positive (*ARK_WARNING*) for warnings. If a function that returns a pointer to memory encounters an error, it sets *error_code* to 0.

4.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `ARKEwtFn` to compute a vector *ewt* containing the weights in the WRMS norm $\|v\|_{WRMS} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(ewt_i\ v_i\right)^2\right)^{1/2}$. These weights will be used in place of those defined in the section *Error norms*.

typedef int **(*ARKEwtFn)** (N_Vector *y*, N_Vector *ewt*, void* *user_data*)

This function computes the WRMS error weights for the vector y.

Arguments:

- *y* – the dependent variable vector at which the weight vector is to be computed.
- *ewt* – the output vector containing the error weights.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: An *ARKEwtFn* function must return 0 if it successfully set the error weights, and -1 otherwise.

Notes: Allocation of memory for *ewt* is handled within ARKStep.

The error weight vector must have all components positive. It is the user's responsibility to perform this test and return -1 if it is not satisfied.

4.6.4 Residual weight function

As an alternative to providing the scalar or vector absolute residual tolerances (when the IVP units differ from the solution units), the user may provide a function of type `ARKRwtFn` to compute a vector *rwt* containing the weights in the WRMS norm $\|v\|_{WRMS} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(rwt_i\ v_i\right)^2\right)^{1/2}$. These weights will be used in place of those defined in the section *Error norms*.

typedef int **(*ARKRwtFn)** (N_Vector *y*, N_Vector *rwt*, void* *user_data*)

This function computes the WRMS residual weights for the vector y.

Arguments:

- *y* – the dependent variable vector at which the weight vector is to be computed.
- *rwt* – the output vector containing the residual weights.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: An *ARKRwtFn* function must return 0 if it successfully set the residual weights, and -1 otherwise.

Notes: Allocation of memory for *rwt* is handled within ARKStep.

The residual weight vector must have all components positive. It is the user's responsibility to perform this test and return -1 if it is not satisfied.

4.6.5 Time step adaptivity function

As an alternative to using one of the built-in time step adaptivity methods for controlling solution error, the user may provide a function of type *ARKAdaptFn* to compute a target step size h for the next integration step. These steps should be chosen as the maximum value such that the error estimates remain below 1.

typedef int **(*ARKAdaptFn)**(N_Vector *y*, realtype *t*, realtype *h1*, realtype *h2*, realtype *h3*, realtype *e1*, realtype *e2*, realtype *e3*, int *q*, int *p*, realtype* *hnew*, void* *user_data*)

This function implements a time step adaptivity algorithm that chooses h satisfying the error tolerances.

Arguments:

- *y* – the current value of the dependent variable vector.
- *t* – the current value of the independent variable.
- *h1* – the current step size, $t_n - t_{n-1}$.
- *h2* – the previous step size, $t_{n-1} - t_{n-2}$.
- *h3* – the step size $t_{n-2} - t_{n-3}$.
- *e1* – the error estimate from the current step, n.
- *e2* – the error estimate from the previous step, $n-1$.
- *e3* – the error estimate from the step $n-2$.
- *q* – the global order of accuracy for the method.
- *p* – the global order of accuracy for the embedded method.
- *hnew* – the output value of the next step size.
- *user_data* – a pointer to user data, the same as the *h_data* parameter that was passed to *ARKStepSetAdaptivityFn()*.

Return value: An *ARKAdaptFn* function should return 0 if it successfully set the next step size, and a non-zero value otherwise.

4.6.6 Explicit stability function

A user may supply a function to predict the maximum stable step size for the explicit portion of the ImEx system, $f^E(t,y)$. While the accuracy-based time step adaptivity algorithms may be sufficient for retaining a stable solution to the ODE system, these may be inefficient if $f^E(t,y)$ contains moderately stiff terms. In this scenario, a user may provide a function of type *ARKExpStabFn* to provide this stability information to ARKStep. This function must set the scalar step size satisfying the stability restriction for the upcoming time step. This value will subsequently be bounded by the user-supplied values for the minimum and maximum allowed time step, and the accuracy-based time step.

typedef int **(*ARKExpStabFn)**(N_Vector *y*, realtype *t*, realtype* *hstab*, void* *user_data*)

This function predicts the maximum stable step size for the explicit portions of the ImEx ODE system.

Arguments:

- *y* – the current value of the dependent variable vector.
- *t* – the current value of the independent variable.
- *hstab* – the output value with the absolute value of the maximum stable step size.
- *user_data* – a pointer to user data, the same as the *estab_data* parameter that was passed to *ARKStepSetStabilityFn()*.

Return value: An *ARKExpStabFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.

Notes: If this function is not supplied, or if it returns *hstab* ≤ 0.0, then ARKStep will assume that there is no explicit stability restriction on the time step size.

4.6.7 Implicit stage prediction function

A user may supply a function to update the prediction for each implicit stage solution. If supplied, this routine will be called *after* any existing ARKStep predictor algorithm completes, so that the predictor may be modified by the user as desired. In this scenario, a user may provide a function of type `ARKStagePredictFn` to provide this implicit predictor to ARKStep. This function takes as input the already-predicted implicit stage solution and the corresponding 'time' for that prediction; it then updates the prediction vector as desired. If the user-supplied routine will construct a full prediction (and thus the ARKStep prediction is irrelevant), it is recommended that the user *not* call `ARKStepSetPredictorMethod()`, thereby leaving the default trivial predictor in place.

typedef int **(*ARKStagePredictFn)** (realtype *t*, N_Vector *zpred*, void* *user_data*)
This function updates the prediction for the implicit stage solution.

Arguments:

- *t* – the current value of the independent variable.
- *zpred* – the ARKStep-predicted stage solution on input, and the user-modified predicted stage solution on output.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: An *ARKStagePredictFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.

Notes: This may be useful if there are bound constraints on the solution, and these should be enforced prior to beginning the nonlinear or linear implicit solver algorithm.

This routine is incompatible with the "minimum correction predictor" – option 5 to the routine `ARKStepSetPredictorMethod()`. If both are selected, then ARKStep will override its built-in implicit predictor routine to instead use option 0 (trivial predictor).

4.6.8 Rootfinding function

If a rootfinding problem is to be solved during the integration of the ODE system, the user must supply a function of type `ARKRootFn`.

typedef int **(*ARKRootFn)** (realtype *t*, N_Vector *y*, realtype* *gout*, void* *user_data*)
This function implements a vector-valued function $g(t, y)$ such that the roots of the *nrtfn* components $g_i(t, y)$ are sought.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *gout* – the output array, of length *nrtfn*, with components $g_i(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: An *ARKRootFn* function should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and ARKStep returns *ARK_RTFUNC_FAIL*).

Notes: Allocation of memory for *gout* is handled within ARKStep.

4.6.9 Jacobian construction (matrix-based linear solvers)

If a matrix-based linear solver module is used (i.e., a non-NULL `SUNMatrix` object was supplied to `ARKStepSetLinearSolver()` in section *A skeleton of the user's main program*), the user may provide a function of type `ARKLsJacFn` to provide the Jacobian approximation or `ARKLsLinSysFn` to provide an approximation of the linear system $\mathcal{A}(\sqcup, \dagger) = M(t) - \gamma J(t, y)$.

typedef int **(*ARKLsJacFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, SUNMatrix *Jac*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the Jacobian matrix $J(t, y) = \frac{\partial f^I}{\partial y}(t, y)$ (or an approximation to it).

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector, namely the predicted value of $y(t)$.
- *fy* – the current value of the vector $f^I(t, y)$.
- *Jac* – the output Jacobian matrix.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.
- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used by an ARKLsJacFn as temporary storage or work space.

Return value: An *ARKLsJacFn* function should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKStep will attempt to correct, while ARKLS sets *last_flag* to *ARKLS_JACFUNC_RECVR*), or a negative value if it failed unrecoverably (in which case the integration is halted, `ARKStepEvolve()` returns *ARK_LSETUP_FAIL* and ARKLS sets *last_flag* to *ARKLS_JACFUNC_UNRECVR*).

Notes: Information regarding the structure of the specific `SUNMatrix` structure (e.g.~number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific `SUNMatrix` interface functions (see the section *Matrix Data Structures* for details).

When using a linear solver of type `SUNLINEARSOLVER_DIRECT`, prior to calling the user-supplied Jacobian function, the Jacobian matrix $J(t, y)$ is zeroed out, so only nonzero elements need to be loaded into *Jac*.

With the default nonlinear solver (the native SUNDIALS Netwon method), each call to the user's `ARKLsJacFn()` function is preceded by a call to the implicit `ARKRhsFn()` user function with the same (t, y) arguments. Thus, the Jacobian function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see *Functions provided by SUNDIALS integrators* for more information).

If the user's `ARKLsJacFn` function uses difference quotient approximations, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the ARKStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

dense:

A user-supplied dense Jacobian function must load the *N* by *N* dense matrix *Jac* with an approximation to the Jacobian matrix $J(t,y)$ at the point (t,y). The accessor macros `SM_ELEMENT_D` and `SM_COLUMN_D` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX_DENSE type. `SM_ELEMENT_D(J, i, j)` references the `(i,j)`-th element of the dense matrix `J` (for `i`, `j` between 0 and N-1). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to *N*, the Jacobian element $J_{m,n}$ can be set using the statement `SM_ELEMENT_D(J, m-1, n-1)` $= J_{m,n}$. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the `j`-th column of `J` (for `j` ranging from 0 to *N*-1), and the elements of the `j`-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1); col_n[m-1]` $= J_{m,n}$. For large problems, it is more efficient to use `SM_COLUMN_D` than to use `SM_ELEMENT_D`. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in section *The SUNMATRIX_DENSE Module*.

band:

A user-supplied banded Jacobian function must load the band matrix *Jac* with the elements of the Jacobian $J(t,y)$ at the point (t,y). The accessor macros `SM_ELEMENT_B`, `SM_COLUMN_B`, and `SM_COLUMN_ELEMENT_B` allow the user to read and write band matrix elements without making specific references to the underlying representation of the SUNMATRIX_BAND type. `SM_ELEMENT_B(J, i, j)` references the `(i,j)`-th element of the band matrix `J`, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to *N* with (m,n) within the band defined by *mupper* and *mlower*, the Jacobian element $J_{m,n}$ can be loaded using the statement `SM_ELEMENT_B(J, m-1, n-1)` $= J_{m,n}$. The elements within the band are those with *-mupper* $\leq m-n \leq$ *mlower*. Alternatively, `SM_COLUMN_B(J, j)` returns a pointer to the diagonal element of the `j`-th column of `J`, and if we assign this address to `realtype *col_j`, then the `i`-th element of the `j`-th column is given by `SM_COLUMN_ELEMENT_B(col_j, i, j)`, counting from 0. Thus, for (m,n) within the band, $J_{m,n}$ can be loaded by setting `col_n = SM_COLUMN_B(J, n-1); SM_COLUMN_ELEMENT_B(col_n, m-1, n-1)` $= J_{m,n}$. The elements of the `j`-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type SUNMATRIX_BAND. The array `col_n` can be indexed from *-mupper* to *mlower*. For large problems, it is more efficient to use `SM_COLUMN_B` and `SM_COLUMN_ELEMENT_B` than to use the `SM_ELEMENT_B` macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in section *The SUNMATRIX_BAND Module*.

sparse:

A user-supplied sparse Jacobian function must load the compressed-sparse-column (CSC) or compressed-sparse-row (CSR) matrix *Jac* with an approximation to the Jacobian matrix $J(t,y)$ at the point (t,y). Storage for *Jac* already exists on entry to this function, although the user should ensure that sufficient space is allocated in *Jac* to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a SUNMATRIX_SPARSE object may be accessed using the macro `SM_NNZ_S` or the routine *`SUNSparseMatrix_NNZ()`*. The SUNMATRIX_SPARSE type is further documented in the section *The SUNMATRIX_SPARSE Module*.

typedef int **(*ARKLsLinSysFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, SUNMatrix *A*, SUNMatrix *M*, booleantype *jok*, booleantype **jcur*, realtype *gamma*, void **user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the linear system matrix $\mathcal{A}(t,y) = M(t) - \gamma J(t,y)$ (or an approximation to it).

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector, namely the predicted value of $y(t)$.
- *fy* – the current value of the vector $f^I(t,y)$.

- *A* – the output linear system matrix.
- *M* – the current mass matrix (this input is `NULL` if $M = I$).
- *jok* – is an input flag indicating whether the Jacobian-related data needs to be updated. The *jok* argument provides for the reuse of Jacobian data. When *jok* = `SUNFALSE`, the Jacobian-related data should be recomputed from scratch. When *jok* = `SUNTRUE` the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of *gamma*). A call with *jok* = `SUNTRUE` can only occur after a call with *jok* = `SUNFALSE`.
- *jcur* – is a pointer to a flag which should be set to `SUNTRUE` if Jacobian data was recomputed, or set to `SUNFALSE` if Jacobian data was not recomputed, but saved data was still reused.
- *gamma* – the scalar γ appearing in the Newton matrix given by $\mathcal{A} = M(t) - \gamma J(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.
- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used by an ARKLsLinSysFn as temporary storage or work space.

Return value: An *ARKLsLinSysFn* function should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKStep will attempt to correct, while ARKLS sets *last_flag* to *ARKLS_JACFUNC_RECVR*), or a negative value if it failed unrecoverably (in which case the integration is halted, `ARKStepEvolve()` returns *ARK_LSETUP_FAIL* and ARKLS sets *last_flag* to *ARKLS_JACFUNC_UNRECVR*).

4.6.10 Jacobian-vector product (matrix-free linear solvers)

When using a matrix-free linear solver module for the implicit stage solves (i.e., a NULL-valued SUNMATRIX argument was supplied to `ARKStepSetLinearSolver()` in the section *A skeleton of the user's main program*), the user may provide a function of type `ARKLsJacTimesVecFn` in the following form, to compute matrix-vector products Jv. If such a function is not supplied, the default is a difference quotient approximation to these products.

typedef int **(*ARKLsJacTimesVecFn)** (N_Vector *v*, N_Vector *Jv*, realtype *t*, N_Vector *y*, N_Vector *fy*, void* *user_data*, N_Vector *tmp*)

This function computes the product Jv where $J(t, y) \approx \frac{\partial f^I}{\partial y}(t, y)$.

Arguments:

- *v* – the vector to multiply.
- *Jv* – the output vector computed.
- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^I(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.
- *tmp* – pointer to memory allocated to a variable of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the Jacobian-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the generic Krylov solver, in which case the integration is halted.

Notes: If the user's `ARKLsJacTimesVecFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To

obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the ARKStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

4.6.11 Jacobian-vector product setup (matrix-free linear solvers)

If the user's Jacobian-times-vector routine requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type *ARKLsJacTimesSetupFn*, defined as follows:

typedef int **(*ARKLsJacTimesSetupFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, void* *user_data*)

This function preprocesses and/or evaluates any Jacobian-related data needed by the Jacobian-times-vector routine.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^I(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to *ARKStepSetUserData()*.

Return value: The value to be returned by the Jacobian-vector setup function should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: Each call to the Jacobian-vector setup function is preceded by a call to the implicit *ARKRhsFn* user function with the same (t, y) arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side.

If the user's *ARKLsJacTimesSetupFn* function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the ARKStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

4.6.12 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLinSol solver module, then the user must provide a function of type *ARKLsPrecSolveFn* to solve the linear system $Pz = r$, where P corresponds to either a left or right preconditioning matrix. Here P should approximate (at least crudely) the Newton matrix $\mathcal{A}(t, y) = M(t) - \gamma J(t, y)$, where $M(t)$ is the mass matrix and $J(t, y) = \frac{\partial f^I}{\partial y}(t, y)$ If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate $\mathcal{A}$.

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

This function solves the preconditioner system $Pz = r$.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^I(t, y)$.
- *r* – the right-hand side vector of the linear system.

- z – the computed output solution vector.
- *gamma* – the scalar γ appearing in the Newton matrix given by $\mathcal{A} = M(t) - \gamma J(t, y)$.
- *delta* – an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made to be less than *delta* in the weighted l_2 norm, i.e. $\left(\sum_{i=1}^{n} (Res_i * ewt_i)^2\right)^{1/2} < \delta$, where $\delta =$ *delta*. To obtain the `N_Vector` *ewt*, call `ARKStepGetErrWeights()`.
- *lr* – an input flag indicating whether the preconditioner solve is to use the left preconditioner (*lr* = 1) or the right preconditioner (*lr* = 2).
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

4.6.13 Preconditioner setup (iterative linear solvers)

If the user's preconditioner routine requires that any data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type `ARKLsPrecSetupFn`.

typedef int **(*ARKLsPrecSetupFn)**(realtype *t*, N_Vector *y*, N_Vector *fy*, booleantype *jok*, booleantype* *jcurPtr*, realtype *gamma*, void* *user_data*)

This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^I(t, y)$.
- *jok* – is an input flag indicating whether the Jacobian-related data needs to be updated. The *jok* argument provides for the reuse of Jacobian data in the preconditioner solve function. When *jok* = `SUNFALSE`, the Jacobian-related data should be recomputed from scratch. When *jok* = `SUNTRUE` the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of *gamma*). A call with *jok* = `SUNTRUE` can only occur after a call with *jok* = `SUNFALSE`.
- *jcurPtr* – is a pointer to a flag which should be set to `SUNTRUE` if Jacobian data was recomputed, or set to `SUNFALSE` if Jacobian data was not recomputed, but saved data was still reused.
- *gamma* – the scalar γ appearing in the Newton matrix given by $\mathcal{A} = M(t) - \gamma J(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization of the resulting approximation to $\mathcal{A} = M(t) - \gamma J(t, y)$.

With the default nonlinear solver (the native SUNDIALS Netwon method), each call to the preconditioner setup function is preceded by a call to the implicit `ARKRhsFn` user function with the same (t, y) arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this

is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see *Functions provided by SUNDIALS integrators* for more information).

This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the Newton iteration.

If the user's `ARKLsPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the ARKStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

4.6.14 Mass matrix construction (matrix-based linear solvers)

If a matrix-based mass-matrix linear solver is used (i.e., a non-NULL SUNMATRIX was supplied to `ARKStepSetMassLinearSolver()` in the section *A skeleton of the user's main program*), the user must provide a function of type `ARKLsMassFn` to provide the mass matrix approximation.

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

This function computes the mass matrix $M(t)$ (or an approximation to it).

Arguments:

- *t* – the current value of the independent variable.
- *M* – the output mass matrix.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.
- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used by an ARKLsMassFn as temporary storage or work space.

Return value: An *ARKLsMassFn* function should return 0 if successful, or a negative value if it failed unrecoverably (in which case the integration is halted, `ARKStepEvolve()` returns *ARK_MASSSETUP_FAIL* and ARKLS sets *last_flag* to *ARKLS_MASSFUNC_UNRECVR*).

Notes: Information regarding the structure of the specific `SUNMatrix` structure (e.g.~number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific `SUNMatrix` interface functions (see the section *Matrix Data Structures* for details).

Prior to calling the user-supplied mass matrix function, the mass matrix $M(t)$ is zeroed out, so only nonzero elements need to be loaded into *M*.

dense:

A user-supplied dense mass matrix function must load the *N* by *N* dense matrix *M* with an approximation to the mass matrix $M(t)$. As discussed above in section *Jacobian construction (matrix-based linear solvers)*, the accessor macros `SM_ELEMENT_D` and `SM_COLUMN_D` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX_DENSE type. Similarly, the SUNMATRIX_DENSE type and accessor macros `SM_ELEMENT_D` and `SM_COLUMN_D` are documented in the section *The SUNMATRIX_DENSE Module*.

band:

A user-supplied banded mass matrix function must load the band matrix *M* with the elements of the mass matrix $M(t)$. As discussed above in section *Jacobian construction (matrix-based linear solvers)*, the accessor macros `SM_ELEMENT_B`, `SM_COLUMN_B`, and `SM_COLUMN_ELEMENT_B` allow the user to read and write band matrix elements without making specific references to the underlying representation of the SUNMATRIX_BAND type. Similarly, the SUNMATRIX_BAND type and the accessor macros `SM_ELEMENT_B`,

SM_COLUMN_B, and SM_COLUMN_ELEMENT_B are documented in the section *The SUNMATRIX_BAND Module*.

sparse:

A user-supplied sparse mass matrix function must load the compressed-sparse-column (CSR) or compressed-sparse-row (CSR) matrix *M* with an approximation to the mass matrix $M(t)$. Storage for *M* already exists on entry to this function, although the user should ensure that sufficient space is allocated in *M* to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and row index arrays as needed. The type of *M* is SUNMATRIX_SPARSE, and the amount of allocated space in a SUNMATRIX_SPARSE object may be accessed using the macro SM_NNZ_S or the routine *SUNSparseMatrix_NNZ()*. The SUNMATRIX_SPARSE type is further documented in the section *The SUNMATRIX_SPARSE Module*.

4.6.15 Mass matrix-vector product (matrix-free linear solvers)

If a matrix-free linear solver is to be used for mass-matrix linear systems (i.e., a NULL-valued SUNMATRIX argument was supplied to *ARKStepSetMassLinearSolver()* in the section *A skeleton of the user's main program*), the user *must* provide a function of type *ARKLsMassTimesVecFn* in the following form, to compute matrix-vector products $M(t)\,v$.

typedef int **(*ARKLsMassTimesVecFn)** (N_Vector *v*, N_Vector *Mv*, realtype *t*, void* *mtimes_data*)
This function computes the product $M(t)\,v$ (or an approximation to it).

Arguments:

- *v* – the vector to multiply.
- *Mv* – the output vector computed.
- *t* – the current value of the independent variable.
- *mtimes_data* – a pointer to user data, the same as the *mtimes_data* parameter that was passed to *ARKStepSetMassTimes()*.

Return value: The value to be returned by the mass-matrix-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the generic Krylov solver, in which case the integration is halted.

4.6.16 Mass matrix-vector product setup (matrix-free linear solvers)

If the user's mass-matrix-times-vector routine requires that any mass matrix-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type *ARKLsMassTimesSetupFn*, defined as follows:

typedef int **(*ARKLsMassTimesSetupFn)** (realtype *t*, void* *mtimes_data*)
This function preprocesses and/or evaluates any mass-matrix-related data needed by the mass-matrix-times-vector routine.

Arguments:

- *t* – the current value of the independent variable.
- *mtimes_data* – a pointer to user data, the same as the *mtimes_data* parameter that was passed to *ARKStepSetMassTimes()*.

Return value: The value to be returned by the mass-matrix-vector setup function should be 0 if successful. Any other return value will result in an unrecoverable error of the ARKLS mass matrix solver interface, in which case the integration is halted.

4.6.17 Mass matrix preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLINEAR solver module for mass matrix linear systems, then the user must provide a function of type `ARKLsMassPrecSolveFn` to solve the linear system $Pz = r$, where P may be either a left or right preconditioning matrix. Here P should approximate (at least crudely) the mass matrix $M(t)$. If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate $M(t)$.

typedef int **(*ARKLsMassPrecSolveFn)**(realtype *t*, N_Vector *r*, N_Vector *z*, realtype *delta*, int *lr*, void* *user_data*)

This function solves the preconditioner system $Pz = r$.

Arguments:

- *t* – the current value of the independent variable.
- *r* – the right-hand side vector of the linear system.
- *z* – the computed output solution vector.
- *delta* – an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made to be less than *delta* in the weighted l_2 norm, i.e. $\left(\sum_{i=1}^{n} (Res_i * ewt_i)^2\right)^{1/2} < \delta$, where $\delta =$ *delta*. To obtain the `N_Vector` *ewt*, call `ARKStepGetErrWeights()`.
- *lr* – an input flag indicating whether the preconditioner solve is to use the left preconditioner (*lr* = 1) or the right preconditioner (*lr* = 2).
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

4.6.18 Mass matrix preconditioner setup (iterative linear solvers)

If the user's mass matrix preconditioner above requires that any problem data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type `ARKLsMassPrecSetupFn`.

typedef int **(*ARKLsMassPrecSetupFn)**(realtype *t*, void* *user_data*)

This function preprocesses and/or evaluates mass-matrix-related data needed by the preconditioner.

Arguments:

- *t* – the current value of the independent variable.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKStepSetUserData()`.

Return value: The value to be returned by the mass matrix preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: The operations performed by this function might include forming a mass matrix and performing an incomplete factorization of the result. Although such operations would typically be performed only once at the beginning of a simulation, these may be required if the mass matrix can change as a function of time.

If both this function and a `ARKLsMassTimesSetupFn` are supplied, all calls to this function will be preceded by a call to the `ARKLsMassTimesSetupFn`, so any setup performed there may be reused.

4.6.19 Vector resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatial adaptivity in a PDE simulation), the ARKStep integrator may be "resized" between integration steps, through calls to the `ARKStepResize()` function. Typically, when performing adaptive simulations the solution is stored in a customized user-supplied data structure, to enable adaptivity without repeated allocation/deallocation of memory. In these scenarios, it is recommended that the user supply a customized vector kernel to interface between SUNDIALS and their problem-specific data structure. If this vector kernel includes a function of type `ARKVecResizeFn` to resize a given vector implementation, then this function may be supplied to `ARKStepResize()` so that all internal ARKStep vectors may be resized, instead of deleting and re-creating them at each call. This resize function should have the following form:

typedef int **(*ARKVecResizeFn)**(N_Vector *y*, N_Vector *ytemplate*, void* *user_data*)

This function resizes the vector *y* to match the dimensions of the supplied vector, *ytemplate*.

Arguments:

- *y* – the vector to resize.
- *ytemplate* – a vector of the desired size.
- *user_data* – a pointer to user data, the same as the *resize_data* parameter that was passed to `ARKStepResize()`.

Return value: An *ARKVecResizeFn* function should return 0 if it successfully resizes the vector *y*, and a non-zero value otherwise.

Notes: If this function is not supplied, then ARKStep will instead destroy the vector *y* and clone a new vector *y* off of *ytemplate*.

4.7 Preconditioner modules

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, ARKode provides two internal preconditioner modules that may be used by ARKStep: a banded preconditioner for serial and threaded problems (ARKBANDPRE) and a band-block-diagonal preconditioner for parallel problems (ARKBBDPRE).

4.7.1 A serial banded preconditioner module

This preconditioner provides a band matrix preconditioner for use with iterative SUNLINSOL modules through the ARKLS linear solver interface, in a serial or threaded setting. It requires that the problem be set up using either the NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS module, due to data access patterns. It also currently requires that the problem involve an identity mass matrix, i.e. $M = I$.

This module uses difference quotients of the ODE right-hand side function f^I to generate a band matrix of bandwidth `ml + mu + 1`, where the number of super-diagonals (`mu`, the upper half-bandwidth) and sub-diagonals (`ml`, the lower half-bandwidth) are specified by the user. This band matrix is used to to form a preconditioner the Krylov linear solver. Although this matrix is intended to approximate the Jacobian $J = \frac{\partial f^I}{\partial y}$, it may be a very crude approximation, since the true Jacobian may not be banded, or its true bandwidth may be larger than `ml + mu + 1`. However, as long as the banded approximation generated for the preconditioner is sufficiently accurate, it may speed convergence of the Krylov iteration.

4.7.1.1 ARKBANDPRE usage

In order to use the ARKBANDPRE module, the user need not define any additional functions. In addition to the header files required for the integration of the ODE problem (see the section *Access to library and header files*), to use the ARKBANDPRE module, the user's program must include the header file `arkode_bandpre.h` which declares the needed function prototypes. The following is a summary of the usage of this module. Steps that are unchanged from the skeleton program presented in *A skeleton of the user's main program* are *italicized*.

1. *Initialize multi-threaded environment (if appropriate)*
2. *Set problem dimensions*
3. *Set vector of initial values*
4. *Create ARKStep object*
5. *Specify integration tolerances*
6. Create iterative linear solver object

   When creating the iterative linear solver object, specify the type of preconditioning (`PREC_LEFT` or `PREC_RIGHT`) to use.
7. *Set linear solver optional inputs*
8. *Attach linear solver module*
9. Initialize the ARKBANDPRE preconditioner module

   Specify the upper and lower half-bandwidths (`mu` and `ml`, respectively) and call

   ```
   ier = ARKBandPrecInit(arkode_mem, N, mu, ml);
   ```

   to allocate memory and initialize the internal preconditioner data.
10. *Set optional inputs*

    Note that the user should not call *ARKStepSetPreconditioner()* as it will overwrite the preconditioner setup and solve functions.
11. *Create nonlinear solver object*
12. *Attach nonlinear solver module*
13. *Set nonlinear solver optional inputs*
14. *Specify rootfinding problem*
15. *Advance solution in time*
16. Get optional outputs

    Additional optional outputs associated with ARKBANDPRE are available by way of the two routines described below, *ARKBandPrecGetWorkSpace()* and *ARKBandPrecGetNumRhsEvals()*.
17. *Deallocate memory for solution vector*
18. *Free solver memory*
19. *Free linear solver memory*

4.7.1.2 ARKBANDPRE user-callable functions

The ARKBANDPRE preconditioner module is initialized and attached by calling the following function:

int **ARKBandPrecInit**(void* *arkode_mem*, sunindextype *N*, sunindextype *mu*, sunindextype *ml*)
Initializes the ARKBANDPRE preconditioner and allocates required (internal) memory for it.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *N* – problem dimension (size of ODE system).
- *mu* – upper half-bandwidth of the Jacobian approximation.
- *ml* – lower half-bandwidth of the Jacobian approximation.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_ILL_INPUT* if an input has an illegal value
- *ARKLS_MEM_FAIL* if a memory allocation request failed

Notes: The banded approximate Jacobian will have nonzero elements only in locations (i, j) with $ml \leq j - i \leq mu$.

The following two optional output functions are available for use with the ARKBANDPRE module:

int **ARKBandPrecGetWorkSpace**(void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)
Returns the sizes of the ARKBANDPRE real and integer workspaces.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lenrwLS* – the number of `realtype` values in the ARKBANDPRE workspace.
- *leniwLS* – the number of integer values in the ARKBANDPRE workspace.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: The workspace requirements reported by this routine correspond only to memory allocated within the ARKBANDPRE module (the banded matrix approximation, banded `SUNLinearSolver` object, and temporary vectors).

The workspaces referred to here exist in addition to those given by the corresponding function `ARKStepGetLSWorkspace()`.

int **ARKBandPrecGetNumRhsEvals**(void* *arkode_mem*, long int* *nfevalsBP*)
Returns the number of calls made to the user-supplied right-hand side function f^I for constructing the finite-difference banded Jacobian approximation used within the preconditioner setup function.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *nfevalsBP* – number of calls to f^I.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: The counter *nfevalsBP* is distinct from the counter *nfevalsLS* returned by the corresponding function `ARKStepGetNumLSRhsEvals()` and also from *nfi_evals* returned by *ARKStepGetNumRhsEvals()*. The total number of right-hand side function evaluations is the sum of all three of these counters, plus the *nfe_evals* counter for f^E calls returned by *ARKStepGetNumRhsEvals()*.

4.7.2 A parallel band-block-diagonal preconditioner module

A principal reason for using a parallel ODE solver (such as ARKode) lies in the solution of partial differential equations (PDEs). Moreover, Krylov iterative methods are used on many such problems due to the nature of the underlying linear system of equations that needs to solved at each time step. For many PDEs, the linear algebraic system is large, sparse and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner is required. Otherwise, the rate of convergence of the Krylov iterative method is usually slow, and degrades as the PDE mesh is refined. Typically, an effective preconditioner must be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used with CVODE for several realistic, large-scale problems *[HT1998]*. It is included in a software module within the ARKode package, and is accessible within the ARKStep time stepping module. This preconditioning module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers through the ARKLS interface. It generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called ARKBBDPRE.

One way to envision these preconditioners is to think of the computational PDE domain as being subdivided into Q non-overlapping subdomains, where each subdomain is assigned to one of the Q MPI tasks used to solve the ODE system. The basic idea is to isolate the preconditioning so that it is local to each process, and also to use a (possibly cheaper) approximate right-hand side function for construction of this preconditioning matrix. This requires the definition of a new function $g(t,y) \approx f^I(t,y)$ that will be used to construct the BBD preconditioner matrix. At present, we assume that the ODE be written in explicit form as

$$\dot{y} = f^E(t,y) + f^I(t,y),$$

where f^I corresponds to the ODE components to be treated implicitly, i.e. this preconditioning module does not support problems with non-identity mass matrices. The user may set $g = f^I$, if no less expensive approximation is desired.

Corresponding to the domain decomposition, there is a decomposition of the solution vector y into Q disjoint blocks y_q, and a decomposition of g into blocks g_q. The block g_q depends both on y_p and on components of blocks $y_{q'}$ associated with neighboring subdomains (so-called ghost-cell data). If we let $\bar{y}_q$ denote y_q augmented with those other components on which g_q depends, then we have

$$g(t,y) = \left[g_1(t,\bar{y}_1), g_2(t,\bar{y}_2), \ldots, g_Q(t,\bar{y}_Q)\right]^T,$$

and each of the blocks $g_q(t,\bar{y}_q)$ is decoupled from one another.

The preconditioner associated with this decomposition has the form

$$P = \text{diag}[P_1, P_2, \ldots, P_Q]$$

where

$$P_q \approx I - \gamma J_q$$

and where J_q is a difference quotient approximation to $\frac{\partial g_q}{\partial y_q}$. This matrix is taken to be banded, with upper and lower half-bandwidths *mudq* and *mldq* defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using *mudq* + *mldq* + 2 evaluations of g_m, but only a matrix of bandwidth *mukeep* + *mlkeep* + 1 is retained. Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of g, if smaller values provide a more efficient preconditioner. The solution of the complete linear system

$$Px = b$$

reduces to solving each of the distinct equations

$$P_q x_q = b_q, \quad q = 1, \ldots, Q,$$

and this is done by banded LU factorization of P_q followed by a banded backsolve.

Similar block-diagonal preconditioners could be considered with different treatments of the blocks P_q. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

4.7.2.1 ARKBBDPRE user-supplied functions

The ARKBBDPRE module calls two user-provided functions to construct P: a required function *gloc* (of type `ARKLocalFn()`) which approximates the right-hand side function $g(t, y) \approx f^I(t, y)$ and which is computed locally, and an optional function *cfn* (of type `ARKCommFn()`) which performs all inter-process communication necessary to evaluate the approximate right-hand side g. These are in addition to the user-supplied right-hand side function f^I. Both functions take as input the same pointer *user_data* that is passed by the user to `ARKStepSetUserData()` and that was passed to the user's function f^I. The user is responsible for providing space (presumably within *user_data*) for components of y that are communicated between processes by *cfn*, and that are then used by *gloc*, which should not do any communication.

typedef int **(*ARKLocalFn)**(sunindextype *Nlocal*, realtype *t*, N_Vector *y*, N_Vector *glocal*, void* *user_data*)

This *gloc* function computes $g(t, y)$. It fills the vector *glocal* as a function of *t* and *y*.

Arguments:

- *Nlocal* – the local vector length.
- *t* – the value of the independent variable.
- *y* – the value of the dependent variable vector on this process.
- *glocal* – the output vector of $g(t, y)$ on this process.
- *user_data* – a pointer to user data, the same as the *user_data* parameter passed to `ARKStepSetUserData()`.

Return value: An *ARKLocalFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKStep will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `ARKStepEvolve()` will return *ARK_LSETUP_FAIL*).

Notes: This function should assume that all inter-process communication of data needed to calculate *glocal* has already been done, and that this data is accessible within user data.

The case where g is mathematically identical to f^I is allowed.

typedef int **(*ARKCommFn)**(sunindextype *Nlocal*, realtype *t*, N_Vector *y*, void* *user_data*)

This *cfn* function performs all inter-process communication necessary for the execution of the *gloc* function above, using the input vector *y*.

Arguments:

- *Nlocal* – the local vector length.
- *t* – the value of the independent variable.
- *y* – the value of the dependent variable vector on this process.
- *user_data* – a pointer to user data, the same as the *user_data* parameter passed to `ARKStepSetUserData()`.

Return value: An *ARKCommFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKStep will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `ARKStepEvolve()` will return *ARK_LSETUP_FAIL*).

Notes: The *cfn* function is expected to save communicated data in space defined within the data structure *user_data*.

Each call to the *cfn* function is preceded by a call to the right-hand side function f^I with the same (t, y) arguments. Thus, *cfn* can omit any communication done by f^I if relevant to the evaluation of *glocal*. If all necessary communication was done in f^I, then *cfn* = `NULL` can be passed in the call to `ARKBBDPrecInit()` (see below).

4.7.2.2 ARKBBDPRE usage

In addition to the header files required for the integration of the ODE problem (see the section *Access to library and header files*), to use the ARKBBDPRE module, the user's program must include the header file `arkode_bbdpre.h` which declares the needed function prototypes.

The following is a summary of the proper usage of this module. Steps that are unchanged from the skeleton program presented in *A skeleton of the user's main program* are *italicized*.

1. *Initialize MPI*
2. *Set problem dimensions*
3. *Set vector of initial values*
4. *Create ARKStep object*
5. *Specify integration tolerances*
6. Create iterative linear solver object

   When creating the iterative linear solver object, specify the type of preconditioning (`PREC_LEFT` or `PREC_RIGHT`) to use.
7. *Set linear solver optional inputs*
8. *Attach linear solver module*
9. Initialize the ARKBBDPRE preconditioner module

   Specify the upper and lower half-bandwidths for computation `mudq` and `mldq`, the upper and lower half-bandwidths for storage `mukeep` and `mlkeep`, and call

   ```
   ier = ARKBBDPrecInit(arkode_mem, Nlocal, mudq, mldq, mukeep, mlkeep,
   dqrely, gloc, cfn);
   ```

to allocate memory and initialize the internal preconditioner data. The last two arguments of *ARKBBDPrecInit()* are the two user-supplied functions of type *ARKLocalFn()* and *ARKCommFn()* described above, respectively.

10. *Set optional inputs*

    Note that the user should not call *ARKStepSetPreconditioner()* as it will overwrite the preconditioner setup and solve functions.

11. *Create nonlinear solver object*
12. *Attach nonlinear solver module*
13. *Set nonlinear solver optional inputs*
14. *Specify rootfinding problem*
15. *Advance solution in time*
16. *Get optional outputs*

    Additional optional outputs associated with ARKBBDPRE are available through the routines *ARKBBDPrecGetWorkSpace()* and *ARKBBDPrecGetNumGfnEvals()*.

17. *Deallocate memory for solution vector*
18. *Free solver memory*
19. *Free linear solver memory*
20. *Finalize MPI*

4.7.2.3 ARKBBDPRE user-callable functions

The ARKBBDPRE preconditioner module is initialized (or re-initialized) and attached to the integrator by calling the following functions:

int **ARKBBDPrecInit**(void* *arkode_mem*, sunindextype *Nlocal*, sunindextype *mudq*, sunindextype *mldq*, sunindextype *mukeep*, sunindextype *mlkeep*, realtype *dqrely*, *ARKLocalFn* *gloc*, *ARKCommFn* *cfn*)

Initializes and allocates (internal) memory for the ARKBBDPRE preconditioner.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *Nlocal* – local vector length.
- *mudq* – upper half-bandwidth to be used in the difference quotient Jacobian approximation.
- *mldq* – lower half-bandwidth to be used in the difference quotient Jacobian approximation.
- *mukeep* – upper half-bandwidth of the retained banded approximate Jacobian block.
- *mlkeep* – lower half-bandwidth of the retained banded approximate Jacobian block.
- *dqrely* – the relative increment in components of y used in the difference quotient approximations. The default is *dqrely* = $\sqrt{\text{unit roundoff}}$, which can be specified by passing *dqrely* = 0.0.
- *gloc* – the name of the C function (of type *ARKLocalFn()*) which computes the approximation $g(t, y) \approx f^I(t, y)$.
- *cfn* – the name of the C function (of type *ARKCommFn()*) which performs all inter-process communication required for the computation of $g(t, y)$.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_ILL_INPUT* if an input has an illegal value
- *ARKLS_MEM_FAIL* if a memory allocation request failed

Notes: If one of the half-bandwidths *mudq* or *mldq* to be used in the difference quotient calculation of the approximate Jacobian is negative or exceeds the value *Nlocal*-1, it is replaced by 0 or *Nlocal*-1 accordingly.

The half-bandwidths *mudq* and *mldq* need not be the true half-bandwidths of the Jacobian of the local block of g when smaller values may provide a greater efficiency.

Also, the half-bandwidths *mukeep* and *mlkeep* of the retained banded approximate Jacobian block may be even smaller than *mudq* and *mldq*, to reduce storage and computational costs further.

For all four half-bandwidths, the values need not be the same on every processor.

The ARKBBDPRE module also provides a re-initialization function to allow solving a sequence of problems of the same size, with the same linear solver choice, provided there is no change in *Nlocal*, *mukeep*, or *mlkeep*. After solving one problem, and after calling `ARKStepReInit()` to re-initialize ARKStep for a subsequent problem, a call to `ARKBBDPrecReInit()` can be made to change any of the following: the half-bandwidths *mudq* and *mldq* used in the difference-quotient Jacobian approximations, the relative increment *dqrely*, or one of the user-supplied functions *gloc* and *cfn*. If there is a change in any of the linear solver inputs, an additional call to the "Set" routines provided by the SUNLINSOL module, and/or one or more of the corresponding `ARKStepSet***` functions, must also be made (in the proper order).

int **`ARKBBDPrecReInit`**(void* *arkode_mem*, sunindextype *mudq*, sunindextype *mldq*, realtype *dqrely*)
Re-initializes the ARKBBDPRE preconditioner module.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *mudq* – upper half-bandwidth to be used in the difference quotient Jacobian approximation.
- *mldq* – lower half-bandwidth to be used in the difference quotient Jacobian approximation.
- *dqrely* – the relative increment in components of y used in the difference quotient approximations. The default is *dqrely* = $\sqrt{\text{unit roundoff}}$, which can be specified by passing *dqrely* = 0.0.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: If one of the half-bandwidths *mudq* or *mldq* is negative or exceeds the value *Nlocal*-1, it is replaced by 0 or *Nlocal*-1 accordingly.

The following two optional output functions are available for use with the ARKBBDPRE module:

int **`ARKBBDPrecGetWorkSpace`**(void* *arkode_mem*, long int* *lenrwBBDP*, long int* *leniwBBDP*)
Returns the processor-local ARKBBDPRE real and integer workspace sizes.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *lenrwBBDP* – the number of `realtype` values in the ARKBBDPRE workspace.

- *leniwBBDP* – the number of integer values in the ARKBBDPRE workspace.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: The workspace requirements reported by this routine correspond only to memory allocated within the ARKBBDPRE module (the banded matrix approximation, banded `SUNLinearSolver` object, temporary vectors). These values are local to each process.

The workspaces referred to here exist in addition to those given by the corresponding function `ARKStepGetLSWorkSpace()`.

int **`ARKBBDPrecGetNumGfnEvals`**(void* *arkode_mem*, long int* *ngevalsBBDP*)

Returns the number of calls made to the user-supplied *gloc* function (of type *ARKLocalFn()*) due to the finite difference approximation of the Jacobian blocks used within the preconditioner setup function.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *ngevalsBBDP* – the number of calls made to the user-supplied *gloc* function.

Return value:

- *ARKLS_SUCCESS* if no errors occurred
- *ARKLS_MEM_NULL* if the ARKStep memory is `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKLS_PMEM_NULL* if the preconditioner memory is `NULL`

In addition to the *ngevalsBBDP* *gloc* evaluations, the costs associated with ARKBBDPRE also include *nlinsetups* LU factorizations, *nlinsetups* calls to *cfn*, *npsolves* banded backsolve calls, and *nfevalsLS* right-hand side function evaluations, where *nlinsetups* is an optional ARKStep output and *npsolves* and *nfevalsLS* are linear solver optional outputs (see the table *Linear solver interface optional output functions*).

4.8 Multigrid Reduction in Time with XBraid

The prior sections discuss using ARKStep in a traditional sequential time integration setting i.e., the solution is advanced from one time to the next where all parallelism resides within the evaluation of a step e.g., the computation of the right-hand side, (non)linear solves, vector operations etc. For example, when discretizing a partial differential equation using a method of lines approach the spatially-discretized equations comprise a large set of ordinary differential equations that can be evolved with ARKStep. In this case the parallelization lies in decomposing the spatial domain unknowns across distributed computational nodes. Considering the strong scaling case at a given spatial resolution, as the problem is spread across greater numbers of computational nodes scalability in the spatial dimension is exhausted and sequential time integration becomes a bottleneck. This bottleneck is largely driven by the hardware shift from faster clock speeds to greater concurrency to achieve performance gains. In this case, at the spatial scaling limit and with stagnant clock speeds, more time steps will lead to an increased runtime.

An alternative approach to sequential time integration is to solve for all time values simultaneously. One such approach is multigrid reduction in time *[F2014]* (MGRIT) which uses a highly parallel iterative method to expose parallelism in the time domain in addition to the spatial parallelization. Starting with an initial temporal grid the multi-

level algorithm constructs successively coarser time grids and uses each coarse grid solution to improve the solution at the next finer scale. In the two level case the MGRIT algorithm is as follows:

1. Relax the solution on the fine grid (parallel-in-time)
2. Restrict the solution to the fine grid (time re-discretization).
3. Solve the residual equation on the coarse grid (serial-in-time).
4. Correct the fine grid solution (parallel-in-time).

Applying this algorithm recursively for the solve step above leads to the multilevel algorithm.

The XBraid library *[XBraid]* implements in the MGRIT algorithm in a non-intrusive manner, enabling the reuse of existing software for sequential time integration. The following sections describe the ARKStep + XBraid interface and the steps necessary to modify an existing code using ARKStep to also use XBraid.

4.8.1 SUNBraid Interface

Interfacing ARKStep with XBraid requires defining two data structures. The first is the XBraid application data structure that contains the data necessary for carrying out a time step and is passed to every interface function (much like the user data pointer in SUNDIALS packages). For this structure the SUNBraid interface defines the generic SUNBraidApp structure described below that serves as the basis for creating integrator-specific or user-defined interfaces to XBraid. The second structure holds the problem state data at a certain time value. This structure is defined by the SUNBraidVector structure and simply contains an N_Vector. In addition to the two data structures several functions defined by the XBraid API are required. These functions include vector operations (e.g., computing vector sums or norms) as well as functions to initialize the problem state, access the current solution, and take a time step.

The ARKBraid interface, built on the SUNBraidApp and SUNBraidVector structures, provides all the functionaly needed combine ARKStep and XBraid for parallel-in-time integration. As such only a minimal number of changes are necessary to update an exsting code that uses ARKStep to also use XBraid.

4.8.1.1 SUNBraidApp

As mentioned above the SUNBraid interface defines the SUNBraidApp structure to hold the data necessary to compute a time step. This structure, like other SUNDIALS generic objects, is defined as a structure consisting of an implementation specific *content* field and an operations structure comprised of a set of function pointers for implmentation-defined operations on the object. Specifically the SUNBraidApp type is defined as

```
/* Define XBraid App structure */
struct _braid_App_struct
{
  void        *content;
  SUNBraidOps ops;
};

/* Pointer to the interface object (same as braid_App) */
typedef struct _braid_App_struct *SUNBraidApp;
```

Here, the SUNBraidOps structure is defined as

```
/* Structure containing function pointers to operations */
struct _SUNBraidOps
{
  int (*getvectmpl)(braid_App app, N_Vector *tmpl);
};
```

```
/* Pointer to operations structure */
typedef struct _SUNBraidOps *SUNBraidOps;
```

The generic SUNBraidApp defines and implements the generic operations acting on a SUNBraidApp obejct. These generic functions are nothing but wrappers to access the specific implementation through the object's operations structure. To illustrate this point we show below the implementation of the *SUNBraidApp_GetVecTmpl()* function:

```
/* Get a template vector from the integrator */
int SUNBraidApp_GetVecTmpl(braid_App app, N_Vector *y)
{
  if (app->ops->getvectmpl == NULL) return SUNBRAID_OPNULL;
  return app->ops->getvectmpl(app, y);
}
```

The SUNBraidApp operations are define below in *SUNBraidOps*.

4.8.1.2 SUNBraidOps

In this section we define the SUNBraidApp operations and, for each operation, we give the function signature, a description of the expected behavior, and an example usage of the function.

int **SUNBraidApp_GetVecTmpl** (braid_App *app*, N_Vector **y*)
This function returns a vector to use as a template for creating new vectors with *N_VClone()*.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *y* – output, the template vector.

Return value:

If this function is not implemented by the SUNBraidApp implementation (i.e., the function pointer is NULL) then this function will return *SUNBRAID_OPNULL*. Otherwise the return value depends on the particular SUNBraidApp implementation. Users are encouraged to utilize the return codes defined in `sundials/sundials_xbraid.h` and listed in *Table: SUNBraid Return Codes*.

Usage:

```
/* Get template vector */
flag = SUNBraidApp_GetVecTmpl(app, y_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

4.8.1.3 SUNBraidApp Utility Functions

In addition to the generic SUNBraidApp operations the following utility functions are provided to assist in creating and destroying a SUNBraidApp instance.

int **SUNBraidApp_NewEmpty** (braid_App **app*)
This function creates a new SUNBraidApp instance with the content and operations initialized to NULL.

Arguments:

- *app* – output, an empty SUNBraidApp instance (XBraid app structure).

Return value:

- *SUNBRAID_SUCCESS* if successful.

- *SUNBRAID_ALLOCFAIL* if a memory allocation failed.

Usage:

```
/* Create empty XBraid interface object */
flag = SUNBraidApp_NewEmpty(app_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

int **SUNBraidApp_FreeEmpty**(braid_App *app)
This function destroys an empty SUNBraidApp instance.

Arguments:

- *app* – input, an empty SUNBraidApp instance (XBraid app structure).

Return value:

- *SUNBRAID_SUCCESS* if successful.

Usage:

```
/* Free empty XBraid interface object */
flag = SUNBraidApp_FreeEmpty(app_ptr);
```

Warning: This function does not free the SUNBraidApp object's content structure. An implementation should free its content before calling *SUNBraidApp_FreeEmpty()* to deallocate the base SUNBraidApp structure.

4.8.1.4 SUNBraidVector

As mentioned above the SUNBraid interface defines the SUNBraidVector structure to store a snapshot of solution data at a single point in time and this structure simply contains an N_Vector. Specifically, the structure is defined as follows:

```
struct _braid_Vector_struct
{
  N_Vector y;
};

/* Poiner to vector wrapper (same as braid_Vector) */
typedef struct _braid_Vector_struct *SUNBraidVector;
```

To assist in creating creating and destroying this structure the following utility functions are provided.

int **SUNBraidVector_New**(N_Vector *y*, SUNBraidVector **u*)
This function creates a new SUNBraidVector wrapping the N_Vector y.

Arguments:

- *y* – input, the N_Vector to wrap.
- *u* – output, the SUNBraidVector wrapping *y*.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *y* is `NULL`.
- *SUNBRAID_ALLOCFAIL* if a memory allocation fails.

Usage:

```
/* Create new vector wrapper */
flag = SUNBraidVector_New(y, u_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

Warning: The SUNBraidVector takes ownership of the wrapped N_Vector and as such the wrapped N_Vector is destroyed when the SUNBraidVector is freed with *SUNBraidVector_Free()*.

int **SUNBraidVector_GetNVector**(SUNBraidVector *u*, N_Vector **y*)
This function retrieves the wrapped N_Vector from the SUNBraidVector.

Arguments:

- *u* – input, the SUNBraidVector wrapping *y*.
- *y* – output, the wrapped N_Vector.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *u* is NULL.
- *SUNBRAID_MEMFAIL* if *y* is NULL.

Usage:

```
/* Create new vector wrapper */
flag = SUNBraidVector_GetNVector(u, y_ptr);
if (flag != SUNBRAID_SUCCESS) return flag;
```

Finally, the SUNBraid interface defines the following vector operations acting on SUNBraidVectors, that consist of then wrappers to compatible SUNDIALS N_Vector operations.

int **SUNBraidVector_Clone**(braid_App *app*, braid_Vector *u*, braid_Vector **v_ptr*)
This function creates a clone of the input SUNBraidVector and copies the values of the input vector *u* into the output vector *v_ptr* using *N_VClone()* and *N_VScale()*.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *u* – input, the SUNBraidVector to clone.
- *v_ptr* – output, the new SUNBraidVector.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *u* is NULL.
- *SUNBRAID_MEMFAIL* if the N_Vector *y* wrapped by *u* is NULL.
- *SUNBRAID_ALLOCFAIL* if a memory allocation fails.

int **SUNBraidVector_Free**(braid_App *app*, braid_Vector *u*)
This function destroys the SUNBraidVector and the wrapped N_Vector using *N_VDestroy()*.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).

- *u* – input, the SUNBraidVector to destroy.

Return value:

- *SUNBRAID_SUCCESS* if successful.

int **SUNBraidVector_Sum**(braid_App *app*, braid_Real *alpha*, braid_Vector *x*, braid_Real *beta*, braid_Vector *y*)

This function computes the vector sum $\alpha x + \beta y \to y$ using `N_VLinearSum()`.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *alpha* – input, the constant α.
- *x* – input, the vector x.
- *beta* – input, the constant β.
- *y* – input/output, the vector y.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *x* or *y* is `NULL`.
- *SUNBRAID_MEMFAIL* if either of the wrapped N_Vectors are `NULL`.

int **SUNBraidVector_SpatialNorm**(braid_App, braid_Vector *u*, braid_Real **norm_ptr*)

This function computes the 2-norm of the vector *u* using `N_VDotProd()`.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *u* – input, the vector *u*.
- *norm_ptr* – output, the L2 norm of *u*.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *u* is `NULL`.
- *SUNBRAID_MEMFAIL* if the wrapped N_Vector is `NULL`.

int **SUNBraidVector_BufSize**(braid_App *app*, braid_Int **size_ptr*, braid_BufferStatus *bstatus*)

This function returns the buffer size for messages to exchange vector data using `SUNBraidApp_GetVecTmpl()` and `N_VBufSize()`.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *size_ptr* – output, the buffer size.
- *bstatus* – input, a status object to query for information on the message type.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- An error flag from `SUNBraidApp_GetVecTmpl()` or `N_VBufSize()`.

int **SUNBraidVector_BufPack**(braid_App *app*, braid_Vector *u*, void **buffer*, braid_BufferStatus *bstatus*)

This function packs the message buffer for exchanging vector data using `N_VBufPack()`.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *u* – input, the vector to pack into the exchange buffer.
- *buffer* – output, the packed exchange buffer to pack.
- *bstatus* – input, a status object to query for information on the message type.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *u* is `NULL`.
- An error flag from `N_VBufPack()`.

int **SUNBraidVector_BufUnpack**(braid_App *app*, void **buffer*, braid_Vector **u_ptr*, braid_BufferStatus *bstatus*)

This function unpacks the message buffer and creates a new N_Vector and SUNBraidVector with the buffer data using `N_VBufUnpack()`, `SUNBraidApp_GetVecTmpl()`, and `N_VClone()`.

Arguments:

- *app* – input, a SUNBraidApp instance (XBraid app structure).
- *buffer* – input, the exchange buffer to unpack.
- *u_ptr* – output, a new SUNBraidVector containing the buffer data.
- *bstatus* – input, a status object to query for information on the message type.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *buffer* is `NULL`.
- *SUNBRAID_ALLOCFAIL* if a memory allocation fails.
- An error flag from `SUNBraidApp_GetVecTmpl()` and `N_VBufUnpack()`.

4.8.1.5 SUNBraid Return Codes

The SUNBraid interface return values are given in *Table: SUNBraid Return Codes*.

Table 4.1: SUNBraid Return Codes

Return value name	Value	Meaning
`SUNBRAID_SUCCESS`	0	The call/operation was successful.
`SUNBRAID_ALLOCFAIL`	−1	A memory allocation failed.
`SUNBRAID_MEMFAIL`	−2	A memory access fail.
`SUNBRAID_OPNULL`	−3	The SUNBraid operation is `NULL`.
`SUNBRAID_ILLINPUT`	−4	An invalid input was provided.
`SUNBRAID_BRAIDFAIL`	−5	An XBraid function failed.
`SUNBRAID_SUNFAIL`	−6	A SUNDIALS function failed.

4.8.2 ARKBraid Interface

This section describes the ARKBraid implementation of a SUNBraidApp for using the ARKStep integration module with XBraid. The following section describes *ARKBraid Initialization and Deallocation Functions* for creating, initializing, and destroying the ARKStep + XBraid interface, *ARKBraid Set Functions* for setting optional inputs, and *ARKBraid Get Functions* for retrieving data from an ARKBraid instance. As noted above, interfacing with XBraid requires providing functions to initialize the problem state, access the current solution, and take a time step. The default ARKBraid functions for each of these actions are defined in *ARKBraid Interface Functions* and may be overridden by user-defined if desired. A skeleton of the user's main or calling program for using the ARKBraid interface is given in *A skeleton of the user's main program with XBraid*. Finally, for advanced users that wish to create their own SUNBraidApp implementation using ARKStep the *Advanced ARKBraid Utility Functions* section describes some helpful functions available to the user.

4.8.2.1 ARKBraid Initialization and Deallocation Functions

This section describes the functions that are called by the user to create, initialize, and destroy an ARKBraid instance. Each user-callable function returns `SUNBRAID_SUCCESS` (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in *Table: SUNBraid Return Codes*.

int **`ARKBraid_Create`**(void **arkode_mem*, braid_App **app*)

This function creates a SUNBraidApp object, sets the content pointer to the private ARKBraid interface structure, and attaches the necessary SUNBraidOps implementations.

Arguments:

- *arkode_mem* – input, a pointer to an ARKStep memory structure.
- *app* – output, an ARKBraid instance (XBraid app structure).

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT arkode_mem* is `NULL`.
- *SUNBRAID_ALLOCFAIL* if a memory allocation failed.

> **Warning:** The ARKBraid interface is ARKStep-specific. Although one could eventually construct an XBraid interface to either ERKStep or MRIStep, those are not supported by this implementation.

int **`ARKBraid_BraidInit`**(MPI_Comm *comm_w*, MPI_Comm *comm_t*, realtype *tstart*, realtype *tstop*, sunindextype *ntime*, braid_App *app*, braid_Core **core*)

This function wraps the XBraid `braid_Init()` function to create the XBraid core memory structure and initializes XBraid with the ARKBraid and SUNBraidVector interface functions.

Arguments:

- *comm_w* – input, the global MPI communicator for space and time.
- *comm_t* – input, the MPI communicator for the time dimension.
- *tstart* – input, the initial time value.
- *tstop* – input, the final time value.
- *ntime* – input, the initial number of grid points in time.
- *app* – input, an ARKBraid instance.
- *core* – output, the XBraid core memory structure.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if either MPI communicator is `MPI_COMM_NULL`, if *ntime* < 2, or if *app* or its content is `NULL`.
- *SUNBRAID_BRAIDFAIL* if the `braid_Init()` call fails. The XBraid return value can be retrieved with *ARKBraid_GetLastBraidFlag()*.

Note: If desired, the default functions for vector initialization, accessing the solution, taking a time step, and computing the spatial norm should be overridden before calling this function. See *ARKBraid Set Functions* for more details.

Warning: The user is responsible for deallocating the XBraid core memory structure with the XBraid function `braid_Destroy()`.

int **ARKBraid_Free**(braid_App **app*)
This function deallocates an ARKBraid instance.

Arguments:

- *app* – input, a pointer to an ARKBraid instance.

Return value:

- *SUNBRAID_SUCCESS* if successful.

4.8.2.2 ARKBraid Set Functions

This section describes the functions that are called by the user to set optional inputs to control the behavior of an ARKBraid instance or to provide alternative XBraid interface functions. Each user-callable function returns `SUNBRAID_SUCCESS` (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in *Table: SUNBraid Return Codes*.

int **ARKBraid_SetStepFn**(braid_App *app*, braid_PtFcnStep *step*)
This function sets the step function provided to XBraid (default *ARKBraid_Step()*).

Arguments:

- *app* – input, an ARKBraid instance.
- *step* – input, an XBraid step function. If *step* is `NULL`, the default function will be used.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content is `NULL`.

Note: This function must be called prior to *ARKBraid_BraidInit()*.

---

int **ARKBraid_SetInitFn**(braid_App *app*, braid_PtFcnInit *init*)
This function sets the vector initialization function provided to XBraid (default *ARKBraid_Init()*).

Arguments:

- *app* – input, an ARKBraid instance.
- *init* – input, an XBraid vector initialization function. If *init* is `NULL`, the default function will be used.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content is `NULL`.

Note: This function must be called prior to *ARKBraid_BraidInit()*.

---

int **ARKBraid_SetSpatialNormFn**(braid_App *app*, braid_PtFcnSpatialNorm *snorm*)
This function sets the spatial norm function provided to XBraid (default `SUNBraid_SpatialNorm()`).

Arguments:

- *app* – input, an ARKBraid instance.
- *snorm* – input, an XBraid spatial norm function. If *snorm* is `NULL`, the default function will be used.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content is `NULL`.

Note: This function must be called prior to *ARKBraid_BraidInit()*.

---

int **ARKBraid_SetAccessFn**(braid_App *app*, braid_PtFcnAccess *access*)
This function sets the user access function provided to XBraid (default *ARKBraid_Access()*).

Arguments:

- *app* – input, an ARKBraid instance.
- *init* – input, an XBraid user access function. If *access* is `NULL`, the default function will be used.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content is `NULL`.

Note: This function must be called prior to *ARKBraid_BraidInit()*.

4.8.2.3 ARKBraid Get Functions

This section describes the functions that are called by the user to retrieve data from an ARKBraid instance. Each user-callable function returns `SUNBRAID_SUCCESS` (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in *Table: SUNBraid Return Codes*.

int **ARKBraid_GetVecTmpl**(braid_App *app*, N_Vector **tmpl*)
This function returns a vector from the ARKStep memory to use as a template for creating new vectors with *N_VClone()* i.e., this is the ARKBraid implementation of `SUNBraidVector_GetVecTmpl()`.

Arguments:

- *app* – input, an ARKBraid instance.
- *tmpl* – output, a template vector.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content or ARKStep memory is `NULL`.

int **ARKBraid_GetARKStepMem**(braid_App *app*, void ***arkode_mem*)
This function returns the ARKStep memory structure pointer attached with *ARKBraid_Create()*.

Arguments:

- *app* – input, an ARKBraid instance.
- *arkode_mem* – output, a pointer to the ARKStep memory structure.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content or ARKStep memory is `NULL`.

int **ARKBraid_GetUserData**(braid_App *app*, void ***user_data*)
This function returns the user data pointer attached with *ARKStepSetUserData()*.

Arguments:

- *app* – input, an ARKBraid instance.
- *user_data* – output, a pointer to the user data structure.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content or ARKStep memory is `NULL`.

int **ARKBraid_GetLastBraidFlag**(braid_App *app*, int **last_flag*)
This function returns the return value from the most recent XBraid function call.

Arguments:

- *app* – input, an ARKBraid instance.
- *last_flag* – output, the XBraid return value.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content is `NULL`.

int **ARKBraid_GetLastARKStepFlag**(braid_App *app*, int **last_flag*)
This function returns the return value from the most recent ARKStep function call.

Arguments:

- *app* – input, an ARKBraid instance.
- *last_flag* – output, the ARKStep return value.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content is `NULL`.

int **ARKBraid_GetSolution**(braid_App *app*, realtype **tout*, N_Vector *yout*)
This function returns final time and state stored with the default access function *ARKBraid_Access()*.

Arguments:

- *app* – input, an ARKBraid instance.
- *last_flag* – output, the ARKStep return value.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content or the stored vector is `NULL`.

Warning: If providing a non-default access function the final time and state are not stored within the ARKBraid structure and this function will return an error. In this case the user should allocate space to store any desired output within the user data pointer attached to ARKStep with *ARKStepSetUserData()*. This user data pointer can be retrieved from the ARKBraid structure with *ARKBraid_GetUserData()*.

4.8.2.4 ARKBraid Interface Functions

This section describes the default XBraid interface functions provided by ARKBraid and called by XBraid to preform certain actions. Any or all of these functions may be overridden by supplying a user-defined function through the set functions defined in *ARKBraid Set Functions*. Each default interface function returns `SUNBRAID_SUCCESS` (i.e., 0) on a successful call and a negative value if an error occurred. The possible return codes are given in *Table: SUNBraid Return Codes*.

int **ARKBraid_Step**(braid_App *app*, braid_Vector *ustop*, braid_Vector *fstop*, braid_Vector *u*, braid_StepStatus *status*)
This is the default step function provided to XBraid. The step function is called by XBraid to advance the vector *u* from one time to the next using the ARStep memory structure provided to *ARKBraid_Create()*. A user-defined step function may be set with *ARKBraid_SetStepFn()*.

Arguments:

- *app* – input, an ARKBraid instance.
- *ustop* – input, *u* vector at the new time *tstop*.
- *fstop* – input, the right-hand side vector at the new time *tstop*.

- *u* - input/output, on input the vector at the start time and on return the vector at the new time.
- *status* – input, a status object to query for information about *u* and to steer XBraid e.g., for temporal refinement.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content or ARKStep memory is `NULL`.
- *SUNBRAID_BRAIDFAIL* if an XBraid function fails. The return value can be retrieved with *ARKBraid_GetLastBraidFlag()*.
- *SUNBRAID_SUNFAIL* if a SUNDIALS function fails. The return value can be retrieved with *ARKBraid_GetLastARKStepFlag()*.

Note: If providing a non-default implemenation of the step function the utility function *ARKBraid_TakeStep()* should be used to advance the input vector *u* to the new time.

---

int **ARKBraid_Init**(braid_App *app*, realtype *t*, braid_Vector **u_ptr*)

This is the default vector initialization function provided to XBraid. The initialization function is called by XBraid to create a new vector and set the initial guess for the solution at time t. When using this default function the initial guess at all time values is the initial condition provided to *ARKStepCreate()*. A user-defined init function may be set with *ARKBraid_SetInitFn()*.

Arguments:

- *app* – input, an ARKBraid instance.
- *t* – input, the initialization time for the output vector.
- *u_ptr* – output, the new and initialized SUNBraidVector.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if *app* is `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content or ARKStep memory is `NULL`.
- *SUNBRAID_ALLOCFAIL* if a memory allocation failed.

Note: If providing a non-default implementation of the vector initialization function the utility functions *SUNBraidApp_GetVecTmpl()* and *SUNBraidVector_New()* can be helpful when creating the new vector returned by this function.

---

int **ARKBraid_Access**(braid_App *app*, braid_Vector *u*, braid_AccessStatus *astatus*)

This is the default access function provided to XBraid. The access function is called by XBraid to retrieve the current solution. When using this default function the final solution time and state are stored within the ARKBraid structure. This information can be retrieved with *ARKBraid_GetSolution()*. A user-defined access function may be set with *ARKBraid_SetAccessFn()*.

Arguments:

- *app* – input, an ARKBraid instance.
- *u* – input, the vector to be accessed.

- *status* – input, a status object to query for information about *u*.

Return value:

- *SUNBRAID_SUCCESS* if successful.
- *SUNBRAID_ILLINPUT* if any of the inputs are `NULL`.
- *SUNBRAID_MEMFAIL* if the *app* content, the wrapped N_Vector, or the ARKStep memory is `NULL`.
- *SUNBRAID_ALLOCFAIL* if allocating storage for the final solution fails.
- *SUNBRAID_BRAIDFAIL* if an XBraid function fails. The return value can be retrieved with *ARKBraid_GetLastBraidFlag()*.

4.8.3 A skeleton of the user's main program with XBraid

In addition to the header files required for the integration of the ODE problem (see the section *Access to library and header files*), to use the ARKBraid interace, the user's program must include the header file `arkode/arkode_xbraid.h` which declares the needed function prototypes.

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP using ARKStep with XBraid for parallel-in-time integration. Most steps are unchanged from the skeleton program presented in *A skeleton of the user's main program*. New or updated steps are **bold**.

1. **Initialize MPI**

   If parallelizing in space and time split the global communicator into communicators for space and time with `braid_SplitCommworld()`.

2. *Set problem dimensions*
3. *Set vector of initial values*
4. *Create ARKStep object*
5. *Specify integration tolerances*
6. *Create matrix object*
7. *Create linear solver object*
8. *Set linear solver optional inputs*
9. *Attach linear solver module*
10. *Create nonlinear solver object*
11. *Attach nonlinear solver module*
12. *Set nonlinear solver optional inputs*
13. *Set optional inputs*
14. **Create ARKBraid interface**

    Call the constructor *ARKBraid_Create()* to create the XBraid app structure.

15. **Set optional ARKBraid inputs**

    See *ARKBraid Set Functions* for ARKBraid inputs.

16. **Initialize the ARKBraid interface**

    Call the initialization function `ARKBraid_Braid()` to create the XBraid core memory structure and attach the ARKBraid interface app and functions.

17. **Set optional XBraid inputs**

    See the XBraid documentation for available XBraid options.

18. **Evolve the problem**

    Call `braid_Drive()` to evolve the problem with MGRIT.

19. **Get optional outputs**

    See *ARKBraid Get Functions* for ARKBraid outputs.

20. *Deallocate memory for solution vector*

21. *Free solver memory*

22. *Free linear solver memory*

23. **Free ARKBraid and XBraid memory**

    Call *`ARKBraid_Free()`* and `braid_Destroy` to deallocate the ARKBraid interface and and XBraid core memory structures respectively.

24. *Finalize MPI*

4.8.4 Advanced ARKBraid Utility Functions

This section describes utility functions utilized in the ARKStep + XBraid interfacing. These functions are used internally by the above ARKBraid interface functions but are exposed to the user to assist in advanced usage of ARKODE and XBraid that requries defining a custom SUNBraidApp implementation.

int **`ARKBraid_TakeStep`**(void **arkode_mem*, realtype *tstart*, realtype *tstop*, N_Vector *y*, int **ark_flag*)

This function advances the vector *y* from *tstart* to *tstop* using a single ARKStep time step with step size *h* = *tstop - start*.

Arguments:

- *arkode_mem* – input, the ARKStep memory structure pointer.
- *tstart* – input, the step start time.
- *tstop* – input, the step stop time.
- *y* – input/output, on input the solution a *tstop* and on return, the solution at time *tstop* if the step was successful (*ark_flag* ≥ 0) or the solution at time *tstart* if the step failed (*ark_flag* < 0).
- *ark_flag* – output, the step status flag. If *ark_flag* is:

  $= 0$ then the step succeeded and, if applicable, met the requested temporal accuracy.

  > 0 then the step succeeded but failed to meet the requested temporal accuracy.

  < 0 then the step failed e.g., a solver failure occurred.

Return value:

If all ARKStep function calls are successful the return value is *ARK_SUCCESS*, otherwise the return value is the error flag returned from the function that failed.

Chapter 5

Using ERKStep for C and C++ Applications

This chapter is concerned with the use of the ERKStep time-stepping module for the solution of nonstiff initial value problems (IVPs) in a C or C++ language setting. The following sections discuss the header files and the layout of the user's main program, and provide descriptions of the ERKStep user-callable functions and user-supplied functions.

The example programs described in the companion document *[R2018]* may be helpful. Those codes may be used as templates for new codes and are included in the ARKode package `examples` subdirectory.

ERKStep uses the input and output constants from the shared ARKode infrastructure. These are defined as needed in this chapter, but for convenience the full list is provided separately in the section *Appendix: ARKode Constants*.

The relevant information on using ERKStep's C and C++ interfaces is detailed in the following sub-sections.

5.1 Access to library and header files

At this point, it is assumed that the installation of ARKode, following the procedure described in the section *ARKode Installation Procedure*, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the appropriate locations for the library and header files required by ARKode. The relevant library files are

- `libdir/libsundials_arkode.lib`,
- `libdir/libsundials_nvec*.lib`,

where the file extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries. The relevant header files are located in the subdirectories

- `incdir/include/arkode`
- `incdir/include/sundials`
- `incdir/include/nvector`

The directories `libdir` and `incdir` are the installation library and include directories, respectively. For a default installation, these are `instdir/lib` and `instdir/include`, respectively, where `instdir` is the directory where SUNDIALS was installed (see the section *ARKode Installation Procedure* for further details).

5.2 Data Types

The `sundials_types.h` file contains the definition of the variable type `realtype`, which is used by the SUNDIALS solvers for all floating-point data, the definition of the integer type `sunindextype`, which is used for vector and matrix indices, and `booleantype`, which is used for certain logic operations within SUNDIALS.

5.2.1 Floating point types

The type "realtype" can be set to `float`, `double`, or `long double`, depending on how SUNDIALS was installed (with the default being `double`). The user can change the precision of the SUNDIALS solvers' floating-point arithmetic at the configuration stage (see the section *Configuration options (Unix/Linux)*).

Additionally, based on the current precision, `sundials_types.h` defines the values BIG_REAL to be the largest value representable as a `realtype`, SMALL_REAL to be the smallest positive value representable as a `realtype`, and UNIT_ROUNDOFF to be the smallest realtype number, ε, such that $1.0 + \varepsilon \neq 1.0$.

Within SUNDIALS, real constants may be set to have the appropriate precision by way of a macro called RCONST. It is this macro that needs the ability to branch on the definition `realtype`. In ANSI C, a floating-point constant with no suffix is stored as a `double`. Placing the suffix "F" at the end of a floating point constant makes it a `float`, whereas using the suffix "L" makes it a `long double`. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines `A` to be a `double` constant equal to 1.0, `B` to be a `float` constant equal to 1.0, and `C` to be a `long double` constant equal to 1.0. The macro call `RCONST(1.0)` automatically expands to 1.0 if `realtype` is `double`, to `1.0F` if `realtype` is `float`, or to `1.0L` if `realtype` is `long double`. SUNDIALS uses the `RCONST` macro internally to declare all of its floating-point constants.

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent, except for any calls to precision-specific standard math library functions. Users can, however, use the types `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the size of `realtype` values that are passed to and from SUNDIALS). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries have been compiled using the same precision (for details see the section *ARKode Installation Procedure*).

5.2.2 Integer types used for vector and matrix indices

The type `sunindextype` can be either a 32- or 64-bit *signed* integer. The default is the portable `int64_t` type, and the user can change it to `int32_t` at the configuration stage. The configuration system will detect if the compiler does not support portable types, and will replace `int32_t` and `int64_t` with `int` and `long int`, respectively, to ensure use of the desired sizes on Linux, Mac OS X, and Windows platforms. SUNDIALS currently does not support *unsigned* integer types for vector and matrix indices, although these could be added in the future if there is sufficient demand.

A user program which uses `sunindextype` to handle vector and matrix indices will work with both index storage types except for any calls to index storage-specific external libraries. (Our `C` and `C++` example programs use `sunindextype`.) Users can, however, use any one of `int`, `long int`, `int32_t`, `int64_t` or `long long int` in their code, assuming that this usage is consistent with the typedef for `sunindextype` on their architecture. Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `sunindextype`, so long as the SUNDIALS libraries use the appropriate index storage type (for details see the section *ARKode Installation Procedure*).

5.3 Header Files

When using ERKStep, the calling program must include several header files so that various macros and data types can be used. The header file that is always required is:

- `arkode/arkode_erkstep.h`, the main header file for the ERKStep time-stepping module, which defines the several types and various constants, includes function prototypes, and includes the shared `arkode/arkode.h` header file.

Note that `arkode.h` includes `sundials_types.h` directly, which defines the types `realtype`, `sunindextype` and `booleantype` and the constants `SUNFALSE` and `SUNTRUE`, so a user program does not need to include `sundials_types.h` directly.

Additionally, the calling program must also include an NVECTOR implementation header file, of the form `nvector/nvector_***.h`, corresponding to the user's preferred data layout and form of parallelism. See the section *Vector Data Structures* for details for the appropriate name. This file in turn includes the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

If the user wishes to manually select between any of the pre-defined ERK Butcher tables, these are defined through a set of constants that are enumerated in the header file `arkode/arkode_butcher_erk.h`, or if a user wishes to manually specify a Butcher table, the corresponding `ARKodeButcherTable` structure is defined in `arkode/arkode_butcher.h`.

5.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP using the ERKStep module. Most of the steps are independent of the NVECTOR implementation used. For the steps that are not, refer to the section *Vector Data Structures* for the specific name of the function to be called or macro to be referenced.

1. Initialize parallel or multi-threaded environment, if appropriate.

   For example, call `MPI_Init` to initialize MPI if used, or set `num_threads`, the number of threads to use within the threaded vector functions, if used.

2. Set problem dimensions, etc.

   This generally includes the problem size, `N`, and may include the local vector length `Nlocal`.

   Note: The variables `N` and `Nlocal` should be of type `sunindextype`.

3. Set vector of initial values

   To set the vector `y0` of initial values, use the appropriate functions defined by the particular NVECTOR implementation.

   For native SUNDIALS vector implementations (except the CUDA and RAJA based ones), use a call of the form

   ```
   y0 = N_VMake_***(..., ydata);
   ```

   if the `realtype` array `ydata` containing the initial values of y already exists. Otherwise, create a new vector by making a call of the form

   ```
   y0 = N_VNew_***(...);
   ```

and then set its elements by accessing the underlying data where it is located with a call of the form

```
ydata = N_VGetArrayPointer_***(y0);
```

See the sections *The NVECTOR_SERIAL Module* through *The NVECTOR_PTHREADS Module* for details.

For the HYPRE and PETSc vector wrappers, first create and initialize the underlying vector, and then create the NVECTOR wrapper with a call of the form

```
y0 = N_VMake_***(yvec);
```

where `yvec` is a HYPRE or PETSc vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer_***(...)` are not available for these vector wrappers. See the sections *The NVECTOR_PARHYP Module* and *The NVECTOR_PETSC Module* for details.

If using either the CUDA- or RAJA-based vector implementations use a call of the form

```
y0 = N_VMake_***(..., c);
```

where `c` is a pointer to a `suncudavec` or `sunrajavec` vector class if this class already exists. Otherwise, create a new vector by making a call of the form

```
N_VGetDeviceArrayPointer_***
```

or

```
N_VGetHostArrayPointer_***
```

Note that the vector class will allocate memory on both the host and device when instantiated. See the sections *The NVECTOR_CUDA Module* and *The NVECTOR_RAJA Module* for details.

4. Create ERKStep object

   Call `arkode_mem = ERKStepCreate(...)` to create the ERKStep memory block. *`ERKStepCreate()`* returns a `void*` pointer to this memory structure. See the section *ERKStep initialization and deallocation functions* for details.

5. Specify integration tolerances

   Call *`ERKStepSStolerances()`* or *`ERKStepSVtolerances()`* to specify either a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. Alternatively, call *`ERKStepWFtolerances()`* to specify a function which sets directly the weights used in evaluating WRMS vector norms. See the section *ERKStep tolerance specification functions* for details.

6. Set optional inputs

   Call `ERKStepSet*` functions to change any optional inputs that control the behavior of ERKStep from their default values. See the section *Optional input functions* for details.

7. Specify rootfinding problem

   Optionally, call *`ERKStepRootInit()`* to initialize a rootfinding problem to be solved during the integration of the ODE system. See the section *Rootfinding initialization function* for general details, and the section *Optional input functions* for relevant optional input calls.

8. Advance solution in time

   For each point at which output is desired, call

```
ier = ERKStepEvolve(arkode_mem, tout, yout, &tret, itask);
```

Here, `itask` specifies the return mode. The vector `yout` (which can be the same as the vector `y0` above) will contain $y(t_{\text{out}})$. See the section *ERKStep solver function* for details.

9. Get optional outputs

   Call `ERKStepGet*` functions to obtain optional output. See the section *Optional output functions* for details.

10. Deallocate memory for solution vector

    Upon completion of the integration, deallocate memory for the vector `y` (or `yout`) by calling the NVECTOR destructor function:

    ```
    N_VDestroy(y);
    ```

11. Free solver memory

    Call `ERKStepFree(&arkode_mem)` to free the memory allocated for the ERKStep module.

12. Finalize MPI, if used

    Call `MPI_Finalize` to terminate MPI.

5.5 ERKStep User-callable functions

This section describes the functions that are called by the user to setup and then solve an IVP using the ERKStep time-stepping module. Some of these are required; however, starting with the section *Optional input functions*, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of ARKode's ERKStep module. In any case, refer to the preceding section, *A skeleton of the user's main program*, for the correct order of these calls.

On an error, each user-callable function returns a negative value (or `NULL` if the function returns a pointer) and sends an error message to the error handler routine, which prints the message to `stderr` by default. However, the user can set a file as error output or can provide her own error handler function (see the section *Optional input functions* for details).

5.5.1 ERKStep initialization and deallocation functions

void* **ERKStepCreate** (*ARKRhsFn f*, realtype *t0*, N_Vector *y0*)
This function allocates and initializes memory for a problem to be solved using the ERKStep time-stepping module in ARKode.

Arguments:

- *f* – the name of the C function (of type `ARKRhsFn()`) defining the right-hand side function in $\dot{y} = f(t, y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.

Return value: If successful, a pointer to initialized problem memory of type `void*`, to be passed to all user-facing ERKStep routines listed below. If unsuccessful, a `NULL` pointer will be returned, and an error message will be printed to `stderr`.

void **ERKStepFree** (void** *arkode_mem*)
This function frees the problem memory *arkode_mem* created by `ERKStepCreate()`.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.

Return value: None

5.5.2 ERKStep tolerance specification functions

These functions specify the integration tolerances. One of them **should** be called before the first call to *ERKStepEvolve()*; otherwise default values of `reltol = 1e-4` and `abstol = 1e-9` will be used, which may be entirely incorrect for a specific problem.

The integration tolerances `reltol` and `abstol` define a vector of error weights, `ewt`. In the case of *ERKStepSStolerances()*, this vector has components

```
ewt[i] = 1.0/(reltol*abs(y[i]) + abstol);
```

whereas in the case of *ERKStepSVtolerances()* the vector components are given by

```
ewt[i] = 1.0/(reltol*abs(y[i]) + abstol[i]);
```

This vector is used in all error tests, which use a weighted RMS norm on all error-like vectors v:

$$\|v\|_{WRMS} = \left(\frac{1}{N} \sum_{i=1}^{N} (v_i \; ewt_i)^2 \right)^{1/2},$$

where N is the problem dimension.

Alternatively, the user may supply a custom function to supply the `ewt` vector, through a call to *ERKStepWFtolerances()*.

int **ERKStepSStolerances**(void* *arkode_mem*, realtype *reltol*, realtype *abstol*)
This function specifies scalar relative and absolute tolerances.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – scalar absolute tolerance.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ERKStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ERKStepSVtolerances**(void* *arkode_mem*, realtype *reltol*, N_Vector *abstol*)
This function specifies a scalar relative tolerance and a vector absolute tolerance (a potentially different absolute tolerance for each vector component).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – vector containing the absolute tolerances for each solution component.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ERKStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ERKStepWFtolerances**(void* *arkode_mem*, *ARKEwtFn efun*)
This function specifies a user-supplied function *efun* to compute the error weight vector `ewt`.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *efun* – the name of the function (of type *ARKEwtFn()*) that implements the error weight vector computation.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_NO_MALLOC* if the ERKStep memory was not allocated by the time-stepping module

5.5.2.1 General advice on the choice of tolerances

For many users, the appropriate choices for tolerance values in `reltol` and `abstol` are a concern. The following pieces of advice are relevant.

1. The scalar relative tolerance `reltol` is to be set to control relative errors. So a value of 10^{-4} means that errors are controlled to .01%. We do not recommend using `reltol` larger than 10^{-3}. On the other hand, `reltol` should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 10^{-15} for double-precision).
2. The absolute tolerances `abstol` (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector y may be so small that pure relative error control is meaningless. For example, if y_i starts at some nonzero value, but in time decays to zero, then pure relative error control on y_i makes no sense (and is overly costly) after y_i is below some noise level. Then `abstol` (if scalar) or `abstol[i]` (if a vector) needs to be set to that noise level. If the different components have different noise levels, then `abstol` should be a vector. For example, see the example problem `ark_robertson.c`, and the discussion of it in the ARKode Examples Documentation *[R2018]*. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the `atols` vector therein. It is impossible to give any general advice on `abstol` values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.
3. Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual step. The final (global) errors are an accumulation of those per-step errors, where that accumulation factor is problem-dependent. A general rule of thumb is to reduce the tolerances by a factor of 10 from the actual desired limits on errors. So if you want .01% relative accuracy (globally), a good choice for `reltol` is 10^{-5}. In any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

5.5.2.2 Advice on controlling nonphysical negative values

In many applications, some components in the true solution are always positive or non-negative, though at times very small. In the numerical solution, however, small negative (nonphysical) values can then occur. In most cases, these values are harmless, and simply need to be controlled, not eliminated, but in other cases any value that violates a constraint may cause a simulation to halt. For both of these scenarios the following pieces of advice are relevant.

1. The best way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.

2. If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in y returned by ERKStep, with magnitude comparable to `abstol` or less, is equivalent to zero as far as the computation is concerned.

3. The user's right-hand side routine f should never change a negative value in the solution vector y to a non-negative value in attempt to "fix" this problem, since this can lead to numerical instability. If the f routine cannot tolerate a zero or negative value (e.g. because there is a square root or log), then the offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input y vector) for the purposes of computing $f(t, y)$.

4. Positivity and non-negativity constraints on components can be enforced by use of the recoverable error return feature in the user-supplied right-hand side function, f. When a recoverable error is encountered, ERKStep will retry the step with a smaller step size, which typically alleviates the problem. However, because this option involves some additional overhead cost, it should only be exercised if the use of absolute tolerances to control the computed values is unsuccessful.

5.5.3 Rootfinding initialization function

As described in the section *Rootfinding*, while solving the IVP, ARKode's time-stepping modules have the capability to find the roots of a set of user-defined functions. To activate the root-finding algorithm, call the following function. This is normally called only once, prior to the first call to `ERKStepEvolve()`, but if the rootfinding problem is to be changed during the solution, `ERKStepRootInit()` can also be called prior to a continuation call to `ERKStepEvolve()`.

int **`ERKStepRootInit`**(void* *arkode_mem*, int *nrtfn*, *ARKRootFn g*)

Initializes a rootfinding problem to be solved during the integration of the ODE system. It must be called after `ERKStepCreate()`, and before `ERKStepEvolve()`.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *nrtfn* – number of functions g_i, an integer ≥ 0.
- *g* – name of user-supplied function, of type `ARKRootFn()`, defining the functions g_i whose roots are sought.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_MEM_FAIL* if there was a memory allocation failure
- *ARK_ILL_INPUT* if *nrtfn* is greater than zero but *g* = `NULL`.

Notes: To disable the rootfinding feature after it has already been initialized, or to free memory associated with ERKStep's rootfinding module, call *ERKStepRootInit* with *nrtfn = 0*.

Similarly, if a new IVP is to be solved with a call to `ERKStepReInit()`, where the new IVP has no rootfinding problem but the prior one did, then call *ERKStepRootInit* with *nrtfn = 0*.

5.5.4 ERKStep solver function

This is the central step in the solution process – the call to perform the integration of the IVP. One of the input arguments (*itask*) specifies one of two modes as to where ERKStep is to return a solution. These modes are modified if the user has set a stop time (with a call to the optional input function `ERKStepSetStopTime()`) or has requested rootfinding.

int **`ERKStepEvolve`**(void* *arkode_mem*, realtype *tout*, N_Vector *yout*, realtype **tret*, int *itask*)

Integrates the ODE over an interval in t.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *tout* – the next time at which a computed solution is desired.
- *yout* – the computed solution vector.
- *tret* – the time corresponding to *yout* (output).
- *itask* – a flag indicating the job of the solver for the next user step.

  The *ARK_NORMAL* option causes the solver to take internal steps until it has just overtaken a user-specified output time, *tout*, in the direction of integration, i.e. $t_{n-1} <$ *tout* $\leq t_n$ for forward integration, or $t_n \leq$ *tout* $< t_{n-1}$ for backward integration. It will then compute an approximation to the solution $y(tout)$ by interpolation (using one of the dense output routines described in the section *Interpolation*).

  The *ARK_ONE_STEP* option tells the solver to only 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 *tout* then the solver will again return an interpolated result; otherwise it will return a copy of the internal solution y_n in the vector *yout*

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_ROOT_RETURN* if `ERKStepEvolve()` succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call `ERKStepGetRootInfo()` to see which g_i were found to have a root at (**tret*).
- *ARK_TSTOP_RETURN* if `ERKStepEvolve()` succeeded and returned at *tstop*.
- *ARK_MEM_NULL* if the *arkode_mem* argument was `NULL`.
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if one of the inputs to `ERKStepEvolve()` is illegal, or some other input to the solver was either illegal or missing. Details will be provided in the error message. Typical causes of this failure:

  1. A component of the error weight vector became zero during internal time-stepping.
  2. A root of one of the root functions was found both at a point t and also very near t.
  3. The initial condition violates the inequality constraints.

- *ARK_TOO_MUCH_WORK* if the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is *MXSTEP_DEFAULT = 500*.
- *ARK_TOO_MUCH_ACC* if the solver could not satisfy the accuracy demanded by the user for some internal step.
- *ARK_ERR_FAILURE* if error test failures occurred either too many times (*ark_maxnef*) during one internal time step or occurred with $|h| = h_{min}$.

- *ARK_VECTOROP_ERR* a vector operation error occured.

Notes: The input vector *yout* can use the same memory as the vector *y0* of initial conditions that was passed to `ERKStepCreate()`.

In *ARK_ONE_STEP* mode, *tout* is used only on the first call, and only to get the direction and a rough scale of the independent variable. All failure return values are negative and so testing the return argument for negative values will trap all `ERKStepEvolve()` failures.

Since interpolation may reduce the accuracy in the reported solution, if full method accuracy is desired the user should issue a call to `ERKStepSetStopTime()` before the call to `ERKStepEvolve()` to specify a fixed stop time to end the time step and return to the user. Upon return from `ERKStepEvolve()`, a copy of the internal solution y_n will be returned in the vector *yout*. Once the integrator returns at a *tstop* time, any future testing for *tstop* is disabled (and can be re-enabled only though a new call to `ERKStepSetStopTime()`).

On any error return in which one or more internal steps were taken by `ERKStepEvolve()`, the returned values of *tret* and *yout* correspond to the farthest point reached in the integration. On all other error returns, *tret* and *yout* are left unchanged from those provided to the routine.

5.5.5 Optional input functions

There are numerous optional input parameters that control the behavior of the ERKStep solver, each of which may be modified from its default value through calling an appropriate input function. The following tables list all optional input functions, grouped by which aspect of ERKStep they control. Detailed information on the calling syntax and arguments for each function are then provided following each table.

The optional inputs are grouped into the following categories:

- General ERKStep options (*Optional inputs for ERKStep*),
- IVP method solver options (*Optional inputs for IVP method selection*),
- Step adaptivity solver options (*Optional inputs for time step adaptivity*), and
- Rootfinding options (*Rootfinding optional input functions*).

For the most casual use of ERKStep, relying on the default set of solver parameters, the reader can skip to the following section, *User-supplied functions*.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so a test on the return arguments for negative values will catch all errors. Finally, a call to an `ERKStepSet***` function can generally be made from the user's calling program at any time and, if successful, takes effect immediately. `ERKStepSet***` functions that cannot be called at any time note this in the "**Notes**:" section of the function documentation.

5.5.5.1 Optional inputs for ERKStep

Optional input	Function name	Default
Return ERKStep solver parameters to their defaults	*ERKStepSetDefaults()*	internal
Set dense output interpolation type	*ERKStepSetInterpolantType()*	ARK_INTERP_HERMITE
Set dense output polynomial degree	*ERKStepSetInterpolantDegree()*	5
Supply a pointer to a diagnostics output file	*ERKStepSetDiagnostics()*	NULL
Supply a pointer to an error output file	*ERKStepSetErrFile()*	stderr
Supply a custom error handler function	*ERKStepSetErrHandlerFn()*	internal fn
Disable time step adaptivity (fixed-step mode)	*ERKStepSetFixedStep()*	disabled
Supply an initial step size to attempt	*ERKStepSetInitStep()*	estimated
Maximum no. of warnings for $t_n + h = t_n$	*ERKStepSetMaxHnilWarns()*	10
Maximum no. of internal steps before *tout*	*ERKStepSetMaxNumSteps()*	500
Maximum absolute step size	*ERKStepSetMaxStep()*	∞
Minimum absolute step size	*ERKStepSetMinStep()*	0.0
Set a value for t_{stop}	*ERKStepSetStopTime()*	∞
Supply a pointer for user data	*ERKStepSetUserData()*	NULL
Maximum no. of ERKStep error test failures	*ERKStepSetMaxErrTestFails()*	7
Set inequality constraints on solution	*ERKStepSetConstraints()*	NULL
Set max number of constraint failures	*ERKStepSetMaxNumConstrFails()*	10

int **ERKStepSetDefaults**(void* *arkode_mem*)

Resets all optional input parameters to ERKStep's original default values.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Does not change problem-defining function pointer f or the *user_data* pointer.

Also leaves alone any data structures or options related to root-finding (those can be reset using *ERKStepRootInit()*).

int **ERKStepSetInterpolantType**(void* *arkode_mem*, int *itype*)

Specifies use of the Lagrange or Hermite interpolation modules (used for dense output – interpolation of solution output values and implicit method predictors).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *itype* – requested interpolant type (ARK_INTERP_HERMITE or ARK_INTERP_LAGRANGE)

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is NULL
- *ARK_MEM_FAIL* if the interpolation module cannot be allocated

- *ARK_ILL_INPUT* if the *itype* argument is not recognized or the interpolation module has already been initialized

Notes: The Hermite interpolation module is described in the Section *Hermite interpolation module*, and the Lagrange interpolation module is described in the Section *Lagrange interpolation module*.

This routine frees any previously-allocated interpolation module, and re-creates one according to the specified argument. Thus any previous calls to `ERKStepSetInterpolantDegree()` will be nullified.

This routine must be called *after* the call to `ERKStepCreate()`. After the first call to `ERKStepEvolve()` the interpolation type may not be changed without first calling `ERKStepReInit()`.

If this routine is not called, the Hermite interpolation module will be used.

int **ERKStepSetInterpolantDegree**(void* *arkode_mem*, int *degree*)

Specifies the degree of the polynomial interpolant used for dense output (i.e. interpolation of solution output values and implicit method predictors).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *degree* – requested polynomial degree.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory or interpolation module are `NULL`
- *ARK_INTERP_FAIL* if this is called after `ERKStepEvolve()`
- *ARK_ILL_INPUT* if an argument has an illegal value or the interpolation module has already been initialized

Notes: Allowed values are between 0 and 5.

This routine should be called *after* `ERKStepCreate()` and *before* `ERKStepEvolve()`. After the first call to `ERKStepEvolve()` the interpolation degree may not be changed without first calling `ERKStepReInit()`.

If a user calls both this routine and `ERKStepSetInterpolantType()`, then `ERKStepSetInterpolantType()` must be called first.

Since the accuracy of any polynomial interpolant is limited by the accuracy of the time-step solutions on which it is based, the *actual* polynomial degree that is used by ERKStep will be the minimum of $q - 1$ and the input *degree*, where q is the order of accuracy for the time integration method.

int **ERKStepSetDenseOrder**(void* *arkode_mem*, int *dord*)

This function is deprecated, and will be removed in a future release. Users should transition to calling `ERKStepSetInterpolantDegree()` *instead.*

int **ERKStepSetDiagnostics**(void* *arkode_mem*, FILE* *diagfp*)

Specifies the file pointer for a diagnostics file where all ERKStep step adaptivity and solver information is written.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *diagfp* – pointer to the diagnostics output file.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This parameter can be `stdout` or `stderr`, although the suggested approach is to specify a pointer to a unique file opened by the user and returned by `fopen`. If not called, or if called with a `NULL` file pointer, all diagnostics output is disabled.

When run in parallel, only one process should set a non-NULL value for this pointer, since statistics from all processes would be identical.

int **ERKStepSetErrFile**(void* *arkode_mem*, FILE* *errfp*)
Specifies a pointer to the file where all ERKStep warning and error messages will be written if the default internal error handling function is used.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *errfp* – pointer to the output file.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value for *errfp* is `stderr`.

Passing a `NULL` value disables all future error message output (except for the case wherein the ERKStep memory pointer is `NULL`). This use of the function is strongly discouraged.

If used, this routine should be called before any other optional input functions, in order to take effect for subsequent error messages.

int **ERKStepSetErrHandlerFn**(void* *arkode_mem*, *ARKErrHandlerFn* *ehfun*, void* *eh_data*)
Specifies the optional user-defined function to be used in handling error messages.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *ehfun* – name of user-supplied error handler function.
- *eh_data* – pointer to user data passed to *ehfun* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Error messages indicating that the ERKStep solver memory is `NULL` will always be directed to `stderr`.

int **ERKStepSetFixedStep**(void* *arkode_mem*, realtype *hfixed*)
Disabled time step adaptivity within ERKStep, and specifies the fixed time step size to use for all internal steps.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hfixed* – value of the fixed step size to use.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass 0.0 to return ERKStep to the default (adaptive-step) mode.

Use of this function is not recommended, since we it gives no assurance of the validity of the computed solutions. It is primarily provided for code-to-code verification testing purposes.

When using *ERKStepSetFixedStep()*, any values provided to the functions *ERKStepSetInitStep()*, *ERKStepSetAdaptivityFn()*, *ERKStepSetMaxErrTestFails()*, *ERKStepSetAdaptivityMethod()*, *ERKStepSetCFLFraction()*, *ERKStepSetErrorBias()*, *ERKStepSetFixedStepBounds()*, *ERKStepSetMaxEFailGrowth()*, *ERKStepSetMaxFirstGrowth()*, *ERKStepSetMaxGrowth()*, *ERKStepSetMinReduction()*, *ERKStepSetSafetyFactor()*, *ERKStepSetSmallNumEFails()* and *ERKStepSetStabilityFn()* will be ignored, since temporal adaptivity is disabled.

If both *ERKStepSetFixedStep()* and *ERKStepSetStopTime()* are used, then the fixed step size will be used for all steps until the final step preceding the provided stop time (which may be shorter). To resume use of the previous fixed step size, another call to *ERKStepSetFixedStep()* must be made prior to calling *ERKStepEvolve()* to resume integration.

It is *not* recommended that *ERKStepSetFixedStep()* be used in concert with *ERKStepSetMaxStep()* or *ERKStepSetMinStep()*, since at best those latter two routines will provide no useful information to the solver, and at worst they may interfere with the desired fixed step size.

int **ERKStepSetInitStep**(void* *arkode_mem*, realtype *hin*)

Specifies the initial time step size ERKStep should use after initialization, re-initialization, or resetting.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hin* – value of the initial step to be attempted ($\neq 0$).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass 0.0 to use the default value.

By default, ERKStep estimates the initial step size to be the solution h of the equation $\left\| \frac{h^2 \ddot{y}}{2} \right\| = 1$, where $\ddot{y}$ is an estimated value of the second derivative of the solution at *t0*.

This routine will also reset the step size and error history.

int **ERKStepSetMaxHnilWarns**(void* *arkode_mem*, int *mxhnil*)

Specifies the maximum number of messages issued by the solver to warn that $t + h = t$ on the next internal step, before ERKStep will instead return with an error.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *mxhnil* – maximum allowed number of warning messages (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 10; set *mxhnil* to zero to specify this default.

A negative value indicates that no warning messages should be issued.

int **ERKStepSetMaxNumSteps**(void* *arkode_mem*, long int *mxsteps*)
Specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time, before ERKStep will return with an error.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *mxsteps* – maximum allowed number of internal steps.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Passing *mxsteps* = 0 results in ERKStep using the default value (500).

Passing *mxsteps* < 0 disables the test (not recommended).

int **ERKStepSetMaxStep**(void* *arkode_mem*, realtype *hmax*)
Specifies the upper bound on the magnitude of the time step size.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hmax* – maximum absolute value of the time step size (≥ 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass *hmax* ≤ 0.0 to set the default value of ∞.

int **ERKStepSetMinStep**(void* *arkode_mem*, realtype *hmin*)
Specifies the lower bound on the magnitude of the time step size.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hmin* – minimum absolute value of the time step size (≥ 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass *hmin* ≤ 0.0 to set the default value of 0.

int **ERKStepSetStopTime**(void* *arkode_mem*, realtype *tstop*)
Specifies the value of the independent variable t past which the solution is not to proceed.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *tstop* – stopping time for the integrator.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default is that no stop time is imposed.

int **ERKStepSetUserData**(void* *arkode_mem*, void* *user_data*)
Specifies the user data block *user_data* and attaches it to the main ERKStep memory block.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *user_data* – pointer to the user data.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: If specified, the pointer to *user_data* is passed to all user-supplied functions for which it is an argument; otherwise `NULL` is passed.

int **ERKStepSetMaxErrTestFails**(void* *arkode_mem*, int *maxnef*)
Specifies the maximum number of error test failures permitted in attempting one step, before returning with an error.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *maxnef* – maximum allowed number of error test failures (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 7; set *maxnef* ≤ 0 to specify this default.

int **ERKStepSetConstraints**(void* *arkode_mem*, N_Vector *constraints*)
Specifies a vector defining inequality constraints for each component of the solution vector y.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *constraints* – vector of constraint flags. If `constraints[i]` is
  - 0.0 then no constraint is imposed on y_i

- 1.0 then y_i will be constrained to be $y_i \geq 0.0$
- -1.0 then y_i will be constrained to be $y_i \leq 0.0$
- 2.0 then y_i will be constrained to be $y_i > 0.0$
- -2.0 then y_i will be constrained to be $y_i < 0.0$

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if the constraints vector contains illegal values

Notes: The presence of a non-`NULL` constraints vector that is not 0.0 in all components will cause constraint checking to be performed. However, a call with 0.0 in all components of `constraints` will result in an illegal input return. A `NULL` constraints vector will disable constraint checking.

After a call to *ERKStepResize()* inequality constraint checking will be disabled and a call to *ERKStepSetConstraints()* is required to re-enable constraint checking.

Since constraint-handling is performed through cutting time steps that would violate the constraints, it is possible that this feature will cause some problems to fail due to an inability to enforce constraints even at the minimum time step size. Additionally, the features *ERKStepSetConstraints()* and *ERKStepSetFixedStep()* are incompatible, and should not be used simultaneously.

int **ERKStepSetMaxNumConstrFails**(void* *arkode_mem*, int *maxfails*)
Specifies the maximum number of constraint failures in a step before ERKStep will return with an error.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *maxfails* – maximum allowed number of constrain failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`

Notes: Passing *maxfails* <= 0 results in ERKStep using the default value (10).

5.5.5.2 Optional inputs for IVP method selection

Optional input	Function name	Default
Set integrator method order	*ERKStepSetOrder()*	4
Set explicit RK table	*ERKStepSetTable()*	internal
Specify explicit RK table number	*ERKStepSetTableNum()*	internal

int **ERKStepSetOrder**(void* *arkode_mem*, int *ord*)
Specifies the order of accuracy for the ERK integration method.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *ord* – requested order of accuracy.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The allowed values are $2 \leq ord \leq 8$. Any illegal input will result in the default value of 4.

Since *ord* affects the memory requirements for the internal ERKStep memory block, it cannot be changed after the first call to `ERKStepEvolve()`, unless `ERKStepReInit()` is called.

int **ERKStepSetTable**(void* *arkode_mem*, *ARKodeButcherTable B*)
Specifies a customized Butcher table for the ERK method.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *B* – the Butcher table for the explicit RK method.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes:

For a description of the `ARKodeButcherTable` type and related functions for creating Butcher tables see *Butcher Table Data Structure*.

No error checking is performed to ensure that either the method order p or the embedding order q specified in the Butcher table structure correctly describe the coefficients in the Butcher table.

Error checking is performed to ensure that the Butcher table is strictly lower-triangular (i.e. that it specifies an ERK method).

If the Butcher table does not contain an embedding, the user *must* call `ERKStepSetFixedStep()` to enable fixed-step mode and set the desired time step size.

int **ERKStepSetTableNum**(void* *arkode_mem*, int *etable*)
Indicates to use a specific built-in Butcher table for the ERK method.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *etable* – index of the Butcher table.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: *etable* should match an existing explicit method from the section *Explicit Butcher tables*. Error-checking is performed to ensure that the table exists, and is not implicit.

5.5.5.3 Optional inputs for time step adaptivity

The mathematical explanation of ARKode's time step adaptivity algorithm, including how each of the parameters below is used within the code, is provided in the section *Time step adaptivity*.

Optional input	Function name	Default
Set a custom time step adaptivity function	`ERKStepSetAdaptivityFn()`	internal
Choose an existing time step adaptivity method	`ERKStepSetAdaptivityMethod()`	0
Explicit stability safety factor	`ERKStepSetCFLFraction()`	0.5
Time step error bias factor	`ERKStepSetErrorBias()`	1.5
Bounds determining no change in step size	`ERKStepSetFixedStepBounds()`	1.0 1.5
Maximum step growth factor on error test fail	`ERKStepSetMaxEFailGrowth()`	0.3
Maximum first step growth factor	`ERKStepSetMaxFirstGrowth()`	10000.0
Maximum allowed general step growth factor	`ERKStepSetMaxGrowth()`	20.0
Minimum allowed step reduction factor on error test fail	`ERKStepSetMinReduction()`	0.1
Time step safety factor	`ERKStepSetSafetyFactor()`	0.96
Error fails before MaxEFailGrowth takes effect	`ERKStepSetSmallNumEFails()`	2
Explicit stability function	`ERKStepSetStabilityFn()`	none

int **ERKStepSetAdaptivityFn**(void* *arkode_mem*, *ARKAdaptFn* *hfun*, void* *h_data*)

Sets a user-supplied time-step adaptivity function.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hfun* – name of user-supplied adaptivity function.
- *h_data* – pointer to user data passed to *hfun* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This function should focus on accuracy-based time step estimation; for stability based time steps the function `ERKStepSetStabilityFn()` should be used instead.

int **ERKStepSetAdaptivityMethod**(void* *arkode_mem*, int *imethod*, int *idefault*, int *pq*, realtype* *adapt_params*)

Specifies the method (and associated parameters) used for time step adaptivity.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *imethod* – accuracy-based adaptivity method choice ($0 \leq$ *imethod* ≤ 5): 0 is PID, 1 is PI, 2 is I, 3 is explicit Gustafsson, 4 is implicit Gustafsson, and 5 is the ImEx Gustafsson.
- *idefault* – flag denoting whether to use default adaptivity parameters (1), or that they will be supplied in the *adapt_params* argument (0).
- *pq* – flag denoting whether to use the embedding order of accuracy p (0) or the method order of accuracy q (1) within the adaptivity algorithm. p is the default.
- *adapt_params[0]* – k_1 parameter within accuracy-based adaptivity algorithms.
- *adapt_params[1]* – k_2 parameter within accuracy-based adaptivity algorithms.
- *adapt_params[2]* – k_3 parameter within accuracy-based adaptivity algorithms.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ERKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: If custom parameters are supplied, they will be checked for validity against published stability intervals. If other parameter values are desired, it is recommended to instead provide a custom function through a call to *ERKStepSetAdaptivityFn()*.

int **ERKStepSetCFLFraction**(void* *arkode_mem*, realtype *cfl_frac*)
Specifies the fraction of the estimated explicitly stable step to use.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *cfl_frac* – maximum allowed fraction of explicitly stable step (default is 0.5).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ERKStepSetErrorBias**(void* *arkode_mem*, realtype *bias*)
Specifies the bias to be applied to the error estimates within accuracy-based adaptivity strategies.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *bias* – bias applied to error in accuracy-based time step estimation (default is 1.5).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value below 1.0 will imply a reset to the default value.

int **ERKStepSetFixedStepBounds**(void* *arkode_mem*, realtype *lb*, realtype *ub*)
Specifies the step growth interval in which the step size will remain unchanged.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *lb* – lower bound on window to leave step size fixed (default is 1.0).
- *ub* – upper bound on window to leave step size fixed (default is 1.5).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any interval *not* containing 1.0 will imply a reset to the default values.

int **ERKStepSetMaxEFailGrowth**(void* *arkode_mem*, realtype *etamxf*)

Specifies the maximum step size growth factor upon multiple successive accuracy-based error failures in the solver.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *etamxf* – time step reduction factor on multiple error fails (default is 0.3).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value outside the interval $(0, 1]$ will imply a reset to the default value.

int **ERKStepSetMaxFirstGrowth**(void* *arkode_mem*, realtype *etamx1*)

Specifies the maximum allowed growth factor in step size following the very first integration step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *etamx1* – maximum allowed growth factor after the first time step (default is 10000.0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value ≤ 1.0 will imply a reset to the default value.

int **ERKStepSetMaxGrowth**(void* *arkode_mem*, realtype *mx_growth*)

Specifies the maximum allowed growth factor in step size between consecutive steps in the integration process.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *mx_growth* – maximum allowed growth factor between consecutive time steps (default is 20.0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value ≤ 1.0 will imply a reset to the default value.

int **ERKStepSetMinReduction**(void* *arkode_mem*, realtype *eta_min*)

Specifies the minimum allowed reduction factor in step size between step attempts, resulting from a temporal error failure in the integration process.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *eta_min* – minimum allowed reduction factor time step after an error test failure (default is 0.1).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any value ≥ 1.0 or ≤ 0.0 will imply a reset to the default value.

int **ERKStepSetSafetyFactor**(void* *arkode_mem*, realtype *safety*)
Specifies the safety factor to be applied to the accuracy-based estimated step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *safety* – safety factor applied to accuracy-based time step (default is 0.96).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ERKStepSetSmallNumEFails**(void* *arkode_mem*, int *small_nef*)
Specifies the threshold for "multiple" successive error failures before the *etamxf* parameter from *ERKStepSetMaxEFailGrowth()* is applied.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *small_nef* – bound to determine 'multiple' for *etamxf* (default is 2).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **ERKStepSetStabilityFn**(void* *arkode_mem*, *ARKExpStabFn* *EStab*, void* *estab_data*)
Sets the problem-dependent function to estimate a stable time step size for the explicit portion of the ODE system.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *EStab* – name of user-supplied stability function.
- *estab_data* – pointer to user data passed to *EStab* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This function should return an estimate of the absolute value of the maximum stable time step for the the ODE system. It is not required, since accuracy-based adaptivity may be sufficient for retaining stability, but this can be quite useful for problems where the right-hand side function $f(t, y)$ may contain stiff terms.

5.5.5.4 Rootfinding optional input functions

The following functions can be called to set optional inputs to control the rootfinding algorithm, the mathematics of which are described in the section *Rootfinding*.

Optional input	Function name	Default
Direction of zero-crossings to monitor	`ERKStepSetRootDirection()`	both
Disable inactive root warnings	`ERKStepSetNoInactiveRootWarn()`	enabled

int **ERKStepSetRootDirection**(void* *arkode_mem*, int* *rootdir*)
Specifies the direction of zero-crossings to be located and returned.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *rootdir* – state array of length *nrtfn*, the number of root functions g_i (the value of *nrtfn* was supplied in the call to `ERKStepRootInit()`). If `rootdir[i] == 0` then crossing in either direction for g_i should be reported. A value of +1 or -1 indicates that the solver should report only zero-crossings where g_i is increasing or decreasing, respectively.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default behavior is to monitor for both zero-crossing directions.

int **ERKStepSetNoInactiveRootWarn**(void* *arkode_mem*)
Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`

Notes: ERKStep will not report the initial conditions as a possible zero-crossing (assuming that one or more components g_i are zero at the initial time). However, if it appears that some g_i is identically zero at the initial time (i.e., g_i is zero at the initial time *and* after the first step), ERKStep will issue a warning which can be disabled with this optional input function.

5.5.6 Interpolated output function

An optional function `ERKStepGetDky()` is available to obtain additional values of solution-related quantities. This function should only be called after a successful return from `ERKStepEvolve()`, as it provides interpolated values either of y or of its derivatives (up to the 5th derivative) interpolated to any value of t in the last internal step

taken by `ERKStepEvolve()`. Internally, this *dense output* algorithm is identical to the algorithm used for the maximum order implicit predictors, described in the section *Maximum order predictor*, except that derivatives of the polynomial model may be evaluated upon request.

int **ERKStepGetDky**(void* *arkode_mem*, realtype *t*, int *k*, N_Vector *dky*)

Computes the *k*-th derivative of the function y at the time *t*, i.e. $\frac{d^{(k)}}{dt^{(k)}}y(t)$, for values of the independent variable satisfying $t_n - h_n \le t \le t_n$, with t_n as current internal time reached, and h_n is the last internal step size successfully used by the solver. This routine uses an interpolating polynomial of degree *min(degree, 5)*, where *degree* is the argument provided to `ERKStepSetInterpolantDegree()`. The user may request *k* in the range {0,. . . ,*min(degree, kmax)*} where *kmax* depends on the choice of interpolation module. For Hermite interpolants *kmax = 5* and for Lagrange interpolants *kmax = 3*.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *t* – the value of the independent variable at which the derivative is to be evaluated.
- *k* – the derivative order requested.
- *dky* – output vector (must be allocated by the user).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_BAD_K* if *k* is not in the range {0,. . . ,*min(degree, kmax)*}.
- *ARK_BAD_T* if *t* is not in the interval $[t_n - h_n, t_n]$
- *ARK_BAD_DKY* if the *dky* vector was `NULL`
- *ARK_MEM_NULL* if the ERKStep memory is `NULL`

Notes: It is only legal to call this function after a successful return from `ERKStepEvolve()`.

A user may access the values t_n and h_n via the functions `ERKStepGetCurrentTime()` and `ERKStepGetLastStep()`, respectively.

5.5.7 Optional output functions

ERKStep provides an extensive set of functions that can be used to obtain solver performance information. We organize these into groups:

1. SUNDIALS version information accessor routines are in the subsection *SUNDIALS version information*,
2. General ERKStep output routines are in the subsection *Main solver optional output functions*,
3. Output routines regarding root-finding results are in the subsection *Rootfinding optional output functions*,
4. General usability routines (e.g. to print the current ERKStep parameters, or output the current Butcher table) are in the subsection *General usability functions*.

Following each table, we elaborate on each function.

Some of the optional outputs, especially the various counters, can be very useful in determining the efficiency of various methods inside ERKStep. For example:

- The counters *nsteps* and *nf_evals* provide a rough measure of the overall cost of a given run, and can be compared between runs with different solver options to suggest which set of options is the most efficient.
- The ratio *nsteps/step_attempts* can measure the quality of the time step adaptivity algorithm, since a poor algorithm will result in more failed steps, and hence a lower ratio.

It is therefore recommended that users retrieve and output these statistics following each run, and take some time to investigate alternate solver options that will be more optimal for their particular problem of interest.

5.5.7.1 SUNDIALS version information

The following functions provide a way to get SUNDIALS version information at runtime.

int **SUNDIALSGetVersion**(char **version*, int *len*)
This routine fills a string with SUNDIALS version information.

Arguments:

- *version* – character array to hold the SUNDIALS version information.
- *len* – allocated length of the *version* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS version

Notes: An array of 25 characters should be sufficient to hold the version information.

int **SUNDIALSGetVersionNumber**(int **major*, int **minor*, int **patch*, char **label*, int *len*)
This routine sets integers for the SUNDIALS major, minor, and patch release numbers and fills a string with the release label if applicable.

Arguments:

- *major* – SUNDIALS release major version number.
- *minor* – SUNDIALS release minor version number.
- *patch* – SUNDIALS release patch version number.
- *label* – string to hold the SUNDIALS release label.
- *len* – allocated length of the *label* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS label

Notes: An array of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to *label*.

5.5.7.2 Main solver optional output functions

Optional output	Function name
Size of ERKStep real and integer workspaces	*ERKStepGetWorkSpace()*
Cumulative number of internal steps	*ERKStepGetNumSteps()*
Actual initial time step size used	*ERKStepGetActualInitStep()*
Step size used for the last successful step	*ERKStepGetLastStep()*
Step size to be attempted on the next step	*ERKStepGetCurrentStep()*
Current internal time reached by the solver	*ERKStepGetCurrentTime()*
Suggested factor for tolerance scaling	*ERKStepGetTolScaleFactor()*
Error weight vector for state variables	*ERKStepGetErrWeights()*
Single accessor to many statistics at once	*ERKStepGetStepStats()*
Name of constant associated with a return flag	*ERKStepGetReturnFlagName()*
No. of explicit stability-limited steps	*ERKStepGetNumExpSteps()*
No. of accuracy-limited steps	*ERKStepGetNumAccSteps()*
No. of attempted steps	*ERKStepGetNumStepAttempts()*
No. of calls to f function	*ERKStepGetNumRhsEvals()*
No. of local error test failures that have occurred	*ERKStepGetNumErrTestFails()*
Current ERK Butcher table	*ERKStepGetCurrentButcherTable()*
Estimated local truncation error vector	*ERKStepGetEstLocalErrors()*
Single accessor to many statistics at once	*ERKStepGetTimestepperStats()*
Number of constraint test failures	*ERKStepGetNumConstrFails()*

int **ERKStepGetWorkSpace**(void* *arkode_mem*, long int* *lenrw*, long int* *leniw*)
Returns the ERKStep real and integer workspace sizes.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *lenrw* – the number of `realtype` values in the ERKStep workspace.
- *leniw* – the number of integer values in the ERKStep workspace.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetNumSteps**(void* *arkode_mem*, long int* *nsteps*)
Returns the cumulative number of internal steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *nsteps* – number of steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetActualInitStep**(void* *arkode_mem*, realtype* *hinused*)
Returns the value of the integration step size used on the first step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.

- *hinused* – actual value of initial step size.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: Even if the value of the initial integration step was specified by the user through a call to `ERKStepSetInitStep()`, this value may have been changed by ERKStep to ensure that the step size fell within the prescribed bounds ($h_{min} \leq h_0 \leq h_{max}$), or to satisfy the local error test condition.

int **ERKStepGetLastStep**(void* *arkode_mem*, realtype* *hlast*)
Returns the integration step size taken on the last successful internal step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hlast* – step size taken on the last internal step.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetCurrentStep**(void* *arkode_mem*, realtype* *hcur*)
Returns the integration step size to be attempted on the next internal step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *hcur* – step size to be attempted on the next internal step.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetCurrentTime**(void* *arkode_mem*, realtype* *tcur*)
Returns the current internal time reached by the solver.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *tcur* – current internal time reached.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetTolScaleFactor**(void* *arkode_mem*, realtype* *tolsfac*)
Returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *tolsfac* – suggested scaling factor for user-supplied tolerances.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetErrWeights**(void* *arkode_mem*, N_Vector *eweight*)
Returns the current error weight vector.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *eweight* – solution error weights at the current time.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: The user must allocate space for *eweight*, that will be filled in by this function.

int **ERKStepGetStepStats**(void* *arkode_mem*, long int* *nsteps*, realtype* *hinused*, realtype* *hlast*, realtype* *hcur*, realtype* *tcur*)
Returns many of the most useful optional outputs in a single call.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *nsteps* – number of steps taken in the solver.
- *hinused* – actual value of initial step size.
- *hlast* – step size taken on the last internal step.
- *hcur* – step size to be attempted on the next internal step.
- *tcur* – current internal time reached.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

char ***ERKStepGetReturnFlagName**(long int *flag*)
Returns the name of the ERKStep constant corresponding to *flag*.

Arguments:

- *flag* – a return flag from an ERKStep function.

Return value: The return value is a string containing the name of the corresponding constant.

int **ERKStepGetNumExpSteps**(void* *arkode_mem*, long int* *expsteps*)
Returns the cumulative number of stability-limited steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetNumAccSteps**(void* *arkode_mem*, long int* *accsteps*)
Returns the cumulative number of accuracy-limited steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *accsteps* – number of accuracy-limited steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetNumStepAttempts**(void* *arkode_mem*, long int* *step_attempts*)
Returns the cumulative number of steps attempted by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *step_attempts* – number of steps attempted by solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetNumRhsEvals**(void* *arkode_mem*, long int* *nf_evals*)
Returns the number of calls to the user's right-hand side function, f (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *nf_evals* – number of calls to the user's $f(t, y)$ function.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetNumErrTestFails**(void* *arkode_mem*, long int* *netfails*)
Returns the number of local error test failures that have occurred (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *netfails* – number of error test failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetCurrentButcherTable**(void* *arkode_mem*, *ARKodeButcherTable* **B*)
Returns the Butcher table currently in use by the solver.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *B* – pointer to Butcher table structure.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: The *ARKodeButcherTable* data structure is defined as a pointer to the following C structure:

```
typedef struct ARKodeButcherTableMem {

  int q;           /* method order of accuracy       */
  int p;           /* embedding order of accuracy    */
  int stages;      /* number of stages               */
  realtype **A;    /* Butcher table coefficients     */
  realtype *c;     /* canopy node coefficients       */
  realtype *b;     /* root node coefficients         */
  realtype *d;     /* embedding coefficients         */

} *ARKodeButcherTable;
```

For more details see *Butcher Table Data Structure*.

int **ERKStepGetEstLocalErrors**(void* *arkode_mem*, N_Vector *ele*)

Returns the vector of estimated local truncation errors for the current step.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *ele* – vector of estimated local truncation errors.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: The user must allocate space for *ele*, that will be filled in by this function.

The values returned in *ele* are valid only after a successful call to *ERKStepEvolve()* (i.e. it returned a non-negative value).

The *ele* vector, together with the *eweight* vector from *ERKStepGetErrWeights()*, can be used to determine how the various components of the system contributed to the estimated local error test. Specifically, that error test uses the WRMS norm of a vector whose components are the products of the components of these two vectors. Thus, for example, if there were recent error test failures, the components causing the failures are those with largest values for the products, denoted loosely as `eweight[i]*ele[i]`.

int **ERKStepGetTimestepperStats**(void* *arkode_mem*, long int* *expsteps*, long int* *accsteps*, long int* *step_attempts*, long int* *nf_evals*, long int* *netfails*)

Returns many of the most useful time-stepper statistics in a single call.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *expsteps* – number of stability-limited steps taken in the solver.
- *accsteps* – number of accuracy-limited steps taken in the solver.
- *step_attempts* – number of steps attempted by the solver.
- *nf_evals* – number of calls to the user's $f(t, y)$ function.
- *netfails* – number of error test failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

int **ERKStepGetNumConstrFails**(void* *arkode_mem*, long int* *nconstrfails*)
Returns the cumulative number of constraint test failures (so far).

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *nconstrfails* – number of constraint test failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

5.5.7.3 Rootfinding optional output functions

Optional output	Function name
Array showing roots found	*ERKStepGetRootInfo()*
No. of calls to user root function	*ERKStepGetNumGEvals()*

int **ERKStepGetRootInfo**(void* *arkode_mem*, int* *rootsfound*)
Returns an array showing which functions were found to have a root.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *rootsfound* – array of length *nrtfn* with the indices of the user functions g_i found to have a root (the value of *nrtfn* was supplied in the call to *ERKStepRootInit()*). For $i = 0 \ldots$ *nrtfn*-1, `rootsfound[i]` is nonzero if g_i has a root, and 0 if not.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: The user must allocate space for *rootsfound* prior to calling this function.

For the components of g_i for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of +1 indicates that g_i is increasing, while a value of -1 indicates a decreasing g_i.

int **ERKStepGetNumGEvals**(void* *arkode_mem*, long int* *ngevals*)
Returns the cumulative number of calls made to the user's root function g.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *ngevals* – number of calls made to g so far.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

5.5.7.4 General usability functions

The following optional routines may be called by a user to inquire about existing solver parameters, to retrieve stored Butcher tables, write the current Butcher table, or even to test a provided Butcher table to determine its analytical order of accuracy. While none of these would typically be called during the course of solving an initial value problem, these may be useful for users wishing to better understand ERKStep and/or specific Runge-Kutta methods.

Optional routine	Function name
Output all ERKStep solver parameters	*ERKStepWriteParameters()*
Output the current Butcher table	*ERKStepWriteButcher()*

int **ERKStepWriteParameters**(void* *arkode_mem*, FILE **fp*)
Outputs all ERKStep solver parameters to the provided file pointer.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *fp* – pointer to use for printing the solver parameters.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: The *fp* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

When run in parallel, only one process should set a non-NULL value for this pointer, since parameters for all processes would be identical.

int **ERKStepWriteButcher**(void* *arkode_mem*, FILE **fp*)
Outputs the current Butcher table to the provided file pointer.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *fp* – pointer to use for printing the Butcher table.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`

Notes: The *fp* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

When run in parallel, only one process should set a non-NULL value for this pointer, since tables for all processes would be identical.

5.5.8 ERKStep re-initialization function

To reinitialize the ERKStep module for the solution of a new problem, where a prior call to *ERKStepCreate()* has been made, the user must call the function *ERKStepReInit()*. The new problem must have the same size as the previous one. This routine retains the current settings for all ERKstep module options and performs the same input checking and initializations that are done in *ERKStepCreate()*, but it performs no memory allocation as is assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration. Following a successful call to *ERKStepReInit()*, call *ERKStepEvolve()* again for the solution of the new problem.

The use of *ERKStepReInit()* requires that the number of Runge Kutta stages, denoted by *s*, be no larger for the new problem than for the previous problem. This condition is automatically fulfilled if the method order *q* is left unchanged.

One important use of the *ERKStepReInit()* function is in the treating of jump discontinuities in the RHS function. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS function *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS function (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ERKStepReInit**(void* *arkode_mem*, *ARKRhsFn f*, realtype *t0*, N_Vector *y0*)

Provides required problem specifications and re-initializes the ERKStep time-stepper module.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- f – the name of the C function (of type *ARKRhsFn()*) defining the right-hand side function in $\dot{y} = f(t, y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: All previously set options are retained but may be updated by calling the appropriate "Set" functions.

If an error occurred, *ERKStepReInit()* also sends an error message to the error handler function.

5.5.9 ERKStep reset function

To reset the ERKStep module to a particular independent variable value and dependent variable vector for the continued solution of a problem, where a prior call to *ERKStepCreate()* has been made, the user must call the function *ERKStepReset()*. Like *ERKStepReInit()* this routine retains the current settings for all ERKStep module options and performs no memory allocations but, unlike *ERKStepReInit()*, this routine performs only a *subset* of the input checking and initializations that are done in *ERKStepCreate()*. In particular this routine retains all internal counter values and the step size/error history. Following a successful call to *ERKStepReset()*, call *ERKStepEvolve()* again to continue solving the problem. By default the next call to *ERKStepEvolve()* will use the step size computed by ERKStep prior to calling *ERKStepReset()*. To set a different step size or have ERKStep estimate a new step size use *ERKStepSetInitStep()*.

One important use of the *ERKStepReset()* function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to *ERKStepReset()*. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and

subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **ERKStepReset**(void* *arkode_mem*, realtype *tR*, N_Vector *yR*)

Resets the current ERKStep time-stepper module state to the provided independent variable value and dependent variable vector.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.
- *tR* – the value of the independent variable t.
- *yR* – the value of the dependent variable vector $y(t_R)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: By default the next call to *ERKStepEvolve()* will use the step size computed by ERKStep prior to calling *ERKStepReset()*. To set a different step size or have ERKStep estimate a new step size use *ERKStepSetInitStep()*.

All previously set options are retained but may be updated by calling the appropriate "Set" functions.

If an error occurred, *ERKStepReset()* also sends an error message to the error handler function.

5.5.10 ERKStep system resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatially-adaptive PDE simulations under a method-of-lines approach), the ERKStep integrator may be "resized" between integration steps, through calls to the *ERKStepResize()* function. This function modifies ERKStep's internal memory structures to use the new problem size, without destruction of the temporal adaptivity heuristics. It is assumed that the dynamical time scales before and after the vector resize will be comparable, so that all time-stepping heuristics prior to calling *ERKStepResize()* remain valid after the call. If instead the dynamics should be recomputed from scratch, the ERKStep memory structure should be deleted with a call to *ERKStepFree()*, and recreated with a call to *ERKStepCreate()*.

To aid in the vector resize operation, the user can supply a vector resize function that will take as input a vector with the previous size, and transform it in-place to return a corresponding vector of the new size. If this function (of type *ARKVecResizeFn()*) is not supplied (i.e. is set to `NULL`), then all existing vectors internal to ERKStep will be destroyed and re-cloned from the new input vector.

In the case that the dynamical time scale should be modified slightly from the previous time scale, an input *hscale* is allowed, that will rescale the upcoming time step by the specified factor. If a value *hscale* $\leq$ 0 is specified, the default of 1.0 will be used.

int **ERKStepResize**(void* *arkode_mem*, N_Vector *ynew*, realtype *hscale*, realtype *t0*, *ARKVecResizeFn* *resize*, void* *resize_data*)

Re-initializes ERKStep with a different state vector but with comparable dynamical time scale.

Arguments:

- *arkode_mem* – pointer to the ERKStep memory block.

- *ynew* – the newly-sized solution vector, holding the current dependent variable values $y(t_0)$.
- *hscale* – the desired scaling factor for the dynamical time scale (i.e. the next step will be of size *h*hscale*).
- *t0* – the current value of the independent variable t_0 (this must be consistent with *ynew*).
- *resize* – the user-supplied vector resize function (of type `ARKVecResizeFn()`.
- *resize_data* – the user-supplied data structure to be passed to *resize* when modifying internal ERKStep vectors.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ERKStep memory was `NULL`
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: If an error occurred, `ERKStepResize()` also sends an error message to the error handler function.

If inequality constraint checking is enabled a call to `ERKStepResize()` will disable constraint checking. A call to `ERKStepSetConstraints()` is required to re-enable constraint checking.

5.5.10.1 Resizing the absolute tolerance array

If using array-valued absolute tolerances, the absolute tolerance vector will be invalid after the call to `ERKStepResize()`, so the new absolute tolerance vector should be re-set **following** each call to `ERKStepResize()` through a new call to `ERKStepSVtolerances()`.

If scalar-valued tolerances or a tolerance function was specified through either `ERKStepSStolerances()` or `ERKStepWFtolerances()`, then these will remain valid and no further action is necessary.

Note: For an example showing usage of the similar `ARKStepResize()` routine, see the supplied serial C example problem, `ark_heat1D_adapt.c`.

5.6 User-supplied functions

The user-supplied functions for ERKStep consist of:

- a function that defines the ODE (required),
- a function that handles error and warning messages (optional),
- a function that provides the error weight vector (optional),
- a function that handles adaptive time step error control (optional),
- a function that handles explicit time step stability (optional),
- a function that defines the root-finding problem(s) to solve (optional),
- a function that handles vector resizing operations, if the underlying vector structure supports resizing (as opposed to deletion/recreation), and if the user plans to call `ERKStepResize()` (optional).

5.6.1 ODE right-hand side

The user must supply a function of type *ARKRhsFn* to specify the right-hand side of the ODE system:

typedef int **(*ARKRhsFn)**(realtype *t*, N_Vector *y*, N_Vector *ydot*, void* *user_data*)

This function computes the ODE right-hand side for a given value of the independent variable t and state vector y.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *ydot* – the output vector that forms the ODE RHS $f(t, y)$.
- *user_data* – the *user_data* pointer that was passed to *ERKStepSetUserData()*.

Return value: An *ARKRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ERKStep will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *ARK_RHSFUNC_FAIL* is returned).

Notes: Allocation of memory for *ydot* is handled within the ERKStep module.

The vector *ydot* may be uninitialized on input; it is the user's responsibility to fill this entire vector with meaningful values.

A recoverable failure error return from the *ARKRhsFn* is typically used to flag a value of the dependent variable y that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made, ERKStep will attempt to recover by reducing the step size in order to avoid this recoverable error return. There are some situations in which recovery is not possible even if the right-hand side function returns a recoverable error flag. One is when this occurs at the very first call to the *ARKRhsFn* (in which case ERKStep returns *ARK_FIRST_RHSFUNC_ERR*).

5.6.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by *errfp* (see *ERKStepSetErrFile()*), the user may provide a function of type *ARKErrHandlerFn* to process any such messages.

typedef void **(*ARKErrHandlerFn)**(int *error_code*, const char* *module*, const char* *function*, char* *msg*, void* *user_data*)

This function processes error and warning messages from ERKStep and its sub-modules.

Arguments:

- *error_code* – the error code.
- *module* – the name of the ERKStep module reporting the error.
- *function* – the name of the function in which the error occurred.
- *msg* – the error message.
- *user_data* – a pointer to user data, the same as the *eh_data* parameter that was passed to *ERKStepSetErrHandlerFn()*.

Return value: An *ARKErrHandlerFn* function has no return value.

Notes: *error_code* is negative for errors and positive (*ARK_WARNING*) for warnings. If a function that returns a pointer to memory encounters an error, it sets *error_code* to 0.

5.6.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `ARKEwtFn` to compute a vector *ewt* containing the weights in the WRMS norm $\|v\|_{WRMS} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(ewt_i\ v_i\right)^2\right)^{1/2}$. These weights will be used in place of those defined in the section *Error norms*.

typedef int **(*ARKEwtFn)**(N_Vector *y*, N_Vector *ewt*, void* *user_data*)

This function computes the WRMS error weights for the vector y.

Arguments:

- *y* – the dependent variable vector at which the weight vector is to be computed.
- *ewt* – the output vector containing the error weights.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ERKStepSetUserData()`.

Return value: An *ARKEwtFn* function must return 0 if it successfully set the error weights, and -1 otherwise.

Notes: Allocation of memory for *ewt* is handled within ERKStep.

The error weight vector must have all components positive. It is the user's responsibility to perform this test and return -1 if it is not satisfied.

5.6.4 Time step adaptivity function

As an alternative to using one of the built-in time step adaptivity methods for controlling solution error, the user may provide a function of type `ARKAdaptFn` to compute a target step size h for the next integration step. These steps should be chosen as the maximum value such that the error estimates remain below 1.

typedef int **(*ARKAdaptFn)**(N_Vector *y*, realtype *t*, realtype *h1*, realtype *h2*, realtype *h3*, realtype *e1*, realtype *e2*, realtype *e3*, int *q*, int *p*, realtype* *hnew*, void* *user_data*)

This function implements a time step adaptivity algorithm that chooses h satisfying the error tolerances.

Arguments:

- *y* – the current value of the dependent variable vector.
- *t* – the current value of the independent variable.
- *h1* – the current step size, $t_n - t_{n-1}$.
- *h2* – the previous step size, $t_{n-1} - t_{n-2}$.
- *h3* – the step size $t_{n-2} - t_{n-3}$.
- *e1* – the error estimate from the current step, n.
- *e2* – the error estimate from the previous step, $n-1$.
- *e3* – the error estimate from the step $n-2$.
- *q* – the global order of accuracy for the method.
- *p* – the global order of accuracy for the embedded method.
- *hnew* – the output value of the next step size.
- *user_data* – a pointer to user data, the same as the *h_data* parameter that was passed to `ERKStepSetAdaptivityFn()`.

Return value: An *ARKAdaptFn* function should return 0 if it successfully set the next step size, and a non-zero value otherwise.

5.6.5 Explicit stability function

A user may supply a function to predict the maximum stable step size for the explicit Runge Kutta method on this problem. While the accuracy-based time step adaptivity algorithms may be sufficient for retaining a stable solution to the ODE system, these may be inefficient if $f(t, y)$ contains moderately stiff terms. In this scenario, a user may provide a function of type `ARKExpStabFn` to provide this stability information to ERKStep. This function must set the scalar step size satisfying the stability restriction for the upcoming time step. This value will subsequently be bounded by the user-supplied values for the minimum and maximum allowed time step, and the accuracy-based time step.

typedef int **(*ARKExpStabFn)**(N_Vector *y*, realtype *t*, realtype* *hstab*, void* *user_data*)

This function predicts the maximum stable step size for the ODE system.

Arguments:

- *y* – the current value of the dependent variable vector.
- *t* – the current value of the independent variable.
- *hstab* – the output value with the absolute value of the maximum stable step size.
- *user_data* – a pointer to user data, the same as the *estab_data* parameter that was passed to `ERKStepSetStabilityFn()`.

Return value: An *ARKExpStabFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.

Notes: If this function is not supplied, or if it returns *hstab* ≤ 0.0, then ERKStep will assume that there is no explicit stability restriction on the time step size.

5.6.6 Rootfinding function

If a rootfinding problem is to be solved during the integration of the ODE system, the user must supply a function of type `ARKRootFn`.

typedef int **(*ARKRootFn)**(realtype *t*, N_Vector *y*, realtype* *gout*, void* *user_data*)

This function implements a vector-valued function $g(t, y)$ such that the roots of the *nrtfn* components $g_i(t, y)$ are sought.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *gout* – the output array, of length *nrtfn*, with components $g_i(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ERKStepSetUserData()`.

Return value: An *ARKRootFn* function should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and ERKStep returns *ARK_RTFUNC_FAIL*).

Notes: Allocation of memory for *gout* is handled within ERKStep.

5.6.7 Vector resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatial adaptivity in a PDE simulation), the ERKStep integrator may be "resized" between integration steps, through calls to the `ERKStepResize()` function. Typically, when performing adaptive simulations the solution is stored in

a customized user-supplied data structure, to enable adaptivity without repeated allocation/deallocation of memory. In these scenarios, it is recommended that the user supply a customized vector kernel to interface between SUNDIALS and their problem-specific data structure. If this vector kernel includes a function of type *ARKVecResizeFn* to resize a given vector implementation, then this function may be supplied to *ERKStepResize()* so that all internal ERKStep vectors may be resized, instead of deleting and re-creating them at each call. This resize function should have the following form:

typedef int **(*ARKVecResizeFn)**(N_Vector *y*, N_Vector *ytemplate*, void* *user_data*)

This function resizes the vector *y* to match the dimensions of the supplied vector, *ytemplate*.

Arguments:

- *y* – the vector to resize.
- *ytemplate* – a vector of the desired size.
- *user_data* – a pointer to user data, the same as the *resize_data* parameter that was passed to *ERKStepResize()*.

Return value: An *ARKVecResizeFn* function should return 0 if it successfully resizes the vector *y*, and a non-zero value otherwise.

Notes: If this function is not supplied, then ERKStep will instead destroy the vector *y* and clone a new vector *y* off of *ytemplate*.

Chapter 6

Using MRIStep for C and C++ Applications

This chapter is concerned with the use of the MRIStep time-stepping module for the solution of two-rate initial value problems (IVPs) in a C or C++ language setting. The following sections discuss the header files and the layout of the user's main program, and provide descriptions of the MRIStep user-callable functions and user-supplied functions.

The example programs described in the companion document *[R2018]* may be helpful. Those codes may be used as templates for new codes and are included in the ARKode package `examples` subdirectory.

MRIStep uses the input and output constants from the shared ARKode infrastructure. These are defined as needed in this chapter, but for convenience the full list is provided separately in the section *Appendix: ARKode Constants*.

The relevant information on using MRIStep's C and C++ interfaces is detailed in the following sub-sections.

6.1 Access to library and header files

At this point, it is assumed that the installation of ARKode, following the procedure described in the section *ARKode Installation Procedure*, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the appropriate locations for the library and header files required by ARKode. The relevant library files are

- `libdir/libsundials_arkode.lib`,
- `libdir/libsundials_nvec*.lib`,

where the file extension `.lib` is typically `.so` for shared libraries and `.a` for static libraries. The relevant header files are located in the subdirectories

- `incdir/include/arkode`
- `incdir/include/sundials`
- `incdir/include/nvector`
- `incdir/include/sunmatrix`
- `incdir/include/sunlinsol`
- `incdir/include/sunnonlinsol`

The directories `libdir` and `incdir` are the installation library and include directories, respectively. For a default installation, these are `instdir/lib` and `instdir/include`, respectively, where `instdir` is the directory where SUNDIALS was installed (see the section *ARKode Installation Procedure* for further details).

6.2 Data Types

The `sundials_types.h` file contains the definition of the variable type `realtype`, which is used by the SUNDIALS solvers for all floating-point data, the definition of the integer type `sunindextype`, which is used for vector and matrix indices, and `booleantype`, which is used for certain logic operations within SUNDIALS.

6.2.1 Floating point types

The type "realtype" can be set to `float`, `double`, or `long double`, depending on how SUNDIALS was installed (with the default being `double`). The user can change the precision of the SUNDIALS solvers' floating-point arithmetic at the configuration stage (see the section *Configuration options (Unix/Linux)*).

Additionally, based on the current precision, `sundials_types.h` defines the values BIG_REAL to be the largest value representable as a `realtype`, SMALL_REAL to be the smallest positive value representable as a `realtype`, and UNIT_ROUNDOFF to be the smallest realtype number, ε, such that $1.0 + \varepsilon \neq 1.0$.

Within SUNDIALS, real constants may be set to have the appropriate precision by way of a macro called RCONST. It is this macro that needs the ability to branch on the definition `realtype`. In ANSI C, a floating-point constant with no suffix is stored as a `double`. Placing the suffix "F" at the end of a floating point constant makes it a `float`, whereas using the suffix "L" makes it a `long double`. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines `A` to be a `double` constant equal to 1.0, `B` to be a `float` constant equal to 1.0, and `C` to be a `long double` constant equal to 1.0. The macro call `RCONST(1.0)` automatically expands to 1.0 if `realtype` is `double`, to `1.0F` if `realtype` is `float`, or to `1.0L` if `realtype` is `long double`. SUNDIALS uses the `RCONST` macro internally to declare all of its floating-point constants.

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent, except for any calls to precision-specific standard math library functions. Users can, however, use the types `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the size of `realtype` values that are passed to and from SUNDIALS). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries have been compiled using the same precision (for details see the section *ARKode Installation Procedure*).

6.2.2 Integer types used for vector and matrix indices

The type `sunindextype` can be either a 32- or 64-bit *signed* integer. The default is the portable `int64_t` type, and the user can change it to `int32_t` at the configuration stage. The configuration system will detect if the compiler does not support portable types, and will replace `int32_t` and `int64_t` with `int`, `long int`, or `long long int` as appropriate, to ensure use of the desired sizes on Linux, Mac OS X, and Windows platforms. SUNDIALS currently does not support *unsigned* integer types for vector and matrix indices, although these could be added in the future if there is sufficient demand.

A user program which uses `sunindextype` to handle vector and matrix indices will work with both index storage types except for any calls to index storage-specific external libraries. (Our `C` and `C++` example programs use `sunindextype`.) Users can, however, use any one of `int`, `long int`, `int32_t`, `int64_t` or `long long int` in their code, assuming that this usage is consistent with the typedef for `sunindextype` on their architecture. Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `sunindextype`, so long as the SUNDIALS libraries use the appropriate index storage type (for details see the section *ARKode Installation Procedure*).

6.3 Header Files

When using MRIStep, the calling program must include several header files so that various macros and data types can be used. The header file that is always required is:

- `arkode/arkode_mristep.h`, the main header file for the MRIStep time-stepping module, which defines the several types and various constants, includes function prototypes, and includes the shared `arkode/arkode.h` header file.

Note that `arkode.h` includes `sundials_types.h` directly, which defines the types `realtype`, `sunindextype`, and `booleantype` and the constants `SUNFALSE` and `SUNTRUE`, so a user program does not need to include `sundials_types.h` directly.

Additionally, the calling program must also include an NVECTOR implementation header file, of the form `nvector/nvector_***.h`, corresponding to the user's preferred data layout and form of parallelism. See the section *Vector Data Structures* for details for the appropriate name. This file in turn includes the header file `sundials_nvector.h` which defines the abstract `N_Vector` data type.

If the user wishes to manually select between any of the pre-defined ERK or DIRK Butcher tables as the basis for a MIS method, these are defined through a set of constants that are enumerated in the header files `arkode/arkode_butcher_erk.h` and `arkode/arkode_butcher_dirk.h`, or if a user wishes to manually specify a Butcher table, the corresponding `ARKodeButcherTable` structure is defined in `arkode/arkode_butcher.h`. Alternatively, slow-to-fast coupling coefficient tables are enumerated in the header file `arkode/arkode_mristp.h`, or if a user wishes to manually specify a coupling table, the corresponding `MRIStepCouplingMem` structure is defined in `arkode/arkode_mristep.h`.

If the user specifies that the slow time scale should be treated implicitly, then each implicit stage will require a nonlinear solver for the resulting system of algebraic equations – the default for this is a modified or inexact Newton iteration, depending on the user's choice of linear solver. If using a non-default nonlinear solver module, or when interacting with a SUNNONLINSOL module directly, the calling program must also include a SUNNONLINSOL header file, of the form `sunnonlinsol/sunnonlinsol_***.h` where `***` is the name of the nonlinear solver module (see the section *Description of the SUNNonlinearSolver Module* for more information). This file in turn includes the header file `sundials_nonlinearsolver.h` which defines the abstract `SUNNonlinearSolver` data type.

If using a nonlinear solver that requires the solution of a linear system of the form $\mathcal{A}x = b$ (e.g., the default Newton iteration), then a linear solver module header file will also be required. The header files corresponding to the SUNDIALS-provided linear solver modules available for use with ARKode are:

- Direct linear solvers:
  - `sunlinsol/sunlinsol_dense.h`, which is used with the dense linear solver module, SUNLINSOL_DENSE;
  - `sunlinsol/sunlinsol_band.h`, which is used with the banded linear solver module, SUNLINSOL_BAND;
  - `sunlinsol/sunlinsol_lapackdense.h`, which is used with the LAPACK dense linear solver module, SUNLINSOL_LAPACKDENSE;
  - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK banded linear solver module, SUNLINSOL_LAPACKBAND;
  - `sunlinsol/sunlinsol_klu.h`, which is used with the KLU sparse linear solver module, SUNLINSOL_KLU;
  - `sunlinsol/sunlinsol_superlumt.h`, which is used with the SuperLU_MT sparse linear solver module, SUNLINSOL_SUPERLUMT;

  - `sunlinsol/sunlinsol_superludist.h`, which is used with the SuperLU_DIST parallel sparse linear solver module, SUNLINSOL_SUPERLUDIST;
  - `sunlinsol/sunlinsol_cusolversp_batchqr.h`, which is used with the batched sparse QR factorization method provided by the NVDIA cuSOLVER library, SUNLINSOL_CUSOLVERSP_BATCHQR;

- Iterative linear solvers:
  - `sunlinsol/sunlinsol_spgmr.h`, which is used with the scaled, preconditioned GMRES Krylov linear solver module, SUNLINSOL_SPGMR;
  - `sunlinsol/sunlinsol_spfgmr.h`, which is used with the scaled, preconditioned FGMRES Krylov linear solver module, SUNLINSOL_SPFGMR;
  - `sunlinsol/sunlinsol_spbcgs.h`, which is used with the scaled, preconditioned Bi-CGStab Krylov linear solver module, SUNLINSOL_SPBCGS;
  - `sunlinsol/sunlinsol_sptfqmr.h`, which is used with the scaled, preconditioned TFQMR Krylov linear solver module, SUNLINSOL_SPTFQMR;
  - `sunlinsol/sunlinsol_pcg.h`, which is used with the scaled, preconditioned CG Krylov linear solver module, SUNLINSOL_PCG;

The header files for the SUNLINSOL_DENSE and SUNLINSOL_LAPACKDENSE linear solver modules include the file `sunmatrix/sunmatrix_dense.h`, which defines the SUNMATRIX_DENSE matrix module, as well as various functions and macros for acting on such matrices.

The header files for the SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND linear solver modules include the file `sunmatrix/sunmatrix_band.h`, which defines the SUNMATRIX_BAND matrix module, as well as various functions and macros for acting on such matrices.

The header files for the SUNLINSOL_KLU and SUNLINSOL_SUPERLUMT linear solver modules include the file `sunmatrix/sunmatrix_sparse.h`, which defines the SUNMATRIX_SPARSE matrix module, as well as various functions and macros for acting on such matrices.

The header file for the SUNLINSOL_CUSOLVERSP_BATCHQR linear solver module includes the file `sunmatrix/sunmatrix_cusparse.h`, which defines the SUNMATRIX_CUSPARSE matrix module, as well as various functions for acting on such matrices.

The header file for the SUNLINSOL_SUPERLUDIST linear solver module includes the file `sunmatrix/sunmatrix_slunrloc.h`, which defines the SUNMATRIX_SLUNRLOC matrix module, as well as various functions for acting on such matrices.

The header files for the Krylov iterative solvers include the file `sundials/sundials_iterative.h`, which enumerates the preconditioning type and (for the SPGMR and SPFGMR solvers) the choices for the Gram-Schmidt orthogonalization process.

Other headers may be needed, according to the choice of preconditioner, etc. For example, if preconditioning for an iterative linear solver were performed using the ARKBBDPRE module, the header `arkode/arkode_bbdpre.h` is needed to access the preconditioner initialization routines.

6.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the integration of an ODE IVP using the MRIStep module. Most of the steps are independent of the NVECTOR, SUNMATRIX, SUNLINSOL and SUNNONLINSOL implementations used. For the steps that are not, refer to the sections *Vector Data Structures*, *Matrix Data Structures*, *Description of the SUNLinearSolver module*, and *Description of the SUNNonlinearSolver Module* for the specific name of the function to be called or macro to be referenced.

1. Initialize parallel or multi-threaded environment, if appropriate.

   For example, call `MPI_Init` to initialize MPI if used, or set `num_threads`, the number of threads to use within the threaded vector functions, if used.

2. Set problem dimensions, etc.

   This generally includes the problem size, `N`, and may include the local vector length `Nlocal`.

   Note: The variables `N` and `Nlocal` should be of type `sunindextype`.

3. Set vector of initial values

   To set the vector `y0` of initial values, use the appropriate functions defined by the particular NVECTOR implementation.

   For native SUNDIALS vector implementations (except the CUDA and RAJA based ones), use a call of the form

   ```
   y0 = N_VMake_***(..., ydata);
   ```

   if the `realtype` array `ydata` containing the initial values of y already exists. Otherwise, create a new vector by making a call of the form

   ```
   y0 = N_VNew_***(...);
   ```

   and then set its elements by accessing the underlying data where it is located with a call of the form

   ```
   ydata = N_VGetArrayPointer_***(y0);
   ```

   See the sections *The NVECTOR_SERIAL Module* through *The NVECTOR_PTHREADS Module* for details.

   For the HYPRE and PETSc vector wrappers, first create and initialize the underlying vector, and then create the NVECTOR wrapper with a call of the form

   ```
   y0 = N_VMake_***(yvec);
   ```

   where `yvec` is a HYPRE or PETSc vector. Note that calls like `N_VNew_***(...)` and `N_VGetArrayPointer_***(...)` are not available for these vector wrappers. See the sections *The NVECTOR_PARHYP Module* and *The NVECTOR_PETSC Module* for details.

   If using either the CUDA- or RAJA-based vector implementations use calls to the module-specific routines

   ```
   y0 = N_VMake_***(...);
   ```

   as applicable. See the sections *The NVECTOR_CUDA Module* and *The NVECTOR_RAJA Module* for details.

4. Create an ARKStep object for the fast (inner) integration

   Call `inner_arkode_mem = ARKStepCreate(...)` to create the ARKStep memory block. *`ARKStepCreate()`* returns a `void*` pointer to this memory structure. See the section *ARKStep initialization and deallocation functions* for details.

5. Configure the fast (inner) integrator

   Specify tolerances, create and attach matrix and/or solver objects, or call `ARKStepSet*` functions to configure the fast integrator as desired. See sections *A skeleton of the user's main program* and *Optional input functions* for details on configuring ARKStep.

   Notes on using ARKStep as a fast integrator:

It is the user's responsibility to create, configure, and attach the `inner_arkode_mem` to the MRIStep module. User-specified options regarding how this fast integration should be performed (e.g., adaptive versus fixed time step, explicit/implicit/ImEx partitioning, algebraic solvers, etc.) will be respected during integration of the fast time scales during MRIStep integration.

If a *user_data* pointer needs to be passed to user functions called by the fast (inner) integrator then it should be attached here by calling *ARKStepSetUserData()*. This *user_data* pointer will only be passed to user-supplied functions that are attached to the fast (inner) integrator. To supply a *user_data* pointer to user-supplied functions called by the slow (outer) integrator the desired pointer should be attached by calling *MRIStepSetUserData()* after creating the MRIStep memory below. Note the *user_data* pointers attached to the inner and outer integrators may be the same or different depending on what is required by the user code.

Specifying a rootfinding problem for the fast integration is not supported. Rootfinding problems should be created and initialized with the slow integrator. See the steps below and *MRIStepRootInit()* for more details.

We note that due to the algorithms supported in MRIStep, the ARKStep module used for the fast time scale must be configured with an identity mass matrix.

6. Create an MRIStep object for the slow (outer) integration

   Call `arkode_mem = MRIStepCreate(..., inner_arkode_mem)` to create the MRIStep memory block. *MRIStepCreate()* returns a `void*` pointer to this memory structure. See the section *MRIStep initialization and deallocation functions* for details.

7. Set the slow step size

   Call *MRIStepSetFixedStep()* to specify the slow time step size.

   Specifically, if MRIStep is configured to use an implicit solver for the slow time scale, then the following steps are recommended:

8. Create and configure implicit solvers

   If MRIStep is configured to use an implicit solver for the slow time scale, then:

   (a) Specify integration tolerances

   Call *MRIStepSStolerances()* or *MRIStepSVtolerances()* to specify either a scalar relative tolerance and scalar absolute tolerance, or a scalar relative tolerance and a vector of absolute tolerances, respectively. Alternatively, call *MRIStepWFtolerances()* to specify a function which sets directly the weights used in evaluating WRMS vector norms. See the section *MRIStep tolerance specification functions* for details.

   (b) Create nonlinear solver object

   If a non-default nonlinear solver object is desired for implicit MRI stage solves (see the section *Nonlinear solver interface functions*), then that nonlinear solver object must be created by using the appropriate functions defined by the particular SUNNONLINSOL implementation (e.g., `NLS = SUNNonlinSol_***(...);` where `***` is the name of the nonlinear solver (see the section *Description of the SUNNonlinearSolver Module* for details).

   For the SUNDIALS-supplied SUNNONLINSOL implementations, the nonlinear solver object may be created using a call of the form

   ```
   SUNNonlinearSolver NLS = SUNNonlinSol_*(...);
   ```

   where `*` can be replaced with "Newton", "FixedPoint", or other options, as discussed in the sections *Nonlinear solver interface functions* and *Description of the SUNNonlinearSolver Module*.

Note: by default, MRIStep will use the Newton nonlinear solver (see section *The SUNNonlinearSolver_Newton implementation*), so a custom nonlinear solver object is only needed when using a *different* solver, or for the user to exercise additional controls over the Newton solver.

(c) Attach nonlinear solver module

If a nonlinear solver object was created above, then it must be attached to MRIStep using the call (for details see the section *Nonlinear solver interface functions*):

```
ier = MRIStepSetNonlinearSolver(...);
```

(d) Set nonlinear solver optional inputs

Call the appropriate set functions for the selected nonlinear solver module to change optional inputs specific to that nonlinear solver. These *must* be called after attaching the nonlinear solver to MRIStep, otherwise the optional inputs will be overridden by MRIStep defaults. See the section *Description of the SUNNonlinearSolver Module* for more information on optional inputs.

(e) Create matrix object

If a nonlinear solver requiring a linear solver will be used (e.g., a Newton iteration) and if that linear solver will be matrix-based, then a template Jacobian matrix must be created by using the appropriate functions defined by the particular SUNMATRIX implementation.

For the SUNDIALS-supplied SUNMATRIX implementations, the matrix object may be created using a call of the form

```
SUNMatrix A = SUNBandMatrix(...);
```

or

```
SUNMatrix A = SUNDenseMatrix(...);
```

or

```
SUNMatrix A = SUNSparseMatrix(...);
```

or similarly for the CUDA and SuperLU_DIST matrix modules (see the sections *The SUNMATRIX_CUSPARSE Module* or *The SUNMATRIX_SLUNRLOC Module* for further information).

NOTE: The dense, banded, and sparse matrix objects are usable only in a serial or threaded environment.

(f) Create linear solver object

If a nonlinear solver requiring a linear solver will be used (e.g., a Newton iteration), then the desired linear solver object(s) must be created by using the appropriate functions defined by the particular SUNLINSOL implementation.

For any of the SUNDIALS-supplied SUNLINSOL implementations, the linear solver object may be created using a call of the form

```
SUNLinearSolver LS = SUNLinSol_*(...);
```

where $*$ can be replaced with "Dense", "SPGMR", or other options, as discussed in the sections *Linear solver interface functions* and *Description of the SUNLinearSolver module*.

(g) Set linear solver optional inputs

Call `*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See the documentation for each SUNLINSOL module in the section *Description of the SUNLinearSolver module* for details.

(h) Attach linear solver module

If a linear solver was created above for implicit MRI stage solves, initialize the ARKLS linear solver interface by attaching the linear solver object (and Jacobian matrix object, if applicable) with the call (for details see the section *Linear solver interface functions*):

```
ier = MRIStepSetLinearSolver(...);
```

9. Set optional inputs

   Call `MRIStepSet*` functions to change any optional inputs that control the behavior of MRIStep from their default values. See the section *Optional input functions* for details.

10. Specify rootfinding problem

    Optionally, call *`MRIStepRootInit()`* to initialize a rootfinding problem to be solved during the integration of the ODE system. See the section *Rootfinding initialization function* for general details, and the section *Optional input functions* for relevant optional input calls.

11. Advance solution in time

    For each point at which output is desired, call

    ```
    ier = MRIStepEvolve(arkode_mem, tout, yout, &tret, itask);
    ```

    Here, `itask` specifies the return mode. The vector `yout` (which can be the same as the vector `y0` above) will contain $y(t_{\text{out}})$. See the section *MRIStep solver function* for details.

12. Get optional outputs

    Call `MRIStepGet*` and/or `ARKStepGet*` functions to obtain optional output from the slow or fast integrators respectively. See the section *Optional output functions* and *Optional output functions* for details.

13. Deallocate memory for solution vector

    Upon completion of the integration, deallocate memory for the vector `y` (or `yout`) by calling the NVECTOR destructor function:

    ```
    N_VDestroy(y);
    ```

14. Free solver memory

    Call `ARKStepFree(&inner_arkode_mem)` and `MRIStepFree(&arkode_mem)` to free the memory allocated for fast and slow integration modules respectively.

15. Free linear solver and matrix memory

    Call *`SUNLinSolFree()`* and (possibly) *`SUNMatDestroy()`* to free any memory allocated for any linear solver and/or matrix objects created above for either the fast or slow integrators.

16. Free nonlinear solver memory

    If a user-supplied `SUNNonlinearSolver` was provided to MRIStep, then call *`SUNNonlinSolFree()`* to free any memory allocated for the nonlinear solver object created above.

17. Finalize MPI, if used

    Call `MPI_Finalize` to terminate MPI.

SUNDIALS provides some linear solvers only as a means for users to get problems running and not as highly efficient solvers. For example, if solving a dense system, we suggest using the LAPACK solvers if the size of the linear system is $> 50,000$ (thanks to A. Nicolai for his testing and recommendation). See the table *SUNDIALS linear solver interfaces and vector implementations that can be used for each* for a listing of the linear solver interfaces available as `SUNLinearSolver` modules and the vector implementations required for use.

6.5 MRIStep User-callable functions

This section describes the functions that are called by the user to setup and then solve an IVP using the MRIStep time-stepping module. Some of these are required; however, starting with the section *Optional input functions*, the functions listed involve optional inputs/outputs or restarting, and those paragraphs may be skipped for a casual use of ARKode's MRIStep module. In any case, refer to the preceding section, *A skeleton of the user's main program*, for the correct order of these calls.

On an error, each user-callable function returns a negative value (or `NULL` if the function returns a pointer) and sends an error message to the error handler routine, which prints the message to `stderr` by default. However, the user can set a file as error output or can provide her own error handler function (see the section *Optional input functions* for details).

6.5.1 MRIStep initialization and deallocation functions

void* **`MRIStepCreate`** (*ARKRhsFn fs*, realtype *t0*, N_Vector *y0*, MRISTEP_ID *inner_step_id*, void* *inner_step_mem*)

This function allocates and initializes memory for a problem to be solved using the MRIStep time-stepping module in ARKode.

Arguments:

- *fs* – the name of the C function (of type `ARKRhsFn()`) defining the slow portion of the right-hand side function in $\dot{y} = f_s(t, y) + f_f(t, y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.
- *inner_step_id* – the identifier for the inner stepper. Currently `MRISTEP_ARKSTEP` is the only supported option.
- *inner_step_mem* – a `void*` pointer to the ARKStep memory block for integrating the fast time scale.

Return value: If successful, a pointer to initialized problem memory of type `void*`, to be passed to all user-facing MRIStep routines listed below. If unsuccessful, a `NULL` pointer will be returned, and an error message will be printed to `stderr`.

void **`MRIStepFree`** (void** *arkode_mem*)

This function frees the problem memory *arkode_mem* created by `MRIStepCreate()`.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.

Return value: None

6.5.2 MRIStep tolerance specification functions

These functions specify the integration tolerances. One of them **should** be called before the first call to `MRIStepEvolve()`; otherwise default values of `reltol = 1e-4` and `abstol = 1e-9` will be used, which may be entirely incorrect for a specific problem.

The integration tolerances `reltol` and `abstol` define a vector of error weights, `ewt`. In the case of `MRIStepSStolerances()`, this vector has components

```
ewt[i] = 1.0/(reltol*abs(y[i]) + abstol);
```

whereas in the case of *MRIStepSVtolerances()* the vector components are given by

```
ewt[i] = 1.0/(reltol*abs(y[i]) + abstol[i]);
```

This vector is used in all error tests, which use a weighted RMS norm on all error-like vectors v:

$$\|v\|_{WRMS} = \left(\frac{1}{N} \sum_{i=1}^{N} (v_i \; ewt_i)^2 \right)^{1/2},$$

where N is the problem dimension.

Alternatively, the user may supply a custom function to supply the `ewt` vector, through a call to *MRIStepWFtolerances()*.

int **MRIStepSStolerances**(void* *arkode_mem*, realtype *reltol*, realtype *abstol*)
This function specifies scalar relative and absolute tolerances.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – scalar absolute tolerance.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_NO_MALLOC* if the MRIStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **MRIStepSVtolerances**(void* *arkode_mem*, realtype *reltol*, N_Vector *abstol*)
This function specifies a scalar relative tolerance and a vector absolute tolerance (a potentially different absolute tolerance for each vector component).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *reltol* – scalar relative tolerance.
- *abstol* – vector containing the absolute tolerances for each solution component.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_NO_MALLOC* if the MRIStep memory was not allocated by the time-stepping module
- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **MRIStepWFtolerances**(void* *arkode_mem*, *ARKEwtFn* *efun*)
This function specifies a user-supplied function *efun* to compute the error weight vector `ewt`.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.

- *efun* – the name of the function (of type *ARKEwtFn()*) that implements the error weight vector computation.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was NULL
- *ARK_NO_MALLOC* if the MRIStep memory was not allocated by the time-stepping module

6.5.2.1 General advice on the choice of tolerances

For many users, the appropriate choices for tolerance values in reltol and abstol are a concern. The following pieces of advice are relevant.

1. The scalar relative tolerance reltol is to be set to control relative errors. So a value of 10^{-4} means that errors are controlled to .01%. We do not recommend using reltol larger than 10^{-3}. On the other hand, reltol should not be so small that it is comparable to the unit roundoff of the machine arithmetic (generally around 10^{-15} for double-precision).
2. The absolute tolerances abstol (whether scalar or vector) need to be set to control absolute errors when any components of the solution vector y may be so small that pure relative error control is meaningless. For example, if y_i starts at some nonzero value, but in time decays to zero, then pure relative error control on y_i makes no sense (and is overly costly) after y_i is below some noise level. Then abstol (if scalar) or abstol[i] (if a vector) needs to be set to that noise level. If the different components have different noise levels, then abstol should be a vector. For example, see the example problem ark_robertson.c, and the discussion of it in the ARKode Examples Documentation *[R2018]*. In that problem, the three components vary between 0 and 1, and have different noise levels; hence the atols vector therein. It is impossible to give any general advice on abstol values, because the appropriate noise levels are completely problem-dependent. The user or modeler hopefully has some idea as to what those noise levels are.
3. Finally, it is important to pick all the tolerance values conservatively, because they control the error committed on each individual step. The final (global) errors are an accumulation of those per-step errors, where that accumulation factor is problem-dependent. A general rule of thumb is to reduce the tolerances by a factor of 10 from the actual desired limits on errors. So if you want .01% relative accuracy (globally), a good choice for reltol is 10^{-5}. In any case, it is a good idea to do a few experiments with the tolerances to see how the computed solution values vary as tolerances are reduced.

6.5.2.2 Advice on controlling nonphysical negative values

In many applications, some components in the true solution are always positive or non-negative, though at times very small. In the numerical solution, however, small negative (nonphysical) values can then occur. In most cases, these values are harmless, and simply need to be controlled, not eliminated, but in other cases any value that violates a constraint may cause a simulation to halt. For both of these scenarios the following pieces of advice are relevant.

1. The best way to control the size of unwanted negative computed values is with tighter absolute tolerances. Again this requires some knowledge of the noise level of these components, which may or may not be different for different components. Some experimentation may be needed.
2. If output plots or tables are being generated, and it is important to avoid having negative numbers appear there (for the sake of avoiding a long explanation of them, if nothing else), then eliminate them, but only in the context of the output medium. Then the internal values carried by the solver are unaffected. Remember that a small negative value in y returned by MRIStep, with magnitude comparable to abstol or less, is equivalent to zero as far as the computation is concerned.
3. The user's right-hand side routine f^S should never change a negative value in the solution vector y to a non-negative value in attempt to "fix" this problem, since this can lead to numerical instability. If the f^S routine

cannot tolerate a zero or negative value (e.g. because there is a square root or log), then the offending value should be changed to zero or a tiny positive number in a temporary variable (not in the input y vector) for the purposes of computing $f^S(t, y)$.

6.5.3 Linear solver interface functions

As previously explained, the Newton iterations used in solving implicit systems within MRIStep require the solution of linear systems of the form

$$\mathcal{A}\left(z_i^{(m)}\right)\delta^{(m+1)} = -G\left(z_i^{(m)}\right)$$

where

$$\mathcal{A} \approx I - \gamma J, \qquad J = \frac{\partial f^S}{\partial y}.$$

ARKode's ARKLS linear solver interface supports all valid `SUNLinearSolver` modules for this task.

Matrix-based `SUNLinearSolver` modules utilize `SUNMatrix` objects to store the approximate Jacobian matrix J, the Newton matrix $\mathcal{A}$, and, when using direct solvers, the factorizations used throughout the solution process.

Matrix-free `SUNLinearSolver` modules instead use iterative methods to solve the Newton systems of equations, and only require the *action* of the matrix on a vector, $\mathcal{A}v$. With most of these methods, preconditioning can be done on the left only, on the right only, on both the left and the right, or not at all. The exceptions to this rule are SPFGMR that supports right preconditioning only and PCG that performs symmetric preconditioning. For the specification of a preconditioner, see the iterative linear solver portions of the sections *Optional input functions* and *User-supplied functions*.

If preconditioning is done, user-supplied functions should be used to define left and right preconditioner matrices P_1 and P_2 (either of which could be the identity matrix), such that the product $P_1 P_2$ approximates the Newton matrix $\mathcal{A} = I - \gamma J$.

To specify a generic linear solver for MRIStep to use for the Newton systems, after the call to `MRIStepCreate()` but before any calls to `MRIStepEvolve()`, the user's program must create the appropriate `SUNLinearSolver` object and call the function `MRIStepSetLinearSolver()`, as documented below. To create the `SUNLinearSolver` object, the user may call one of the SUNDIALS-packaged SUNLinSol module constructor routines via a call of the form

```
SUNLinearSolver LS = SUNLinSol_*(...);
```

The current list of such constructor routines includes `SUNLinSol_Dense()`, `SUNLinSol_Band()`, `SUNLinSol_LapackDense()`, `SUNLinSol_LapackBand()`, `SUNLinSol_KLU()`, `SUNLinSol_SuperLUMT()`, `SUNLinSol_SuperLUDIST()`, `SUNLinSol_cuSolverSp_batchQR()`, `SUNLinSol_SPGMR()`, `SUNLinSol_SPFGMR()`, `SUNLinSol_SPBCGS()`, `SUNLinSol_SPTFQMR()`, and `SUNLinSol_PCG()`.

Alternately, a user-supplied `SUNLinearSolver` module may be created and used instead. The use of each of the generic linear solvers involves certain constants, functions and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the specific `SUNMatrix` or `SUNLinearSolver` module in question, as described in the sections *Matrix Data Structures* and *Description of the SUNLinearSolver module*.

Once this solver object has been constructed, the user should attach it to MRIStep via a call to `MRIStepSetLinearSolver()`. The first argument passed to this function is the MRIStep memory pointer returned by `MRIStepCreate()`; the second argument is the `SUNLinearSolver` object created above. The third argument is an optional `SUNMatrix` object to accompany matrix-based `SUNLinearSolver` inputs (for matrix-free linear solvers, the third argument should be `NULL`). A call to this function initializes the ARKLS linear

solver interface, linking it to the MRIStep integrator, and allows the user to specify additional parameters and routines pertinent to their choice of linear solver.

int **MRIStepSetLinearSolver**(void* *arkode_mem*, SUNLinearSolver *LS*, SUNMatrix *J*)

This function specifies the `SUNLinearSolver` object that MRIStep should use, as well as a template Jacobian `SUNMatrix` object (if applicable).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *LS* – the `SUNLinearSolver` object to use.
- *J* – the template Jacobian `SUNMatrix` object to use (or `NULL` if not applicable).

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_MEM_FAIL* if there was a memory allocation failure
- *ARKLS_ILL_INPUT* if ARKLS is incompatible with the provided *LS* or *J* input objects, or the current `N_Vector` module.

Notes: If *LS* is a matrix-free linear solver, then the *J* argument should be `NULL`.

If *LS* is a matrix-based linear solver, then the template Jacobian matrix *J* will be used in the solve process, so if additional storage is required within the `SUNMatrix` object (e.g. for factorization of a banded matrix), ensure that the input object is allocated with sufficient size (see the documentation of the particular SUNMATRIX type in the section *Matrix Data Structures* for further information).

When using sparse linear solvers, it is typically much more efficient to supply *J* so that it includes the full sparsity pattern of the Newton system matrices $\mathcal{A} = I - \gamma J$, even if *J* itself has zeros in nonzero locations of I. The reasoning for this is that $\mathcal{A}$ is constructed in-place, on top of the user-specified values of *J*, so if the sparsity pattern in *J* is insufficient to store $\mathcal{A}$ then it will need to be resized internally by MRIStep.

6.5.4 Nonlinear solver interface functions

When changing the nonlinear solver in MRIStep, after the call to *MRIStepCreate()* but before any calls to *MRIStepEvolve()*, the user's program must create the appropriate SUNNonlinSol object and call *MRIStepSetNonlinearSolver()*, as documented below. If any calls to *MRIStepEvolve()* have been made, then MRIStep will need to be reinitialized by calling *MRIStepReInit()* to ensure that the nonlinear solver is initialized correctly before any subsequent calls to *MRIStepEvolve()*.

The first argument passed to the routine *MRIStepSetNonlinearSolver()* is the MRIStep memory pointer returned by *MRIStepCreate()*; the second argument passed to this function is the desired SUNNonlinSol object to use for solving the nonlinear system for each implicit stage. A call to this function attaches the nonlinear solver to the main MRIStep integrator.

int **MRIStepSetNonlinearSolver**(void* *arkode_mem*, SUNNonlinearSolver *NLS*)

This function specifies the `SUNNonlinearSolver` object that MRIStep should use for implicit stage solves.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *NLS* – the `SUNNonlinearSolver` object to use.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_MEM_FAIL* if there was a memory allocation failure
- *ARK_ILL_INPUT* if MRIStep is incompatible with the provided *NLS* input object.

Notes: MRIStep will use the Newton SUNNonlinSol module by default; a call to this routine replaces that module with the supplied *NLS* object.

6.5.5 Rootfinding initialization function

As described in the section *Rootfinding*, while solving the IVP, ARKode's time-stepping modules have the capability to find the roots of a set of user-defined functions. In the MRIStep module root finding is performed between slow solution time steps only (i.e., it is not performed within the sub-stepping a fast time scales). To activate the root-finding algorithm, call the following function. This is normally called only once, prior to the first call to `MRIStepEvolve()`, but if the rootfinding problem is to be changed during the solution, `MRIStepRootInit()` can also be called prior to a continuation call to `MRIStepEvolve()`.

int **`MRIStepRootInit`**(void* *arkode_mem*, int *nrtfn*, *ARKRootFn g*)

Initializes a rootfinding problem to be solved during the integration of the ODE system. It must be called after `MRIStepCreate()`, and before `MRIStepEvolve()`.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nrtfn* – number of functions g_i, an integer ≥ 0.
- *g* – name of user-supplied function, of type `ARKRootFn()`, defining the functions g_i whose roots are sought.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_MEM_FAIL* if there was a memory allocation failure
- *ARK_ILL_INPUT* if *nrtfn* is greater than zero but *g* = `NULL`.

Notes: To disable the rootfinding feature after it has already been initialized, or to free memory associated with MRIStep's rootfinding module, call *MRIStepRootInit* with *nrtfn = 0*.

Similarly, if a new IVP is to be solved with a call to `MRIStepReInit()`, where the new IVP has no rootfinding problem but the prior one did, then call *MRIStepRootInit* with *nrtfn = 0*.

Rootfinding is only supported for the slow (outer) integrator and should not be actived for the fast (inner) integrator.

6.5.6 MRIStep solver function

This is the central step in the solution process – the call to perform the integration of the IVP. The input argument *itask* specifies one of two modes as to where MRIStep is to return a solution. These modes are modified if the user has set a stop time (with a call to the optional input function `MRIStepSetStopTime()`) or has requested rootfinding.

int **`MRIStepEvolve`**(void* *arkode_mem*, realtype *tout*, N_Vector *yout*, realtype **tret*, int *itask*)

Integrates the ODE over an interval in t.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *tout* – the next time at which a computed solution is desired.
- *yout* – the computed solution vector.
- *tret* – the time corresponding to *yout* (output).
- *itask* – a flag indicating the job of the solver for the next user step.

  The *ARK_NORMAL* option causes the solver to take internal steps until it has just overtaken a user-specified output time, *tout*, in the direction of integration, i.e. $t_{n-1} < tout \leq t_n$ for forward integration, or $t_n \leq tout < t_{n-1}$ for backward integration. It will then compute an approximation to the solution $y(tout)$ by interpolation (using one of the dense output routines described in the section *Interpolation*).

  The *ARK_ONE_STEP* option tells the solver to only 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 *tout* then the solver will again return an interpolated result; otherwise it will return a copy of the internal solution y_n in the vector *yout*

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_ROOT_RETURN* if `MRIStepEvolve()` succeeded, and found one or more roots. If the number of root functions, *nrtfn*, is greater than 1, call `MRIStepGetRootInfo()` to see which g_i were found to have a root at (**tret*).
- *ARK_TSTOP_RETURN* if `MRIStepEvolve()` succeeded and returned at *tstop*.
- *ARK_MEM_NULL* if the *arkode_mem* argument was `NULL`.
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if one of the inputs to `MRIStepEvolve()` is illegal, or some other input to the solver was either illegal or missing. Details will be provided in the error message. Typical causes of this failure:

  1. A component of the error weight vector became zero during internal time-stepping.
  2. The linear solver initialization function (called by the user after calling `ARKStepCreate()`) failed to set the linear solver-specific *lsolve* field in *arkode_mem*.
  3. A root of one of the root functions was found both at a point t and also very near t.

- *ARK_TOO_MUCH_WORK* if the solver took *mxstep* internal steps but could not reach *tout*. The default value for *mxstep* is *MXSTEP_DEFAULT = 500*.
- *ARK_CONV_FAILURE* if convergence test failures occurred too many times (*ark_maxncf*) during one internal time step.
- *ARK_LINIT_FAIL* if the linear solver's initialization function failed.
- *ARK_LSETUP_FAIL* if the linear solver's setup routine failed in an unrecoverable manner.
- *ARK_LSOLVE_FAIL* if the linear solver's solve routine failed in an unrecoverable manner.
- *ARK_VECTOROP_ERR* a vector operation error occured.
- *ARK_INNERSTEP_FAILED* if the inner stepper returned with an unrecoverable error. The value returned from the inner stepper can be obtained with `MRIStepGetLastInnerStepFlag()`.
- *ARK_INVALID_TABLE* if an invalid coupling table was provided.

Notes: The input vector *yout* can use the same memory as the vector *y0* of initial conditions that was passed to `MRIStepCreate()`.

In *ARK_ONE_STEP* mode, *tout* is used only on the first call, and only to get the direction and a rough scale of the independent variable.

All failure return values are negative and so testing the return argument for negative values will trap all `MRIStepEvolve()` failures.

Since interpolation may reduce the accuracy in the reported solution, if full method accuracy is desired the user should issue a call to `MRIStepSetStopTime()` before the call to `MRIStepEvolve()` to specify a fixed stop time to end the time step and return to the user. Upon return from `MRIStepEvolve()`, a copy of the internal solution y_n will be returned in the vector *yout*. Once the integrator returns at a *tstop* time, any future testing for *tstop* is disabled (and can be re-enabled only though a new call to `MRIStepSetStopTime()`).

On any error return in which one or more internal steps were taken by `MRIStepEvolve()`, the returned values of *tret* and *yout* correspond to the farthest point reached in the integration. On all other error returns, *tret* and *yout* are left unchanged from those provided to the routine.

6.5.7 Optional input functions

There are numerous optional input parameters that control the behavior of the MRIStep solver, each of which may be modified from its default value through calling an appropriate input function. The following tables list all optional input functions, grouped by which aspect of MRIStep they control. Detailed information on the calling syntax and arguments for each function are then provided following each table.

The optional inputs are grouped into the following categories:

- General MRIStep options (*Optional inputs for MRIStep*), and
- IVP method solver options (*Optional inputs for IVP method selection*)
- Implicit stage solver options (*Optional inputs for implicit stage solves*),
- Linear solver interface options (*Linear solver interface optional input functions*), and
- Rootfinding options (MRIStep_CInterface.MRIStepRootfindingInputTable).

For the most casual use of MRIStep, relying on the default set of solver parameters, the reader can skip to the following section, *User-supplied functions*.

We note that, on an error return, all of the optional input functions send an error message to the error handler function. All error return values are negative, so a test on the return arguments for negative values will catch all errors. Finally, a call to an `MRIStepSet***` function can generally be made from the user's calling program at any time and, if successful, takes effect immediately. `MRIStepSet***` functions that cannot be called at any time note this in the "**Notes**:" section of the function documentation.

6.5.7.1 Optional inputs for MRIStep

Optional input	Function name	Default
Return MRIStep solver parameters to their defaults	*MRIStepSetDefaults()*	internal
Set dense output interpolation type	*MRIStepSetInterpolantType()*	ARK_INTERP_HERMITE
Set dense output polynomial degree	*MRIStepSetInterpolantDegree()*	5
Supply a pointer to a diagnostics output file	*MRIStepSetDiagnostics()*	NULL
Supply a pointer to an error output file	*MRIStepSetErrFile()*	stderr
Supply a custom error handler function	*MRIStepSetErrHandlerFn()*	internal fn
Run with fixed-step sizes	*MRIStepSetFixedStep()*	required
Maximum no. of warnings for $t_n + h = t_n$	*MRIStepSetMaxHnilWarns()*	10
Maximum no. of internal steps before *tout*	*MRIStepSetMaxNumSteps()*	500
Set a value for t_{stop}	*MRIStepSetStopTime()*	∞
Supply a pointer for user data	*MRIStepSetUserData()*	NULL
Supply a function to be called prior to the inner integration	*MRIStepSetPreInnerFn()*	NULL
Supply a function to be called after the inner integration	*MRIStepSetPostInnerFn()*	NULL

int **MRIStepSetDefaults**(void* *arkode_mem*)

Resets all optional input parameters to MRIStep's original default values.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This function does not change problem-defining function pointers *fs* and *ff* or the *user_data* pointer. It also does not affect any data structures or options related to root-finding (those can be reset using *MRIStepRootInit()*).

int **MRIStepSetInterpolantType**(void* *arkode_mem*, int *itype*)

Specifies use of the Lagrange or Hermite interpolation modules (used for dense output – interpolation of solution output values and implicit method predictors).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *itype* – requested interpolant type (ARK_INTERP_HERMITE or ARK_INTERP_LAGRANGE)

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is NULL
- *ARK_MEM_FAIL* if the interpolation module cannot be allocated
- *ARK_ILL_INPUT* if the *itype* argument is not recognized or the interpolation module has already been initialized

Notes: The Hermite interpolation module is described in the Section *Hermite interpolation module*, and the Lagrange interpolation module is described in the Section *Lagrange interpolation module*.

This routine frees any previously-allocated interpolation module, and re-creates one according to the specified argument. Thus any previous calls to `MRIStepSetInterpolantDegree()` will be nullified.

This routine must be called *after* the call to `MRIStepCreate()`. After the first call to `MRIStepEvolve()` the interpolation type may not be changed without first calling `MRIStepReInit()`.

If this routine is not called, the Hermite interpolation module will be used.

int **MRIStepSetInterpolantDegree**(void* *arkode_mem*, int *degree*)

Specifies the degree of the polynomial interpolant used for dense output (i.e. interpolation of solution output values and implicit method predictors).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *degree* – requested polynomial degree.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory or interpolation module are `NULL`
- *ARK_INTERP_FAIL* if this is called after `MRIStepEvolve()`
- *ARK_ILL_INPUT* if an argument has an illegal value or the interpolation module has already been initialized

Notes: Allowed values are between 0 and 5.

This routine should be called *after* `MRIStepCreate()` and *before* `MRIStepEvolve()`. After the first call to `MRIStepEvolve()` the interpolation degree may not be changed without first calling `MRIStepReInit()`.

If a user calls both this routine and `MRIStepSetInterpolantType()`, then `MRIStepSetInterpolantType()` must be called first.

Since the accuracy of any polynomial interpolant is limited by the accuracy of the time-step solutions on which it is based, the *actual* polynomial degree that is used by MRIStep will be the minimum of $q - 1$ and the input *degree*, where q is the order of accuracy for the time integration method.

int **MRIStepSetDenseOrder**(void* *arkode_mem*, int *dord*)

This function is deprecated, and will be removed in a future release. Users should transition to calling `MRIStepSetInterpolantDegree()` *instead.*

int **MRIStepSetDiagnostics**(void* *arkode_mem*, FILE* *diagfp*)

Specifies the file pointer for a diagnostics file where all MRIStep step adaptivity and solver information is written.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *diagfp* – pointer to the diagnostics output file.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This parameter can be `stdout` or `stderr`, although the suggested approach is to specify a pointer to a unique file opened by the user and returned by `fopen`. If not called, or if called with a `NULL` file pointer, all diagnostics output is disabled.

When run in parallel, only one process should set a non-NULL value for this pointer, since statistics from all processes would be identical.

int **MRIStepSetErrFile**(void* *arkode_mem*, FILE* *errfp*)
Specifies a pointer to the file where all MRIStep warning and error messages will be written if the default internal error handling function is used.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *errfp* – pointer to the output file.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value for *errfp* is `stderr`.

Passing a `NULL` value disables all future error message output (except for the case wherein the MRIStep memory pointer is `NULL`). This use of the function is strongly discouraged.

If used, this routine should be called before any other optional input functions, in order to take effect for subsequent error messages.

int **MRIStepSetErrHandlerFn**(void* *arkode_mem*, *ARKErrHandlerFn* *ehfun*, void* *eh_data*)
Specifies the optional user-defined function to be used in handling error messages.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *ehfun* – name of user-supplied error handler function.
- *eh_data* – pointer to user data passed to *ehfun* every time it is called.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Error messages indicating that the MRIStep solver memory is `NULL` will always be directed to `stderr`.

int **MRIStepSetFixedStep**(void* *arkode_mem*, realtype *hs*)
Set the slow step size used within MRIStep.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *hs* – value of the outer (slow) step size.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes:

The step sizes used by the inner (fast) stepper may be controlled through calling the appropriate "Set" routines on the inner integrator.

int **`MRIStepSetMaxHnilWarns`**(void* *arkode_mem*, int *mxhnil*)
Specifies the maximum number of messages issued by the solver to warn that $t + h = t$ on the next internal step, before MRIStep will instead return with an error.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *mxhnil* – maximum allowed number of warning messages (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 10; set *mxhnil* to zero to specify this default.

A negative value indicates that no warning messages should be issued.

int **`MRIStepSetMaxNumSteps`**(void* *arkode_mem*, long int *mxsteps*)
Specifies the maximum number of steps to be taken by the solver in its attempt to reach the next output time, before MRIStep will return with an error.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *mxsteps* – maximum allowed number of internal steps.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Passing *mxsteps* = 0 results in MRIStep using the default value (500).

Passing *mxsteps* < 0 disables the test (not recommended).

int **`MRIStepSetStopTime`**(void* *arkode_mem*, realtype *tstop*)
Specifies the value of the independent variable t past which the solution is not to proceed.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *tstop* – stopping time for the integrator.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default is that no stop time is imposed.

int **MRIStepSetUserData**(void* *arkode_mem*, void* *user_data*)
Specifies the user data block *user_data* for the outer integrator and attaches it to the main MRIStep memory block.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *user_data* – pointer to the user data.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: If specified, the pointer to *user_data* is passed to all user-supplied functions called by the outer integrator for which it is an argument; otherwise `NULL` is passed.

To attach a user data block to the inner integrator call the appropriate *SetUserData* function for the inner integrator memory structure (e.g., `ARKStepSetUserData()` if the inner stepper is ARKStep). This pointer may be the same as or different from the pointer attached to the outer integrator depending on what is required by the user code.

int **MRIStepSetPreInnerFn**(void* *arkode_mem*, *MRIStepPreInnerFn* *prefn*)
Specifies the function called *before* each inner integration.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *prefn* – the name of the C function (of type `MRIStepPreInnerFn()`) defining pre inner integration function.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`

int **MRIStepSetPostInnerFn**(void* *arkode_mem*, *MRIStepPostInnerFn* *postfn*)
Specifies the function called *after* each inner integration.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *postfn* – the name of the C function (of type `MRIStepPostInnerFn()`) defining post inner integration function.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`

6.5.7.2 Optional inputs for IVP method selection

Optional input	Function name	Default
Set MRI coupling coefficients	`MRIStepSetCoupling()`	internal
Set MRI outer RK table	`MRIStepSetTable()`	internal
Specify MRI outer table number	`MRIStepSetTableNum()`	internal

int **MRIStepSetCoupling**(void* *arkode_mem*, *MRIStepCoupling* *C*)
Specifies a customized set of slow-to-fast coupling coefficients for the MRI method.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *C* – the table of coupling coefficients for the MRI method.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes:

For a description of the `MRIStepCoupling` type and related functions for creating Butcher tables see the Section *MRI Coupling Coefficients Data Structure*.

int **MRIStepSetTable**(void* *arkode_mem*, int *q*, *ARKodeButcherTable* *B*)
Specifies a customized slow Butcher table for a traditional MIS method.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *q* – global order of accuracy for the MRI method.
- *B* – the Butcher table for the outer (slow) RK method.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes:

For a description of the `ARKodeButcherTable` type and related functions for creating Butcher tables see *Butcher Table Data Structure*.

Internally, this function directly calls the utility routine `MRIStepCoupling_MIStoMRI()` to convert from the input table B to an MRI coupling table. As such, all constraints on B stated for that function apply here as well: it must have explicit first stage (i.e., $c_1 = 0$ and $A_{1,j} = 0$ for $1 \le j \le s$) and sorted abcissae (i.e., $c_i \ge c_{i-1}$ for $2 \le i \le s$).

The input value of q is used rather than the order encoded in the Butcher table since the overall order of the MRI method may differ from the order of the outer table (see equation (2.12) and surrounding discussion). No error checking is performed to ensure that q correctly describes the order of the overall MRI method.

int **MRIStepSetTableNum**(void* *arkode_mem*, int *itable*)
Indicates to use a specific built-in Butcher table or MRI coupling table for the MRI outer (slow) method.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *itable* – index of the outer (slow) Butcher or MRI table.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Depending on the value of *itable*, this routine will do one of two things.

If *itable* specifies a built-in ERK or DIRK Butcher table, then this will internally retrieve the `ARKodeButcherTable` corresponding to the *itable* argument, and then immediately call the utility routine `MRIStepSetTable()` to convert this to an MRI coupling table and store the result within MRIStep. In this case, *itable* should match one of the methods from the section *Explicit Butcher tables* or the section *Implicit Butcher tables*. Error-checking is performed to ensure that this table exists and satisfies the restrictions listed above for `MRIStepSetTable()`. This approach assumes that the overall MRI method order equals $min(q, 2)$, where q is the order of accuracy for the Butcher table indicated by *itable*; if in fact the overall method differs from this assumed value, it is recommended that the user instead call either `MRIStepSetCoupling()` or `MRIStepSetTable()` directly.

If *itable* instead specifies a built-in MRI coupling table, then this will internally retrieve the `MRIStepCoupling` table via the routine `MRIStepCoupling_LoadTable()`, and store the result in MRIStep. In this case, *itable* should match one of the methods from the section *MRIStepCoupling tables*.

Optional inputs for implicit stage solves

The mathematical explanation for the nonlinear solver strategies used by MRIStep, including how each of the parameters below is used within the code, is provided in the section *Nonlinear solver methods*.

Optional input	Function name	Default
Specify linearly implicit f^S	`MRIStepSetLinear()`	`SUNFALSE`
Specify nonlinearly implicit f^S	`MRIStepSetNonlinear()`	`SUNTRUE`
Implicit predictor method	`MRIStepSetPredictorMethod()`	0
Maximum number of nonlinear iterations	`MRIStepSetMaxNonlinIters()`	3
Coefficient in the nonlinear convergence test	`MRIStepSetNonlinConvCoef()`	0.1
Nonlinear convergence rate constant	`MRIStepSetNonlinCRDown()`	0.3
Nonlinear residual divergence ratio	`MRIStepSetNonlinRDiv()`	2.3
User-provided implicit stage predictor	`MRIStepSetStagePredictFn()`	`NULL`

int **`MRIStepSetLinear`**(void* *arkode_mem*, int *timedepend*)

Specifies that the implicit slow right-hand side function, $f^S(t, y)$ is linear in y.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *timedepend* – flag denoting whether the Jacobian of $f^S(t, y)$ is time-dependent (1) or not (0). Alternately, when using a matrix-free iterative linear solver this flag denotes time dependence of the preconditioner.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Tightens the linear solver tolerances and takes only a single Newton iteration. Calls `MRIStepSetDeltaGammaMax()` to enforce Jacobian recomputation when the step size ratio changes by more than 100 times the unit roundoff (since nonlinear convergence is not tested). Only applicable when used in combination with the modified or inexact Newton iteration (not the fixed-point solver).

The only SUNDIALS-provided SUNNonlinearSolver module that is compatible with the `MRIStepSetLinear()` option is the Newton solver.

int **MRIStepSetNonlinear**(void* *arkode_mem*)
Specifies that the implicit slow right-hand side function, $f^S(t,y)$ is nonlinear in y.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This is the default behavior of MRIStep, so the function is primarily useful to undo a previous call to `MRIStepSetLinear()`. Calls `MRIStepSetDeltaGammaMax()` to reset the step size ratio threshold to the default value.

int **MRIStepSetPredictorMethod**(void* *arkode_mem*, int *method*)
Specifies the method to use for predicting implicit solutions.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *method* – method choice ($0 \leq$ *method* ≤ 4):
  - 0 is the trivial predictor,
  - 1 is the maximum order (dense output) predictor,
  - 2 is the variable order predictor, that decreases the polynomial degree for more distant RK stages,
  - 3 is the cutoff order predictor, that uses the maximum order for early RK stages, and a first-order predictor for distant RK stages,
  - 4 is the bootstrap predictor, that uses a second-order predictor based on only information within the current step.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 0. If *method* is set to an undefined value, this default predictor will be used.

int **MRIStepSetMaxNonlinIters**(void* *arkode_mem*, int *maxcor*)
Specifies the maximum number of nonlinear solver iterations permitted per slow MRI stage within each time step.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *maxcor* – maximum allowed solver iterations per stage (> 0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value or if the SUNNONLINSOL module is `NULL`
- *ARK_NLS_OP_ERR* if the SUNNONLINSOL object returned a failure flag

Notes: The default value is 3; set *maxcor* ≤ 0 to specify this default.

int **MRIStepSetNonlinConvCoef**(void* *arkode_mem*, realtype *nlscoef*)
Specifies the safety factor used within the nonlinear solver convergence test.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nlscoef* – coefficient in nonlinear solver convergence test (> 0.0).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 0.1; set *nlscoef* ≤ 0 to specify this default.

int **MRIStepSetNonlinCRDown**(void* *arkode_mem*, realtype *crdown*)
Specifies the constant used in estimating the nonlinear solver convergence rate.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *crdown* – nonlinear convergence rate estimation constant (default is 0.3).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **MRIStepSetNonlinRDiv**(void* *arkode_mem*, realtype *rdiv*)
Specifies the nonlinear correction threshold beyond which the iteration will be declared divergent.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *rdiv* – tolerance on nonlinear correction size ratio to declare divergence (default is 2.3).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **MRIStepSetStagePredictFn**(void* *arkode_mem*, *ARKStagePredictFn PredictStage*)
Sets the user-supplied function to update the implicit stage predictor prior to execution of the nonlinear or linear solver algorithms that compute the implicit stage solution.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *PredictStage* – name of user-supplied predictor function. If `NULL`, then any previously-provided stage prediction function will be disabled.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`

Notes: See the section *Implicit stage prediction function* for more information on this user-supplied routine.

Linear solver interface optional input functions

The mathematical explanation of the linear solver methods available to MRIStep is provided in the section *Linear solver methods*. We group the user-callable routines into four categories: general routines concerning the update frequency for matrices and/or preconditioners, optional inputs for matrix-based linear solvers, optional inputs for matrix-free linear solvers, and optional inputs for iterative linear solvers. We note that the matrix-based and matrix-free groups are mutually exclusive, whereas the "iterative" tag can apply to either case.

6.6 Optional inputs for the ARKLS linear solver interface

As discussed in the section *Updating the linear solver*, ARKode strives to reuse matrix and preconditioner data for as many solves as possible to amortize the high costs of matrix construction and factorization. To that end, MRIStep provides user-callable routines to modify this behavior. Recall that the Newton system matrices that arise within an implicit stage solve are $\mathcal{A}(t,z) \approx I - \gamma J(t,z)$, where the implicit right-hand side function has Jacobian matrix $J(t,z) = \frac{\partial f^S(t,z)}{\partial z}$.

The matrix or preconditioner for $\mathcal{A}$ can only be updated within a call to the linear solver 'setup' routine. In general, the frequency with which the linear solver setup routine is called may be controlled with the *msbp* argument to `MRIStepSetLSetupFrequency()`. When this occurs, the validity of $\mathcal{A}$ for successive time steps intimately depends on whether the corresponding γ and J inputs remain valid.

At each call to the linear solver setup routine the decision to update $\mathcal{A}$ with a new value of γ, and to reuse or reevaluate Jacobian information, depends on several factors including:

- the success or failure of previous solve attempts,
- the success or failure of the previous time step attempts,
- the change in γ from the value used when constructing $\mathcal{A}$, and
- the number of steps since Jacobian information was last evaluated.

The frequency with which to update Jacobian information can be controlled with the *msbj* argument to `MRIStepSetJacEvalFrequency()`. We note that this is only checked *within* calls to the linear solver setup routine, so values *msbj* $<$ *msbp* do not make sense. For linear-solvers with user-supplied preconditioning the above factors are used to determine whether to recommend updating the Jacobian information in the preconditioner (i.e.,

whether to set *jok* to SUNFALSE in calling the user-supplied ARKLsPrecSetupFn()). For matrix-based linear solvers these factors determine whether the matrix $J(t,y) = \frac{\partial f^S(t,y)}{\partial y}$ should be updated (either with an internal finite difference approximation or a call to the user-supplied *ARKLsJacFn*); if not then the previous value is reused and the system matrix $\mathcal{A}(t,y) \approx I - \gamma J(t,y)$ is recomputed using the current γ value.

Optional input	Function name	Default
Max change in step signaling new J	*MRIStepSetDeltaGammaMax()*	0.2
Linear solver setup frequency	*MRIStepSetLSetupFrequency()*	20
Jacobian / preconditioner update frequency	*MRIStepSetJacEvalFrequency()*	51

int **MRIStepSetDeltaGammaMax**(void* *arkode_mem*, realtype *dgmax*)
Specifies a scaled step size ratio tolerance, beyond which the linear solver setup routine will be signaled.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *dgmax* – tolerance on step size ratio change before calling linear solver setup routine (default is 0.2).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is NULL
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Any non-positive parameter will imply a reset to the default value.

int **MRIStepSetLSetupFrequency**(void* *arkode_mem*, int *msbp*)
Specifies the frequency of calls to the linear solver setup routine.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *msbp* – the linear solver setup frequency.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is NULL

Notes: Positive values of **msbp** specify the linear solver setup frequency. For example, an input of 1 means the setup function will be called every time step while an input of 2 means it will be called called every other time step. If **msbp** is 0, the default value of 20 will be used. A negative value forces a linear solver step at each implicit stage.

int **MRIStepSetJacEvalFrequency**(void* *arkode_mem*, long int *msbj*)
Specifies the frequency for recomputing the Jacobian or recommending a preconditioner update.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *msbj* – the Jacobian re-computation or preconditioner update frequency.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the MRIStep memory was NULL.
- *ARKLS_LMEM_NULL* if the linear solver memory was NULL.

Notes: The Jacobian update frequency is only checked *within* calls to the linear solver setup routine, as such values of *msbj* $<$ *msbp* will result in recomputing the Jacobian every *msbp* steps. See `MRIStepSetLSetupFrequency()` for setting the linear solver steup frequency *msbp*.

Passing a value *msbj* ≤ 0 indicates to use the default value of 50.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `MRIStepSetLinearSolver()`.

6.7 Optional inputs for matrix-based `SUNLinearSolver` modules

Optional input	Function name	Default
Jacobian function	`MRIStepSetJacFn()`	DQ
Linear system function	`MRIStepSetLinSysFn()`	internal
Enable or disable linear solution scaling	`MRIStepSetLinearSolutionScaling()`	on

When using matrix-based linear solver modules, the ARKLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t, y)$ or the linear system $I - \gamma J$. The function to evaluate the Jacobian must be of type `ARKLsJacFn()`. The user can supply a custom Jacobian function, or if using a dense or banded J can use the default internal difference quotient approximation that comes with the ARKLS interface. At present, we do not supply a corresponding routine to approximate Jacobian entries in sparse matrices J. To specify a user-supplied Jacobian function *jac*, MRIStep provides the function `MRIStepSetJacFn()`. Alternatively, a function of type `ARKLsLinSysFn()` can be provided to evaluate the matrix $I - \gamma J$. By default, ARKLS uses an internal linear system function leveraging the SUNMATRIX API to form the matrix $I - \gamma J$. To specify a user-supplied linear system function *linsys*, MRIStep provides the function `MRIStepSetLinSysFn()`. In either case the matrix information will be updated infrequently to reduce matrix construction and, with direct solvers, factorization costs. As a result the value of γ may not be current and a scaling factor is applied to the solution of the linear system to account for lagged value of γ. See *Lagged matrix information* for more details. The function `MRIStepSetLinearSolutionScaling()` can be used to disable this scaling when necessary, e.g., when providing a custom linear solver that updates the matrix using the current γ as part of the solve.

The ARKLS interface passes the user data pointer to the Jacobian and linear system functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian or linear system functions, without using global data in the program. The user data pointer may be specified through `MRIStepSetUserData()`.

int **`MRIStepSetJacFn`**(void* *arkode_mem*, *ARKLsJacFn* *jac*)

Specifies the Jacobian approximation routine to be used for the matrix-based solver with the ARKLS interface.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *jac* – name of user-supplied Jacobian approximation function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This routine must be called after the ARKLS linear solver interface has been initialized through a call to `MRIStepSetLinearSolver()`.

By default, ARKLS uses an internal difference quotient function for dense and band matrices. If `NULL` is passed in for *jac*, this default is used. An error will occur if no *jac* is supplied when using other matrix types.

The function type *ARKLsJacFn()* is described in the section *User-supplied functions*.

int **MRIStepSetLinSysFn**(void* *arkode_mem*, *ARKLsLinSysFn linsys*)
Specifies the linear system approximation routine to be used for the matrix-based solver with the ARKLS interface.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *linsys* – name of user-supplied linear system approximation function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This routine must be called after the ARKLS linear solver interface has been initialized through a call to *MRIStepSetLinearSolver()*.

By default, ARKLS uses an internal linear system function that leverages the SUNMATRIX API to form the system $I - \gamma J$. If `NULL` is passed in for *linsys*, this default is used.

The function type *ARKLsLinSysFn()* is described in the section *User-supplied functions*.

int **MRIStepSetLinearSolutionScaling**(void* *arkode_mem*, booleantype *onoff*)
Enables or disables scaling the linear system solution to account for a change in γ in the linear system. For more details see *Lagged matrix information*.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *onoff* – flag to enable (`SUNTRUE`) or disable (`SUNFALSE`) scaling

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_ILL_INPUT* if the attached linear solver is not matrix-based

Notes: Linear solution scaling is enabled by default when a matrix-based linear solver is attached.

6.8 Optional inputs for matrix-free `SUNLinearSolver` modules

Optional input	Function name	Default
Jv functions (*jtimes* and *jtsetup*)	*MRIStepSetJacTimes()*	DQ, none
Jv DQ rhs function (*jtimesRhsFn*)	*MRIStepSetJacTimesRhsFn()*	fs

As described in the section *Linear solver methods*, when solving the Newton linear systems with matrix-free methods, the ARKLS interface requires a *jtimes* function to compute an approximation to the product between the Jacobian matrix $J(t, y)$ and a vector v. The user can supply a custom Jacobian-times-vector approximation function, or use the default internal difference quotient function that comes with the ARKLS interface.

A user-defined Jacobian-vector function must be of type `ARKLsJacTimesVecFn` and can be specified through a call to `MRIStepSetJacTimes()` (see the section *User-supplied functions* for specification details). As with the user-supplied preconditioner functions, the evaluation and processing of any Jacobian-related data needed by the user's Jacobian-times-vector function is done in the optional user-supplied function of type `ARKLsJacTimesSetupFn` (see the section *User-supplied functions* for specification details). As with the preconditioner functions, a pointer to the user-defined data structure, *user_data*, specified through `MRIStepSetUserData()` (or a `NULL` pointer otherwise) is passed to the Jacobian-times-vector setup and product functions each time they are called.

int **MRIStepSetJacTimes**(void* *arkode_mem*, *ARKLsJacTimesSetupFn jtsetup*, *ARKLsJacTimesVecFn jtimes*)

Specifies the Jacobian-times-vector setup and product functions.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *jtsetup* – user-defined Jacobian-vector setup function. Pass `NULL` if no setup is necessary.
- *jtimes* – user-defined Jacobian-vector product function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`.
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.
- *ARKLS_SUNLS_FAIL* if an error occurred when setting up the Jacobian-vector product in the `SUNLinearSolver` object used by the ARKLS interface.

Notes: The default is to use an internal finite difference quotient for *jtimes* and to leave out *jtsetup*. If `NULL` is passed to *jtimes*, these defaults are used. A user may specify non-`NULL` *jtimes* and `NULL` *jtsetup* inputs.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `MRIStepSetLinearSolver()`.

The function types `ARKLsJacTimesSetupFn` and `ARKLsJacTimesVecFn` are described in the section *User-supplied functions*.

When using the internal difference quotient the user may optionally supply an alternative implicit right-hand side function for use in the Jacobian-vector product approximation by calling `MRIStepSetJacTimesRhsFn()`. The alternative implicit right-hand side function should compute a suitable (and differentiable) approximation to the f^S function provided to `MRIStepCreate()`. For example, as done in *[DFWBT2010]*, the alternative function may use lagged values when evaluating a nonlinearity in f^S to avoid differencing a potentially non-differentiable factor.

int **MRIStepSetJacTimesRhsFn**(void* *arkode_mem*, *ARKRhsFn jtimesRhsFn*)

Specifies an alternative implicit right-hand side function for use in the internal Jacobian-vector product difference quotient approximation.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *jtimesRhsFn* – the name of the C function (of type `ARKRhsFn()`) defining the alternative right-hand side function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`.

- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.

Notes: The default is to use the implicit right-hand side function provided to `MRIStepCreate()` in the internal difference quotient. If the input implicit right-hand side function is `NULL`, the default is used.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to `MRIStepSetLinearSolver()`.

6.9 Optional inputs for iterative `SUNLinearSolver` modules

Optional input	Function name	Default
Newton preconditioning functions	`MRIStepSetPreconditioner()`	`NULL`, `NULL`
Newton linear and nonlinear tolerance ratio	`MRIStepSetEpsLin()`	0.05
Newton linear solve tolerance conversion factor	`MRIStepSetLSNormFactor()`	vector length

As described in the section *Linear solver methods*, when using an iterative linear solver the user may supply a preconditioning operator to aid in solution of the system. This operator consists of two user-supplied functions, *psetup* and *psolve*, that are supplied to MRIStep using the function `MRIStepSetPreconditioner()`. The *psetup* function supplied to these routines should handle evaluation and preprocessing of any Jacobian data needed by the user's preconditioner solve function, *psolve*. The user data pointer received through `MRIStepSetUserData()` (or a pointer to `NULL` if user data was not specified) is passed to the *psetup* and *psolve* functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program.

Also, as described in the section *Linear iteration error control*, the ARKLS interface requires that iterative linear solvers stop when the norm of the preconditioned residual satisfies

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}$$

where the default $\epsilon_L = 0.05$, which may be modified by the user through the `MRIStepSetEpsLin()` function.

int **`MRIStepSetPreconditioner`**(void* *arkode_mem*, *ARKLsPrecSetupFn psetup*, *ARKLsPrecSolveFn psolve*)

Specifies the user-supplied preconditioner setup and solve functions.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *psetup* – user defined preconditioner setup function. Pass `NULL` if no setup is needed.
- *psolve* – user-defined preconditioner solve function.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`.
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.
- *ARKLS_SUNLS_FAIL* if an error occurred when setting up preconditioning in the `SUNLinearSolver` object used by the ARKLS interface.

Notes: The default is `NULL` for both arguments (i.e., no preconditioning).

This function must be called *after* the ARKLS system solver interface has been initialized through a call to *MRIStepSetLinearSolver()*.

Both of the function types *ARKLsPrecSetupFn()* and *ARKLsPrecSolveFn()* are described in the section *User-supplied functions*.

int **MRIStepSetEpsLin**(void* *arkode_mem*, realtype *eplifac*)
Specifies the factor by which the tolerance on the nonlinear iteration is multiplied to get a tolerance on the linear iteration.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *eplifac* – linear convergence safety factor.

Return value:

- *ARKLS_SUCCESS* if successful.
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`.
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`.
- *ARKLS_ILL_INPUT* if an input has an illegal value.

Notes: Passing a value *eplifac* ≤ 0 indicates to use the default value of 0.05.

This function must be called *after* the ARKLS system solver interface has been initialized through a call to *MRIStepSetLinearSolver()*.

int **MRIStepSetLSNormFactor**(void* *arkode_mem*, realtype *nrmfac*)
Specifies the factor to use when converting from the integrator tolerance (WRMS norm) to the linear solver tolerance (L2 norm) for Newton linear system solves e.g., `tol_L2 = fac * tol_WRMS`.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nrmfac* – the norm conversion factor. If *nrmfac* is:

  > 0 then the provided value is used.

  $= 0$ then the conversion factor is computed using the vector length i.e., `nrmfac = sqrt(N_VGetLength(y))` (*default*).

  < 0 then the conversion factor is computed using the vector dot product i.e., `nrmfac = sqrt(N_VDotProd(v,v))` where all the entries of `v` are one.

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`.

Notes: This function must be called *after* the ARKLS system solver interface has been initialized through a call to *MRIStepSetLinearSolver()*.

The following functions can be called to set optional inputs to control the rootfinding algorithm, the mathematics of which are described in the section *Rootfinding*.

Optional input	Function name	Default
Direction of zero-crossings to monitor	*MRIStepSetRootDirection()*	both
Disable inactive root warnings	*MRIStepSetNoInactiveRootWarn()*	enabled

int **MRIStepSetRootDirection**(void* *arkode_mem*, int* *rootdir*)
Specifies the direction of zero-crossings to be located and returned.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *rootdir* – state array of length *nrtfn*, the number of root functions g_i (the value of *nrtfn* was supplied in the call to `MRIStepRootInit()`). If `rootdir[i] == 0` then crossing in either direction for g_i should be reported. A value of +1 or -1 indicates that the solver should report only zero-crossings where g_i is increasing or decreasing, respectively.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`
- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default behavior is to monitor for both zero-crossing directions.

int **MRIStepSetNoInactiveRootWarn**(void* *arkode_mem*)
Disables issuing a warning if some root function appears to be identically zero at the beginning of the integration.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`

Notes: MRIStep will not report the initial conditions as a possible zero-crossing (assuming that one or more components g_i are zero at the initial time). However, if it appears that some g_i is identically zero at the initial time (i.e., g_i is zero at the initial time *and* after the first step), MRIStep will issue a warning which can be disabled with this optional input function.

6.9.1 Interpolated output function

An optional function `MRIStepGetDky()` is available to obtain additional values of solution-related quantities. This function should only be called after a successful return from `MRIStepEvolve()`, as it provides interpolated values either of y or of its derivatives (up to the 3rd derivative) interpolated to any value of t in the last internal step taken by `MRIStepEvolve()`. Internally, this *dense output* algorithm is identical to the algorithm used for the maximum order implicit predictors, described in the section *Maximum order predictor*, except that derivatives of the polynomial model may be evaluated upon request.

int **MRIStepGetDky**(void* *arkode_mem*, realtype *t*, int *k*, N_Vector *dky*)
Computes the *k*-th derivative of the function y at the time *t*, i.e. $\frac{d^{(k)}}{dt^{(k)}}y(t)$, for values of the independent variable satisfying $t_n - h_n \le t \le t_n$, with t_n as current internal time reached, and h_n is the last internal step size successfully used by the solver. This routine uses an interpolating polynomial of degree *min(degree, 5)*, where *degree* is the argument provided to `MRIStepSetInterpolantDegree()`. The user may request *k* in the range {0,…,*min(degree, kmax)*} where *kmax* depends on the choice of interpolation module. For Hermite interpolants *kmax* = *5* and for Lagrange interpolants *kmax* = *3*.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.

- *t* – the value of the independent variable at which the derivative is to be evaluated.
- *k* – the derivative order requested.
- *dky* – output vector (must be allocated by the user).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_BAD_K* if *k* is not in the range {0,. . . ,*min(degree, kmax)*}.
- *ARK_BAD_T* if *t* is not in the interval $[t_n - h_n, t_n]$
- *ARK_BAD_DKY* if the *dky* vector was `NULL`
- *ARK_MEM_NULL* if the MRIStep memory is `NULL`

Notes: It is only legal to call this function after a successful return from `MRIStepEvolve()`.

A user may access the values t_n and h_n via the functions `MRIStepGetCurrentTime()` and `MRIStepGetLastStep()`, respectively.

6.9.2 Optional output functions

MRIStep provides an extensive set of functions that can be used to obtain solver performance information. We organize these into groups:

1. SUNDIALS version information accessor routines are in the subsection *SUNDIALS version information*,
2. General MRIStep output routines are in the subsection *Main solver optional output functions*,
3. MRIStep implicit solver output routines are in the subsection *Implicit solver optional output functions*,
4. Linear solver output routines are in the subsection *Linear solver interface optional output functions* and
5. General usability routines (e.g. to print the current MRIStep parameters, or output the current coupling table) are in the subsection *General usability functions*.
6. Output routines regarding root-finding results are in the subsection *Rootfinding optional output functions*,

Following each table, we elaborate on each function.

Some of the optional outputs, especially the various counters, can be very useful in determining the efficiency of various methods inside MRIStep. For example:

- The number of steps and right-hand side evaluations at both the slow and fast time scales provide a rough measure of the overall cost of a given run, and can be compared between runs with different solver options to suggest which set of options is the most efficient.
- The ratio *nniters/nsteps* measures the performance of the nonlinear iteration in solving the nonlinear systems at each implicit stage, providing a measure of the degree of nonlinearity in the problem. Typical values of this for a Newton solver on a general problem range from 1.1 to 1.8.
- When using a Newton nonlinear solver, the ratio *njevals/nniters* (in the case of a direct linear solver), and the ratio *npevals/nniters* (in the case of an iterative linear solver) can measure the overall degree of nonlinearity in the problem, since these are updated infrequently, unless the Newton method convergence slows.
- When using a Newton nonlinear solver, the ratio *njevals/nniters* (when using a direct linear solver), and the ratio *nliters/nniters* (when using an iterative linear solver) can indicate the quality of the approximate Jacobian or preconditioner being used. For example, if this ratio is larger for a user-supplied Jacobian or Jacobian-vector product routine than for the difference-quotient routine, it can indicate that the user-supplied Jacobian is inaccurate.

It is therefore recommended that users retrieve and output these statistics following each run, and take some time to investigate alternate solver options that will be more optimal for their particular problem of interest.

6.9.2.1 SUNDIALS version information

The following functions provide a way to get SUNDIALS version information at runtime.

int **SUNDIALSGetVersion**(char **version*, int *len*)
This routine fills a string with SUNDIALS version information.

Arguments:

- *version* – character array to hold the SUNDIALS version information.
- *len* – allocated length of the *version* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS version

Notes: An array of 25 characters should be sufficient to hold the version information.

int **SUNDIALSGetVersionNumber**(int **major*, int **minor*, int **patch*, char **label*, int *len*)
This routine sets integers for the SUNDIALS major, minor, and patch release numbers and fills a string with the release label if applicable.

Arguments:

- *major* – SUNDIALS release major version number.
- *minor* – SUNDIALS release minor version number.
- *patch* – SUNDIALS release patch version number.
- *label* – string to hold the SUNDIALS release label.
- *len* – allocated length of the *label* character array.

Return value:

- 0 if successful
- -1 if the input string is too short to store the SUNDIALS label

Notes: An array of 10 characters should be sufficient to hold the label information. If a label is not used in the release version, no information is copied to *label*.

6.9.2.2 Main solver optional output functions

Optional output	Function name
Size of MRIStep real and integer workspaces	*MRIStepGetWorkSpace()*
Cumulative number of internal steps	*MRIStepGetNumSteps()*
Step size used for the last successful step	*MRIStepGetLastStep()*
Current internal time reached by the solver	*MRIStepGetCurrentTime()*
Current internal solution reached by the solver	*MRIStepGetCurrentState()*
Current γ value used by the solver	*MRIStepGetCurrentGamma()*
Error weight vector for state variables	*MRIStepGetErrWeights()*
Suggested factor for tolerance scaling	*MRIStepGetTolScaleFactor()*
Name of constant associated with a return flag	*MRIStepGetReturnFlagName()*
No. of calls to the *fs* function	*MRIStepGetNumRhsEvals()*
Current MRI coupling tables	*MRIStepGetCurrentCoupling()*
Last inner stepper return value	*MRIStepGetLastInnerStepFlag()*

int **MRIStepGetWorkSpace**(void* *arkode_mem*, long int* *lenrw*, long int* *leniw*)
Returns the MRIStep real and integer workspace sizes.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *lenrw* – the number of `realtype` values in the MRIStep workspace.
- *leniw* – the number of integer values in the MRIStep workspace.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **MRIStepGetNumSteps**(void* *arkode_mem*, long int* *nssteps*, long int* *nfsteps*)
Returns the cumulative number of slow and fast internal steps taken by the solver (so far).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nssteps* – number of slow steps taken in the solver.
- *nfsteps* – number of fast steps taken in the solver.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **MRIStepGetLastStep**(void* *arkode_mem*, realtype* *hlast*)
Returns the integration step size taken on the last successful internal step.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *hlast* – step size taken on the last internal step.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **`MRIStepGetCurrentTime`**(void* *arkode_mem*, realtype* *tcur*)
Returns the current internal time reached by the solver.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *tcur* – current internal time reached.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **`MRIStepGetCurrentState`**(void **arkode_mem*, N_Vector **ycur*)
Returns the current internal solution reached by the solver.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *ycur* – current internal solution.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: Users should exercise extreme caution when using this function, as altering values of *ycur* may lead to undesirable behavior, depending on the particular use case and on when this routine is called.

int **`MRIStepGetCurrentGamma`**(void **arkode_mem*, realtype **gamma*)
Returns the current internal value of γ used in the implicit solver Newton matrix (see equation (2.28)).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *gamma* – current step size scaling factor in the Newton system.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **`MRIStepGetTolScaleFactor`**(void* *arkode_mem*, realtype* *tolsfac*)
Returns a suggested factor by which the user's tolerances should be scaled when too much accuracy has been requested for some internal step.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *tolsfac* – suggested scaling factor for user-supplied tolerances.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **`MRIStepGetErrWeights`**(void* *arkode_mem*, N_Vector *eweight*)
Returns the current error weight vector.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *eweight* – solution error weights at the current time.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: The user must allocate space for *eweight*, that will be filled in by this function.

char ***MRIStepGetReturnFlagName**(long int *flag*)
Returns the name of the MRIStep constant corresponding to *flag*.

Arguments:

- *flag* – a return flag from an MRIStep function.

Return value: The return value is a string containing the name of the corresponding constant.

int **MRIStepGetNumRhsEvals**(void* *arkode_mem*, long int* *nfs_evals*)
Returns the number of calls to the user's outer (slow) right-hand side function, fs (so far).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nfs_evals* – number of calls to the user's $fs(t, y)$ function.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **MRIStepGetCurrentCoupling**(void* *arkode_mem*, *MRIStepCoupling* **C*)
Returns the MRI coupling table currently in use by the solver.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *C* – pointer to slow-to-fast MRI coupling structure.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: The *MRIStepCoupling* data structure is defined in the header file `arkode/arkode_mristep.h`. It is defined as a pointer to the following C structure:

```
struct MRIStepCouplingMem {

   int nmat;         /* number of MRI coupling matrices             */
   int stages;       /* size of coupling matrices (stages * stages) */
   int q;            /* method order of accuracy                    */
   int p;            /* embedding order of accuracy                 */
   realtype ***G;    /* coupling matrices [nmat][stages][stages]    */
   realtype *c;      /* abcissae                                    */

 };
 typedef MRIStepCouplingMem *MRIStepCoupling;
```

For more details see *MRI Coupling Coefficients Data Structure*.

int **MRIStepGetLastInnerStepFlag**(void* *arkode_mem*, int* *flag*)
Returns the last return value from the inner stepper.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *flag* – inner stepper return value.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

6.9.2.3 Implicit solver optional output functions

Optional output	Function name
No. of calls to linear solver setup function	`MRIStepGetNumLinSolvSetups()`
No. of nonlinear solver iterations	`MRIStepGetNumNonlinSolvIters()`
No. of nonlinear solver convergence failures	`MRIStepGetNumNonlinSolvConvFails()`
Single accessor to all nonlinear solver statistics	`MRIStepGetNonlinSolvStats()`

int **MRIStepGetNumLinSolvSetups**(void* *arkode_mem*, long int* *nlinsetups*)
Returns the number of calls made to the linear solver's setup routine (so far).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nlinsetups* – number of linear solver setup calls made.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: This is only accumulated for the 'life' of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **MRIStepGetNumNonlinSolvIters**(void* *arkode_mem*, long int* *nniters*)
Returns the number of nonlinear solver iterations performed (so far).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nniters* – number of nonlinear iterations performed.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_NLS_OP_ERR* if the SUNNONLINSOL object returned a failure flag

Notes: This is only accumulated for the 'life' of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **MRIStepGetNumNonlinSolvConvFails**(void* *arkode_mem*, long int* *nncfails*)
Returns the number of nonlinear solver convergence failures that have occurred (so far).

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nncfails* – number of nonlinear convergence failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: This is only accumulated for the 'life' of the nonlinear solver object; the counter is reset whenever a new nonlinear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **MRIStepGetNonlinSolvStats**(void* *arkode_mem*, long int* *nniters*, long int* *nncfails*)
Returns all of the nonlinear solver statistics in a single call.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nniters* – number of nonlinear iterations performed.
- *nncfails* – number of nonlinear convergence failures.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_NLS_OP_ERR* if the SUNNONLINSOL object returned a failure flag

Notes: These are only accumulated for the 'life' of the nonlinear solver object; the counters are reset whenever a new nonlinear solver module is 'attached' to MRIStep, or when MRIStep is resized.

6.9.2.4 Linear solver interface optional output functions

The following optional outputs are available from the ARKLS modules: workspace requirements, number of calls to the Jacobian routine, number of calls to the implicit right-hand side routine for finite-difference Jacobian approximation or Jacobian-vector product approximation, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector setup and product routines, and last return value from an ARKLS function. Note that, where the name of an output would otherwise conflict with the name of an optional output from the main solver, a suffix LS (for Linear Solver) has been added here (e.g. *lenrwLS*).

Optional output	Function name
Size of real and integer workspaces	*MRIStepGetLinWorkSpace()*
No. of Jacobian evaluations	*MRIStepGetNumJacEvals()*
No. of preconditioner evaluations	*MRIStepGetNumPrecEvals()*
No. of preconditioner solves	*MRIStepGetNumPrecSolves()*
No. of linear iterations	*MRIStepGetNumLinIters()*
No. of linear convergence failures	*MRIStepGetNumLinConvFails()*
No. of Jacobian-vector setup evaluations	*MRIStepGetNumJTSetupEvals()*
No. of Jacobian-vector product evaluations	*MRIStepGetNumJtimesEvals()*
No. of *fs* calls for finite diff. J or Jv evals.	*MRIStepGetNumLinRhsEvals()*
Last return from a linear solver function	*MRIStepGetLastLinFlag()*
Name of constant associated with a return flag	*MRIStepGetLinReturnFlagName()*

int **MRIStepGetLinWorkSpace**(void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)
Returns the real and integer workspace used by the ARKLS linear solver interface.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *lenrwLS* – the number of `realtype` values in the ARKLS workspace.
- *leniwLS* – the number of integer values in the ARKLS workspace.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The workspace requirements reported by this routine correspond only to memory allocated within this interface and to memory allocated by the `SUNLinearSolver` object attached to it. The template Jacobian matrix allocated by the user outside of ARKLS is not included in this report.

In a parallel setting, the above values are global (i.e. summed over all processors).

int **MRIStepGetNumJacEvals**(void* *arkode_mem*, long int* *njevals*)
Returns the number of Jacobian evaluations.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *njevals* – number of Jacobian evaluations.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **MRIStepGetNumPrecEvals**(void* *arkode_mem*, long int* *npevals*)
Returns the total number of preconditioner evaluations, i.e. the number of calls made to *psetup* with `jok` = `SUNFALSE` and that returned `*jcurPtr` = `SUNTRUE`.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *npevals* – the current number of calls to *psetup*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **`MRIStepGetNumPrecSolves`**(void* *arkode_mem*, long int* *npsolves*)
Returns the number of calls made to the preconditioner solve function, *psolve*.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *npsolves* – the number of calls to *psolve*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **`MRIStepGetNumLinIters`**(void* *arkode_mem*, long int* *nliters*)
Returns the cumulative number of linear iterations.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nliters* – the current number of linear iterations.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **`MRIStepGetNumLinConvFails`**(void* *arkode_mem*, long int* *nlcfails*)
Returns the cumulative number of linear convergence failures.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nlcfails* – the current number of linear convergence failures.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`

- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **`MRIStepGetNumJTSetupEvals`**(void* *arkode_mem*, long int* *njtsetup*)
Returns the cumulative number of calls made to the user-supplied Jacobian-vector setup function, *jtsetup*.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *njtsetup* – the current number of calls to *jtsetup*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **`MRIStepGetNumJtimesEvals`**(void* *arkode_mem*, long int* *njvevals*)
Returns the cumulative number of calls made to the Jacobian-vector product function, *jtimes*.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *njvevals* – the current number of calls to *jtimes*.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **`MRIStepGetNumLinRhsEvals`**(void* *arkode_mem*, long int* *nfevalsLS*)
Returns the number of calls to the user-supplied implicit right-hand side function f^S for finite difference Jacobian or Jacobian-vector product approximation.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *nfevalsLS* – the number of calls to the user implicit right-hand side function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: The value *nfevalsLS* is incremented only if the default internal difference quotient function is used.

This is only accumulated for the 'life' of the linear solver object; the counter is reset whenever a new linear solver module is 'attached' to MRIStep, or when MRIStep is resized.

int **MRIStepGetLastLinFlag**(void* *arkode_mem*, long int* *lsflag*)
Returns the last return value from an ARKLS routine.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *lsflag* – the value of the last return flag from an ARKLS function.

Return value:

- *ARKLS_SUCCESS* if successful
- *ARKLS_MEM_NULL* if the MRIStep memory was `NULL`
- *ARKLS_LMEM_NULL* if the linear solver memory was `NULL`

Notes: If the ARKLS setup function failed when using the `SUNLINSOL_DENSE` or `SUNLINSOL_BAND` modules, then the value of *lsflag* is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the (dense or banded) Jacobian matrix. For all other failures, *lsflag* is negative.

Otherwise, if the ARKLS setup function failed (*MRIStepEvolve()* returned *ARK_LSETUP_FAIL*), then *lsflag* will be *SUNLS_PSET_FAIL_UNREC*, *SUNLS_ASET_FAIL_UNREC* or *SUNLS_PACKAGE_FAIL_UNREC*.

If the ARKLS solve function failed (*MRIStepEvolve()* returned *ARK_LSOLVE_FAIL*), then *lsflag* contains the error return flag from the `SUNLinearSolver` object, which will be one of: *SUNLS_MEM_NULL*, indicating that the `SUNLinearSolver` memory is `NULL`; *SUNLS_ATIMES_NULL*, indicating that a matrix-free iterative solver was provided, but is missing a routine for the matrix-vector product approximation, *SUNLS_ATIMES_FAIL_UNREC*, indicating an unrecoverable failure in the Jv function; *SUNLS_PSOLVE_NULL*, indicating that an iterative linear solver was configured to use preconditioning, but no preconditioner solve routine was provided, *SUNLS_PSOLVE_FAIL_UNREC*, indicating that the preconditioner solve function failed unrecoverably; *SUNLS_GS_FAIL*, indicating a failure in the Gram-Schmidt procedure (SPGMR and SPFGMR only); *SUNLS_QRSOL_FAIL*, indicating that the matrix R was found to be singular during the QR solve phase (SPGMR and SPFGMR only); or *SUNLS_PACKAGE_FAIL_UNREC*, indicating an unrecoverable failure in an external iterative linear solver package.

char ***MRIStepGetLinReturnFlagName**(long int *lsflag*)
Returns the name of the ARKLS constant corresponding to *lsflag*.

Arguments:

- *lsflag* – a return flag from an ARKLS function.

Return value: The return value is a string containing the name of the corresponding constant. If using the `SUNLINSOL_DENSE` or `SUNLINSOL_BAND` modules, then if $1 \le$ *lsflag* $\le n$ (LU factorization failed), this routine returns "NONE".

6.9.2.5 General usability functions

The following optional routines may be called by a user to inquire about existing solver parameters or write the current MRI coupling table. While neither of these would typically be called during the course of solving an initial value problem, these may be useful for users wishing to better understand MRIStep.

Optional routine	Function name
Output all MRIStep solver parameters	*MRIStepWriteParameters()*
Output the current MRI coupling table	*MRIStepWriteCoupling()*

int **MRIStepWriteParameters**(void* *arkode_mem*, FILE **fp*)
Outputs all MRIStep solver parameters to the provided file pointer.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *fp* – pointer to use for printing the solver parameters.

Return value:

- *ARKS_SUCCESS* if successful
- *ARKS_MEM_NULL* if the MRIStep memory was `NULL`

Notes: The *fp* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

When run in parallel, only one process should set a non-NULL value for this pointer, since parameters for all processes would be identical.

int **MRIStepWriteCoupling**(void* *arkode_mem*, FILE **fp*)
Outputs the current MRI coupling table to the provided file pointer.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *fp* – pointer to use for printing the Butcher tables.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: The *fp* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

When run in parallel, only one process should set a non-NULL value for this pointer, since tables for all processes would be identical.

6.9.2.6 Rootfinding optional output functions

Optional output	Function name
Array showing roots found	*MRIStepGetRootInfo()*
No. of calls to user root function	*MRIStepGetNumGEvals()*

int **MRIStepGetRootInfo**(void* *arkode_mem*, int* *rootsfound*)
Returns an array showing which functions were found to have a root.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *rootsfound* – array of length *nrtfn* with the indices of the user functions g_i found to have a root (the value of *nrtfn* was supplied in the call to *MRIStepRootInit()*). For $i = 0 \ldots$ *nrtfn*-1, `rootsfound[i]` is nonzero if g_i has a root, and 0 if not.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Notes: The user must allocate space for *rootsfound* prior to calling this function.

For the components of g_i for which a root was found, the sign of `rootsfound[i]` indicates the direction of zero-crossing. A value of +1 indicates that g_i is increasing, while a value of -1 indicates a decreasing g_i.

int **MRIStepGetNumGEvals**(void* *arkode_mem*, long int* *ngevals*)
Returns the cumulative number of calls made to the user's root function g.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *ngevals* – number of calls made to g so far.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

6.9.3 MRIStep re-initialization function

To reinitialize the MRIStep module for the solution of a new problem, where a prior call to *MRIStepCreate()* has been made, the user must call the function *MRIStepReInit()*. The new problem must have the same size as the previous one. This routine retains the current settings for all ARKstep module options and performs the same input checking and initializations that are done in *MRIStepCreate()*, but it performs no memory allocation as is assumes that the existing internal memory is sufficient for the new problem. A call to this re-initialization routine deletes the solution history that was stored internally during the previous integration. Following a successful call to *MRIStepReInit()*, call *MRIStepEvolve()* again for the solution of the new problem.

The use of *MRIStepReInit()* requires that the number of Runge Kutta stages for both the slow and fast methods be no larger for the new problem than for the previous problem.

One important use of the *MRIStepReInit()* function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to this routine. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **MRIStepReInit**(void* *arkode_mem*, *ARKRhsFn* *fs*, realtype *t0*, N_Vector *y0*)
Provides required problem specifications and re-initializes the MRIStep outer (slow) stepper.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *fs* – the name of the C function (of type *ARKRhsFn()*) defining the slow right-hand side function in $\dot{y} = f_s(t, y) + f_f(t, y)$.
- *t0* – the initial value of t.
- *y0* – the initial condition vector $y(t_0)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: If the inner (fast) stepper also needs to be reinitialized, its reinitialization function should be called before calling `MRIStepReInit()` to reinitialize the outer stepper.

All previously set options are retained but may be updated by calling the appropriate "Set" functions.

If an error occurred, `MRIStepReInit()` also sends an error message to the error handler function.

6.9.4 MRIStep reset function

To reset the MRIStep module to a particular independent variable value and dependent variable vector for the continued solution of a problem, where a prior call to `MRIStepCreate()` has been made, the user must call the function `MRIStepReset()`. Like `MRIStepReInit()` this routine retains the current settings for all MRIStep module options and performs no memory allocations but, unlike `MRIStepReInit()`, this routine performs only a *subset* of the input checking and initializations that are done in `MRIStepCreate()`. In particular this routine retains all internal counter values and the step size/error history and does not reinitialize the linear and/or nonlinear solver but it does indicate that a linear solver setup is necessary in the next step. Following a successful call to `MRIStepReset()`, call `MRIStepEvolve()` again to continue solving the problem. By default the next call to `MRIStepEvolve()` will use the step size computed by MRIStep prior to calling `MRIStepReset()`. To set a different step size or have MRIStep estimate a new step size use `MRIStepSetInitStep()`.

One important use of the `MRIStepReset()` function is in the treating of jump discontinuities in the RHS functions. Except in cases of fairly small jumps, it is usually more efficient to stop at each point of discontinuity and restart the integrator with a readjusted ODE model, using a call to `MRIStepReset()`. To stop when the location of the discontinuity is known, simply make that location a value of `tout`. To stop when the location of the discontinuity is determined by the solution, use the rootfinding feature. In either case, it is critical that the RHS functions *not* incorporate the discontinuity, but rather have a smooth extension over the discontinuity, so that the step across it (and subsequent rootfinding, if used) can be done efficiently. Then use a switch within the RHS functions (communicated through `user_data`) that can be flipped between the stopping of the integration and the restart, so that the restarted problem uses the new values (which have jumped). Similar comments apply if there is to be a jump in the dependent variable vector.

int **`MRIStepReset`**(void* *arkode_mem*, realtype *tR*, N_Vector *yR*)

Resets the current MRIStep outer (slow) time-stepper module state to the provided independent variable value and dependent variable vector.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *tR* – the value of the independent variable t.
- *yR* – the value of the dependent variable vector $y(t_R)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_MEM_FAIL* if a memory allocation failed
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: If the inner (fast) stepper also needs to be reset, its reset function should be called before calling `MRIStepReset()` to reset the outer stepper.

All previously set options are retained but may be updated by calling the appropriate "Set" functions.

If an error occurred, *MRIStepReset()* also sends an error message to the error handler function.

6.9.5 MRIStep system resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatially-adaptive PDE simulations under a method-of-lines approach), the MRIStep integrator may be "resized" between *slow* integration steps, through calls to the *MRIStepResize()* function. This function modifies MRIStep's internal memory structures to use the new problem size.

To aid in the vector resize operation, the user can supply a vector resize function that will take as input a vector with the previous size, and transform it in-place to return a corresponding vector of the new size. If this function (of type *ARKVecResizeFn()*) is not supplied (i.e. is set to `NULL`), then all existing vectors internal to MRIStep will be destroyed and re-cloned from the new input vector.

int **MRIStepResize**(void* *arkode_mem*, N_Vector *ynew*, realtype *t0*, *ARKVecResizeFn* *resize*, void* *resize_data*)

Re-initializes MRIStep with a different state vector.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *ynew* – the newly-sized solution vector, holding the current dependent variable values $y(t_0)$.
- *t0* – the current value of the independent variable t_0 (this must be consistent with *ynew*).
- *resize* – the user-supplied vector resize function (of type *ARKVecResizeFn()*.
- *resize_data* – the user-supplied data structure to be passed to *resize* when modifying internal MRIStep vectors.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`
- *ARK_NO_MALLOC* if *arkode_mem* was not allocated.
- *ARK_ILL_INPUT* if an argument has an illegal value.

Notes: If an error occurred, *MRIStepResize()* also sends an error message to the error handler function.

6.9.5.1 Resizing the absolute tolerance array

If using array-valued absolute tolerances, the absolute tolerance vector will be invalid after the call to *MRIStepResize()*, so the new absolute tolerance vector should be re-set **following** each call to *MRIStepResize()* through a new call to *MRIStepSVtolerances()*.

If scalar-valued tolerances or a tolerance function was specified through either *MRIStepSStolerances()* or *MRIStepWFtolerances()*, then these will remain valid and no further action is necessary.

Note: For an example showing usage of the similar *ARKStepResize()* routine, see the supplied serial C example problem, `ark_heat1D_adapt.c`.

6.10 User-supplied functions

The user-supplied functions for MRIStep consist of:

- a function that defines the slow portion of the ODE (required),
- a function that handles error and warning messages (optional),
- a function that provides the error weight vector (optional),
- a function that updates the implicit stage prediction (optional),
- a function that defines the root-finding problem(s) to solve (optional),
- one or two functions that provide Jacobian-related information for the linear solver, if the method is implicit at the slow time scale and a Newton-based nonlinear iteration is chosen (optional),
- one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms, if the method is implicit at the slow time scale and a Newton-based nonlinear iteration and iterative linear solver are chosen (optional),
- a function that handles vector resizing operations, if the underlying vector structure supports resizing (as opposed to deletion/recreation), and if the user plans to call `MRIStepResize()` (optional), and
- functions to be called before and after each inner integration to perform any communication or memory transfers of forcing data supplied by the outer integrator to the inner integrator, or state data supplied by the inner integrator to the outer integrator.

Additionally, a user may supply a custom set of slow-to-fast coupling coefficients for the MRI method.

6.10.1 ODE right-hand side

The user must supply a function of type `ARKRhsFn` to specify the "slow" right-hand side of the ODE system:

typedef int **(*ARKRhsFn)** (realtype *t*, N_Vector *y*, N_Vector *ydot*, void* *user_data*)

This function computes a portion of the ODE right-hand side for a given value of the independent variable t and state vector y.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *ydot* – the output vector that forms a portion the ODE RHS $f(t, y)$.
- *user_data* – the *user_data* pointer that was passed to `MRIStepSetUserData()`.

Return value: An *ARKRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case MRIStep will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *ARK_RHSFUNC_FAIL* is returned).

Notes: Allocation of memory for *ydot* is handled within the MRIStep module.

The vector *ydot* may be uninitialized on input; it is the user's responsibility to fill this entire vector with meaningful values.

A recoverable failure error return from the *ARKRhsFn* is typically used to flag a value of the dependent variable y that is "illegal" in some way (e.g., negative where only a non-negative value is physically meaningful). If such a return is made within an implicit solve, MRIStep may attempt to recover by repeating the nonlinear iteration in order to avoid this recoverable error return. However, since MRIStep currently requires fixed time

stepping at the slow time scale, no other recovery mechanisms are available, and MRIStep may halt on a recoverable error flag.

6.10.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by *errfp* (see `MRIStepSetErrFile()`), the user may provide a function of type `ARKErrHandlerFn` to process any such messages.

typedef void **(*ARKErrHandlerFn)**(int *error_code*, const char* *module*, const char* *function*, char* *msg*, void* *user_data*)

This function processes error and warning messages from MRIStep and its sub-modules.

Arguments:

- *error_code* – the error code.
- *module* – the name of the MRIStep module reporting the error.
- *function* – the name of the function in which the error occurred.
- *msg* – the error message.
- *user_data* – a pointer to user data, the same as the *eh_data* parameter that was passed to `MRIStepSetErrHandlerFn()`.

Return value: An *ARKErrHandlerFn* function has no return value.

Notes: *error_code* is negative for errors and positive (*ARK_WARNING*) for warnings. If a function that returns a pointer to memory encounters an error, it sets *error_code* to 0.

6.10.3 Error weight function

As an alternative to providing the relative and absolute tolerances, the user may provide a function of type `ARKEwtFn` to compute a vector *ewt* containing the weights in the WRMS norm $\|v\|_{WRMS} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(ewt_i\ v_i\right)^2\right)^{1/2}$. These weights will be used in place of those defined in the section *Error norms*.

typedef int **(*ARKEwtFn)**(N_Vector *y*, N_Vector *ewt*, void* *user_data*)

This function computes the WRMS error weights for the vector y.

Arguments:

- *y* – the dependent variable vector at which the weight vector is to be computed.
- *ewt* – the output vector containing the error weights.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.

Return value: An *ARKEwtFn* function must return 0 if it successfully set the error weights, and -1 otherwise.

Notes: Allocation of memory for *ewt* is handled within MRIStep.

The error weight vector must have all components positive. It is the user's responsibility to perform this test and return -1 if it is not satisfied.

6.10.4 Implicit stage prediction function

A user may supply a function to update the prediction for each implicit stage solution. If supplied, this routine will be called *after* any existing MRIStep predictor algorithm completes, so that the predictor may be modified by the user as desired. In this scenario, a user may provide a function of type `ARKStagePredictFn` to provide this implicit predictor to MRIStep. This function takes as input the already-predicted implicit stage solution and the corresponding 'time' for that prediction; it then updates the prediction vector as desired. If the user-supplied routine will construct a full prediction (and thus the MRIStep prediction is irrelevant), it is recommended that the user *not* call `MRIStepSetPredictorMethod()`, thereby leaving the default trivial predictor in place.

typedef int **(*ARKStagePredictFn)** (realtype *t*, N_Vector *zpred*, void* *user_data*)

This function updates the prediction for the implicit stage solution.

Arguments:

- *t* – the current value of the independent variable.
- *zpred* – the MRIStep-predicted stage solution on input, and the user-modified predicted stage solution on output.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.

Return value: An *ARKStagePredictFn* function should return 0 if it successfully set the upcoming stable step size, and a non-zero value otherwise.

Notes: This may be useful if there are bound constraints on the solution, and these should be enforced prior to beginning the nonlinear or linear implicit solver algorithm.

6.10.5 Rootfinding function

If a rootfinding problem is to be solved during the integration of the ODE system, the user must supply a function of type `ARKRootFn`.

typedef int **(*ARKRootFn)** (realtype *t*, N_Vector *y*, realtype* *gout*, void* *user_data*)

This function implements a vector-valued function $g(t, y)$ such that the roots of the *nrtfn* components $g_i(t, y)$ are sought.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *gout* – the output array, of length *nrtfn*, with components $g_i(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.

Return value: An *ARKRootFn* function should return 0 if successful or a non-zero value if an error occurred (in which case the integration is halted and MRIStep returns *ARK_RTFUNC_FAIL*).

Notes: Allocation of memory for *gout* is handled within MRIStep.

6.10.6 Jacobian construction (matrix-based linear solvers)

If a matrix-based linear solver module is used (i.e., a non-NULL `SUNMatrix` object was supplied to `MRIStepSetLinearSolver()` in section *A skeleton of the user's main program*), the user may provide a function of type `ARKLsJacFn` to provide the Jacobian approximation or `ARKLsLinSysFn` to provide an approximation of the linear system $A = I - \gamma J$.

typedef int **(*ARKLsJacFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, SUNMatrix *Jac*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the Jacobian matrix $J = \frac{\partial f^S}{\partial y}$ (or an approximation to it).

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector, namely the predicted value of $y(t)$.
- *fy* – the current value of the vector $f^S(t, y)$.
- *Jac* – the output Jacobian matrix.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.
- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used by an ARKLsJacFn as temporary storage or work space.

Return value: An *ARKLsJacFn* function should return 0 if successful, a positive value if a recoverable error occurred (in which case MRIStep will attempt to correct, while ARKLS sets *last_flag* to *ARKLS_JACFUNC_RECVR*), or a negative value if it failed unrecoverably (in which case the integration is halted, `MRIStepEvolve()` returns *ARK_LSETUP_FAIL* and ARKLS sets *last_flag* to *ARKLS_JACFUNC_UNRECVR*).

Notes: Information regarding the structure of the specific `SUNMatrix` structure (e.g.~number of rows, upper/lower bandwidth, sparsity type) may be obtained through using the implementation-specific `SUNMatrix` interface functions (see the section *Matrix Data Structures* for details).

When using a linear solver of type `SUNLINEARSOLVER_DIRECT`, prior to calling the user-supplied Jacobian function, the Jacobian matrix $J(t, y)$ is zeroed out, so only nonzero elements need to be loaded into *Jac*.

With the default nonlinear solver (the native SUNDIALS Netwon method), each call to the user's `ARKLsJacFn()` function is preceded by a call to the implicit `ARKRhsFn()` user function with the same (t, y) arguments. Thus, the Jacobian function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see *Functions provided by SUNDIALS integrators* for more information).

If the user's `ARKLsJacFn` function uses difference quotient approximations, then it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the MRIStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

dense:

A user-supplied dense Jacobian function must load the *N* by *N* dense matrix *Jac* with an approximation to the Jacobian matrix $J(t, y)$ at the point (t, y). The accessor macros `SM_ELEMENT_D` and `SM_COLUMN_D` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the SUNMATRIX_DENSE type. `SM_ELEMENT_D(J, i, j)` references the `(i,j)`-th element of the dense matrix `J` (for `i`, `j` between 0 and N-1). This macro is meant for small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to *N*, the Jacobian element $J_{m,n}$ can be set using the statement `SM_ELEMENT_D(J, m-1, n-1)` $= J_{m,n}$. Alternatively, `SM_COLUMN_D(J, j)` returns a pointer to the first element of the `j`-th column of `J` (for `j` ranging from 0 to *N*-1), and the elements of the `j`-th column can then be accessed using ordinary array indexing. Consequently, $J_{m,n}$ can be loaded using the statements `col_n = SM_COLUMN_D(J, n-1); col_n[m-1]` $= J_{m,n}$. For large problems, it is more efficient to use `SM_COLUMN_D` than to use `SM_ELEMENT_D`. Note that both of these macros number rows and columns starting from 0. The SUNMATRIX_DENSE type and accessor macros are documented in section *The SUNMATRIX_DENSE Module*.

band:

A user-supplied banded Jacobian function must load the band matrix *Jac* with the elements of the Jacobian $J(t,y)$ at the point (t,y). The accessor macros `SM_ELEMENT_B`, `SM_COLUMN_B`, and `SM_COLUMN_ELEMENT_B` allow the user to read and write band matrix elements without making specific references to the underlying representation of the SUNMATRIX_BAND type. `SM_ELEMENT_B(J, i, j)` references the `(i,j)`-th element of the band matrix `J`, counting from 0. This macro is meant for use in small problems for which efficiency of access is not a major concern. Thus, in terms of the indices m and n ranging from 1 to *N* with (m,n) within the band defined by *mupper* and *mlower*, the Jacobian element $J_{m,n}$ can be loaded using the statement `SM_ELEMENT_B(J, m-1, n-1)` $= J_{m,n}$. The elements within the band are those with *-mupper* $\leq m-n \leq$ *mlower*. Alternatively, `SM_COLUMN_B(J, j)` returns a pointer to the diagonal element of the `j`-th column of `J`, and if we assign this address to `realtype *col_j`, then the `i`-th element of the `j`-th column is given by `SM_COLUMN_ELEMENT_B(col_j, i, j)`, counting from 0. Thus, for (m,n) within the band, $J_{m,n}$ can be loaded by setting `col_n = SM_COLUMN_B(J, n-1);` `SM_COLUMN_ELEMENT_B(col_n, m-1, n-1)` $= J_{m,n}$. The elements of the `j`-th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type SUNMATRIX_BAND. The array `col_n` can be indexed from *-mupper* to *mlower*. For large problems, it is more efficient to use `SM_COLUMN_B` and `SM_COLUMN_ELEMENT_B` than to use the `SM_ELEMENT_B` macro. As in the dense case, these macros all number rows and columns starting from 0. The SUNMATRIX_BAND type and accessor macros are documented in section *The SUNMATRIX_BAND Module*.

sparse:

A user-supplied sparse Jacobian function must load the compressed-sparse-column (CSC) or compressed-sparse-row (CSR) matrix *Jac* with an approximation to the Jacobian matrix $J(t,y)$ at the point (t,y). Storage for *Jac* already exists on entry to this function, although the user should ensure that sufficient space is allocated in *Jac* to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and index arrays as needed. The amount of allocated space in a SUNMATRIX_SPARSE object may be accessed using the macro `SM_NNZ_S` or the routine *`SUNSparseMatrix_NNZ()`*. The SUNMATRIX_SPARSE type is further documented in the section *The SUNMATRIX_SPARSE Module*.

typedef int **(*ARKLsLinSysFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, SUNMatrix *A*, SUNMatrix *M*, booleantype *jok*, booleantype **jcur*, realtype *gamma*, void **user_data*, N_Vector *tmp1*, N_Vector *tmp2*, N_Vector *tmp3*)

This function computes the linear system matrix $A = I - \gamma J$ (or an approximation to it).

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector, namely the predicted value of $y(t)$.
- *fy* – the current value of the vector $f^S(t,y)$.
- *A* – the output linear system matrix.
- *M* – the argument will be `NULL` since MRIStep does not support non-identity mass matrices.
- *jok* – is an input flag indicating whether the Jacobian-related data needs to be updated. The *jok* argument provides for the reuse of Jacobian data. When *jok* = `SUNFALSE`, the Jacobian-related data should be recomputed from scratch. When *jok* = `SUNTRUE` the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of *gamma*). A call with *jok* = `SUNTRUE` can only occur after a call with *jok* = `SUNFALSE`.
- *jcur* – is a pointer to a flag which should be set to `SUNTRUE` if Jacobian data was recomputed, or set to `SUNFALSE` if Jacobian data was not recomputed, but saved data was still reused.
- *gamma* – the scalar γ appearing in the Newton matrix given by $A = I - \gamma J$.

- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.
- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used by an ARKLsLinSysFn as temporary storage or work space.

Return value: An *ARKLsLinSysFn* function should return 0 if successful, a positive value if a recoverable error occurred (in which case MRIStep will attempt to correct, while ARKLS sets *last_flag* to *ARKLS_JACFUNC_RECVR*), or a negative value if it failed unrecoverably (in which case the integration is halted, `MRIStepEvolve()` returns *ARK_LSETUP_FAIL* and ARKLS sets *last_flag* to *ARKLS_JACFUNC_UNRECVR*).

6.10.7 Jacobian-vector product (matrix-free linear solvers)

When using a matrix-free linear solver module for the implicit stage solves (i.e., a NULL-valued SUNMATRIX argument was supplied to `MRIStepSetLinearSolver()` in the section *A skeleton of the user's main program*), the user may provide a function of type `ARKLsJacTimesVecFn` in the following form, to compute matrix-vector products Jv. If such a function is not supplied, the default is a difference quotient approximation to these products.

typedef int **(*ARKLsJacTimesVecFn)**(N_Vector *v*, N_Vector *Jv*, realtype *t*, N_Vector *y*, N_Vector *fy*, void* *user_data*, N_Vector *tmp*)

This function computes the product $Jv = \left(\frac{\partial f^S}{\partial y}\right) v$ (or an approximation to it).

Arguments:

- *v* – the vector to multiply.
- *Jv* – the output vector computed.
- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^S(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.
- *tmp* – pointer to memory allocated to a variable of type `N_Vector` which can be used as temporary storage or work space.

Return value: The value to be returned by the Jacobian-vector product function should be 0 if successful. Any other return value will result in an unrecoverable error of the generic Krylov solver, in which case the integration is halted.

Notes: If the user's `ARKLsJacTimesVecFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the MRIStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

6.10.8 Jacobian-vector product setup (matrix-free linear solvers)

If the user's Jacobian-times-vector routine requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type `ARKLsJacTimesSetupFn`, defined as follows:

typedef int **(*ARKLsJacTimesSetupFn)**(realtype *t*, N_Vector *y*, N_Vector *fy*, void* *user_data*)

This function preprocesses and/or evaluates any Jacobian-related data needed by the Jacobian-times-vector routine.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^S(t, y)$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.

Return value: The value to be returned by the Jacobian-vector setup function should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: Each call to the Jacobian-vector setup function is preceded by a call to the implicit `ARKRhsFn` user function with the same (t, y) arguments. Thus, the setup function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side.

If the user's `ARKLsJacTimesSetupFn` function uses difference quotient approximations, it may need to access quantities not in the argument list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the MRIStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

6.10.9 Preconditioner solve (iterative linear solvers)

If a user-supplied preconditioner is to be used with a SUNLinSol solver module, then the user must provide a function of type `ARKLsPrecSolveFn` to solve the linear system $Pz = r$, where P corresponds to either a left or right preconditioning matrix. Here P should approximate (at least crudely) the Newton matrix $A = I - \gamma J$, where $J = \frac{\partial f^S}{\partial y}$ If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate A.

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

This function solves the preconditioner system $Pz = r$.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^S(t, y)$.
- *r* – the right-hand side vector of the linear system.
- *z* – the computed output solution vector.
- *gamma* – the scalar γ appearing in the Newton matrix given by $A = I - \gamma J$.
- *delta* – an input tolerance to be used if an iterative method is employed in the solution. In that case, the residual vector $Res = r - Pz$ of the system should be made to be less than *delta* in the weighted l_2 norm, i.e. $\left(\sum_{i=1}^{n} (Res_i * ewt_i)^2\right)^{1/2} < \delta$, where $\delta =$ *delta*. To obtain the `N_Vector` *ewt*, call `MRIStepGetErrWeights()`.
- *lr* – an input flag indicating whether the preconditioner solve is to use the left preconditioner (*lr* = 1) or the right preconditioner (*lr* = 2).
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `MRIStepSetUserData()`.

Return value: The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

6.10.10 Preconditioner setup (iterative linear solvers)

If the user's preconditioner routine requires that any data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type *ARKLsPrecSetupFn*.

typedef int **(*ARKLsPrecSetupFn)**(realtype *t*, N_Vector *y*, N_Vector *fy*, booleantype *jok*, booleantype* *jcurPtr*, realtype *gamma*, void* *user_data*)

This function preprocesses and/or evaluates Jacobian-related data needed by the preconditioner.

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *fy* – the current value of the vector $f^S(t, y)$.
- *jok* – is an input flag indicating whether the Jacobian-related data needs to be updated. The *jok* argument provides for the reuse of Jacobian data in the preconditioner solve function. When *jok* = `SUNFALSE`, the Jacobian-related data should be recomputed from scratch. When *jok* = `SUNTRUE` the Jacobian data, if saved from the previous call to this function, can be reused (with the current value of *gamma*). A call with *jok* = `SUNTRUE` can only occur after a call with *jok* = `SUNFALSE`.
- *jcurPtr* – is a pointer to a flag which should be set to `SUNTRUE` if Jacobian data was recomputed, or set to `SUNFALSE` if Jacobian data was not recomputed, but saved data was still reused.
- *gamma* – the scalar γ appearing in the Newton matrix given by $A = I - \gamma J$.
- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to *MRIStepSetUserData()*.

Return value: The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error (in which case the step will be retried), or negative for an unrecoverable error (in which case the integration is halted).

Notes: The operations performed by this function might include forming a crude approximate Jacobian, and performing an LU factorization of the resulting approximation to $A = I - \gamma J$.

With the default nonlinear solver (the native SUNDIALS Netwon method), each call to the preconditioner setup function is preceded by a call to the implicit *ARKRhsFn* user function with the same (t, y) arguments. Thus, the preconditioner setup function can use any auxiliary data that is computed and saved during the evaluation of the implicit ODE right-hand side. In the case of a user-supplied or external nonlinear solver, this is also true if the nonlinear system function is evaluated prior to calling the linear solver setup function (see *Functions provided by SUNDIALS integrators* for more information).

This function is not called in advance of every call to the preconditioner solve function, but rather is called only as often as needed to achieve convergence in the Newton iteration.

If the user's *ARKLsPrecSetupFn* function uses difference quotient approximations, it may need to access quantities not in the call list. These include the current step size, the error weights, etc. To obtain these, the user will need to add a pointer to the `ark_mem` structure to their `user_data`, and then use the MRIStepGet* functions listed in *Optional output functions*. The unit roundoff can be accessed as `UNIT_ROUNDOFF`, which is defined in the header file `sundials_types.h`.

6.10.11 Vector resize function

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when using spatial adaptivity in a PDE simulation), the MRIStep integrator may be "resized" between integration steps, through calls to the `MRIStepResize()` function. Typically, when performing adaptive simulations the solution is stored in a customized user-supplied data structure, to enable adaptivity without repeated allocation/deallocation of memory. In these scenarios, it is recommended that the user supply a customized vector kernel to interface between SUNDIALS and their problem-specific data structure. If this vector kernel includes a function of type `ARKVecResizeFn` to resize a given vector implementation, then this function may be supplied to `MRIStepResize()` so that all internal MRIStep vectors may be resized, instead of deleting and re-creating them at each call. This resize function should have the following form:

typedef int **(*ARKVecResizeFn)**(N_Vector *y*, N_Vector *ytemplate*, void* *user_data*)

This function resizes the vector *y* to match the dimensions of the supplied vector, *ytemplate*.

Arguments:

- *y* – the vector to resize.
- *ytemplate* – a vector of the desired size.
- *user_data* – a pointer to user data, the same as the *resize_data* parameter that was passed to `MRIStepResize()`.

Return value: An *ARKVecResizeFn* function should return 0 if it successfully resizes the vector *y*, and a non-zero value otherwise.

Notes: If this function is not supplied, then MRIStep will instead destroy the vector *y* and clone a new vector *y* off of *ytemplate*.

6.10.12 Pre inner integrator communication function

The user may supply a function of type `MRIStepPreInnerFn` that will be called *before* each inner integration to perform any communication or memory transfers of forcing data supplied by the outer integrator to the inner integrator for the inner integration.

typedef int **(*MRIStepPreInnerFn)**(realtype *t*, N_Vector* *f*, int *num_vecs*, void* *user_data*)

Arguments:

- *t* – the current value of the independent variable.
- *f* – an `N_Vector` array of outer forcing vectors.
- *num_vecs* – the number of vectors in the `N_Vector` array.
- *user_data* – the *user_data* pointer that was passed to `MRIStepSetUserData()`.

Return value: An *MRIStepPreInnerFn* function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred. As the MRIStep module only supports fixed step sizes at this time any non-zero return value will halt the integration.

Notes: In a heterogeneous computing environment if any data copies between the host and device vector data are necessary, this is where that should occur.

6.10.13 Post inner integrator communication function

The user may supply a function of type `MRIStepPostInnerFn` that will be called *after* each inner integration to perform any communication or memory transfers of state data supplied by the inner integrator to the outer integrator for the outer integration.

typedef int **(*MRIStepPostInnerFn)** (realtype *t*, N_Vector *y*, void* *user_data*)

Arguments:

- *t* – the current value of the independent variable.
- *y* – the current value of the dependent variable vector.
- *user_data* – the *user_data* pointer that was passed to `MRIStepSetUserData()`.

Return value: An *MRIStepPostInnerFn* function should return 0 if successful, a positive value if a recoverable error occurred, or a negative value if an unrecoverable error occurred. As the MRIStep module only supports fixed step sizes at this time any non-zero return value will halt the integration.

Notes: In a heterogeneous computing environment if any data copies between the host and device vector data are necessary, this is where that should occur.

tocdepth 3

6.10.14 MRI Coupling Coefficients Data Structure

As mentioned in the section *MRIStep User-callable functions*, a user may supply a custom set of coupling coefficients between fast and slow time scales (through `MRIStepSetCoupling()`). MRIStep uses a custom data type, `MRIStepCoupling`, to store these coefficients, and provides several related utility routines. As described in the Section *MRIStep – Multirate infinitesimal step methods*, the coupling from slow to fast time scales in MRI methods may be encoded by a vector of slow 'stage time' abcissae, $c^S \in \mathbb{R}^{s+1}$ and a set of coupling matrices $\Gamma^{\{k\}} \in \mathbb{R}^{(s+1)\times(s+1)}$. We therefore define the `MRIStepCoupling` structure to be

typedef MRIStepCouplingMem ***MRIStepCoupling**

where `MRIStepCouplingMem` is the structure

```
struct MRIStepCouplingMem {

  int nmat;
  int stages;
  int q;
  int p;
  realtype ***G;
  realtype *c;

};
```

Here,

- `nmat` corresponds to the number of $\Gamma^{\{k\}}$ matrices used for coupling,
- `stages` is the number of entries in `c`, i.e., $s+1$ above,
- `q` and `p` indicate the orders of accuracy for both the MRI method and the embedding, respectively,
- `G` is a three-dimensional array with dimensions `[nmat][stages][stages]` containing the set of $\Gamma^{\{k\}}$ matrices, and
- `c` is an array of length `stages` containing slow abcissae c^S for the MRI method.

6.10.14.1 MRIStepCoupling functions

Function name	Description
MRIStepCoupling_LoadTable()	Loads a pre-defined MRIStepCoupling table
MRIStepCoupling_Alloc()	Allocate an empty MRIStepCoupling table
MRIStepCoupling_Create()	Create a new MRIStepCoupling table from coefficients
MRIStepCoupling_MIStoMRI()	Create a new MRIStepCoupling table from a slow Butcher table
MRIStepCoupling_Copy()	Create a copy of a MRIStepCoupling table
MRIStepCoupling_Space()	Get the MRIStepCoupling table real and integer workspace sizes
MRIStepCoupling_Free()	Deallocate a MRIStepCoupling table
MRIStepCoupling_Write()	Write the MRIStepCoupling table to an output file

MRIStepCoupling **MRIStepCoupling_LoadTable** (int *imethod*)

Retrieves a specified MRIStepCoupling table. The prototype for this function, as well as the integer names for each provided method, are defined in top of the header file `arkode/arkode_mristep.h`. For further information on the current set of coupling tables and their corresponding identifiers, see *MRIStepCoupling tables*.

Arguments:

- *itable* – MRIStepCoupling table identifier to load.

Return value:

- *MRIStepCoupling* structure if successful.
- `NULL` pointer if *itable* was invalid or an allocation error occured.

MRIStepCoupling **MRIStepCoupling_Alloc** (int *nmat*, int *stages*)

Allocates an empty MRIStepCoupling table.

Arguments:

- *nmat* – number of $\Gamma^{\{k\}}$ matrices in the coupling table.
- *stages* – number of stages in the coupling table.

Return value:

- *MRIStepCoupling* structure if successful.
- `NULL` pointer if *stages* was invalid or an allocation error occured.

MRIStepCoupling **MRIStepCoupling_Create** (int *nmat*, int *stages*, int *q*, int *p*, realtype **G*, realtype **c*)

Allocates an MRIStepCoupling table and fills it with the given values.

Arguments:

- *nmat* – number of $\Gamma^{\{k\}}$ matrices in the coupling table.
- *stages* – number of stages in the MRI method.
- *q* – global order of accuracy for the MRI method.
- *p* – global order of accuracy for the embedded MRI method.
- *G* – array of coefficients defining the $\Gamma^{\{k\}}$ matrices. This should be stored as a 1D array of size *nmat*stages*stages*, in row-major order.
- *c* – array (of length *stages*) of slow abcissae for the MRI method.

Return value:

- *MRIStepCoupling* structure if successful.

- NULL pointer if *stages* was invalid or an allocation error occured.

Notes: As embeddings are not currently supported in MRIStep, then p should be equal to zero.

MRIStepCoupling **MRIStepCoupling_MIStoMRI** (*ARKodeButcherTable* B, int q, int p)
Creates an MRI coupling table for a traditional MIS method based on the slow Butcher table B, following the formula shown in (2.11).

Arguments:

- B – the ARKodeButcherTable for the 'slow' MIS method.
- q – the overall order of the MIS/MRI method.
- p – the overall order of the MIS/MRI embedding.

Return value:

- MRIStepCoupling structure if successful.
- NULL pointer an allocation error occured.

Notes: The s-stage slow Butcher table must have an explicit first stage (i.e., $c_1 = 0$ and $A_{1,j} = 0$ for $1 \le j \le s$) and sorted abcissae (i.e., $c_i \ge c_{i-1}$ for $2 \le i \le s$).

Since an MIS method is at most third order accurate, and even then only if it meets certain compatibility criteria (see (2.12)), the values of q and p may differ from the method and embedding orders of accuracy for the Runge–Kutta encoded in B, which is why these arguments should be supplied separately.

As embeddings are not currently supported in MRIStep, then p should be equal to zero.

MRIStepCoupling **MRIStepCoupling_Copy** (*MRIStepCoupling* C)
Creates copy of the given coupling table.

Arguments:

- C – the coupling table to copy.

Return value:

- MRIStepCoupling structure if successful.
- NULL pointer an allocation error occured.

void **MRIStepCoupling_Space** (*MRIStepCoupling* C, sunindextype **liw*, sunindextype **lrw*)
Get the real and integer workspace size for an MRIStepCoupling table.

Arguments:

- C – the coupling table.
- *lenrw* – the number of realtype values in the coupling table workspace.
- *leniw* – the number of integer values in the coupling table workspace.

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_MEM_NULL* if the Butcher table memory was NULL.

void **MRIStepCoupling_Free** (*MRIStepCoupling* C)
Deallocate the coupling table memory.

Arguments:

- C – the MRIStepCoupling table.

void **MRIStepCoupling_Write**(*MRIStepCoupling C*, FILE **outfile*)
Write the coupling table to the provided file pointer.

Arguments:

- *C* – the MRIStepCoupling table.
- *outfile* – pointer to use for printing the table.

Notes: The *outfile* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

6.10.14.2 MRIStepCoupling tables

MRIStep currently includes two classes of MRI coupling tables: those that encode methods that are explicit at the slow time scale, and those that are diagonally-implicit and solve-decoupled at the slow time scale. We list the current identifiers, multirate order of accuracy, and relevant references for each in the tables below. For the implicit methods, we also list the number of implicit solves per step that are required at the slow time scale.

Explicit MRI coupling tables:

Table name	Order	Reference
`MIS_KW3`	3	*[SKAW2009]*
`MRI_GARK_ERK45a`	4	*[S2019]*

Diagonally-implicit, solve-decoupled MRI coupling tables:

Table name	Order	Implicit Solves	Reference
`MRI_GARK_IRK21a`	2	1	*[S2019]*
`MRI_GARK_ESDIRK34a`	4	3	*[S2019]*

Chapter 7

Using ARKode for Fortran Applications

Fortran 2003 interfaces to each of the time-stepping modules as well as a Fortran 77 style interface to the ARKStep time-stepping module are provided to support the use of ARKode, for the solution of ODE systems, in a mixed Fortran/C setting. While ARKode is written in C, it is assumed here that the user's calling program and user-supplied problem-defininig rotuines are written in Fortran.

7.1 ARKode Fortran 2003 Interface Modules

The ARKode Fortran 2003 modules define interfaces to most of the ARKode C API using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. AKRode provides four Fortran 2003 modules:

- `farkode_arkstep_mod`, `farkode_erkstep_mod`, `farkode_mristep_mod` provide interfaces to the ARKStep, ERKStep, and MRIStep time-stepping modules respectively
- `farkode_mod` which interfaces to the components of ARKode which are shared by the time-stepping modules

All interfaced functions are named after the corresponding C function, but with a leading 'F'. For example. the ARKStep function `ARKStepCreate` is interfaced as `FARKStepCreate`. Thus, the steps to use an ARKode time-stepping module from Fortran are identical (ignoring language differences) to using it from C/C++.

The Fortran 2003 ARKode interface modules can be accessed by the `use` statement, i.e. `use farkode_mod`, and linking to the library `libsundials_farkode_mod.lib` in addition to `libsundials_farkode.lib`. Further information on the location of installed modules is provided in the Chapter *ARKode Installation Procedure*.

The Fortran 2003 interface modules were generated with SWIG Fortran, a fork of SWIG *[JPE2019]*. Users who are interested in the SWIG code used in the generation process should contact the SUNDIALS development team.

7.1.1 SUNDIALS Fortran 2003 Interface Modules

All of the generic SUNDIALS modules provide Fortran 2003 interface modules. Many of the generic module implementations provide Fortran 2003 interfaces (a complete list of modules with Fortran 2003 interfaces is given in *Table: SUNDIALS Fortran 2003 Interface Modules*. A module can be accessed with the `use` statement, e.g. `use fnvector_openmp_mod`, and linking to the Fortran 2003 library in addition to the C library, e.g. `libsundials_fnvecpenmp_mod.lib` and `libsundials_nvecopenmp.lib`.

The Fortran 2003 interfaces leverage the `iso_c_binding` module and the `bind(C)` attribute to closely follow the SUNDIALS C API (ignoring language differences). The generic SUNDIALS structures, e.g. `N_Vector`, are

interfaced as Fortran derived types, and function signatures are matched but with an `F` prepending the name, e.g. `FN_VConst` instead of `N_VConst`. Constants are named exactly as they are in the C API. Accordingly, using SUNDIALS via the Fortran 2003 interfaces looks just like using it in C. Some caveats stemming from the language differences are discussed in the section *Notable Fortran/C usage differences*. A discussion on the topic of equivalent data types in C and Fortran 2003 is presented in section *Data Types*.

Further information on the Fortran 2003 interfaces specific to modules is given in the NVECTOR, SUNMatrix, SUNLinearSolver, and SUNNonlinearSolver sections alongside the C documentation (chapters *Vector Data Structures*, *Matrix Data Structures*, *Description of the SUNLinearSolver module*, and *Description of the SUNNonlinearSolver Module* respectively). For details on where the Fortran 2003 module (`.mod`) files and libraries are installed see Appendix *ARKode Installation Procedure*.

7.1.1.1 Table: SUNDIALS Fortran 2003 Interface Modules

Module	Fortran 2003 Module Name
NVECTOR	`fsundials_nvector_mod`
NVECTOR_SERIAL	`fnvector_serial_mod`
NVECTOR_OPENMP	`fnvector_openmp_mod`
NVECTOR_PTHREADS	`fnvector_pthreads_mod`
NVECTOR_PARALLEL	`fnvector_parallel_mod`
NVECTOR_PARHYP	Not interfaced
NVECTOR_PETSC	Not interfaced
NVECTOR_CUDA	Not interfaced
NVECTOR_RAJA	Not interfaced
NVECTOR_MANVECTOR	`fnvector_manyvector_mod`
NVECTOR_MPIMANVECTOR	`fnvector_mpimanyvector_mod`
NVECTOR_MPIPLUSX	`fnvector_mpiplusx_mod`
SUNMATRIX	`fsundials_matrix_mod`
SUNMATRIX_BAND	`fsunmatrix_band_mod`
SUNMATRIX_DENSE	`fsunmatrix_dense_mod`
SUNMATRIX_SPARSE	`fsunmatrix_sparse_mod`
SUNLINSOL	`fsundials_linearsolver_mod`
SUNLINSOL_BAND	`fsunlinsol_band_mod`
SUNLINSOL_DENSE	`fsunlinsol_dense_mod`
SUNLINSOL_LAPACKBAND	Not interfaced
SUNLINSOL_LAPACKDENSE	Not interfaced
SUNLINSOL_KLU	`fsunlinsol_klu_mod`
SUNLINSOL_SLUMT	Not interfaced
SUNLINSOL_SLUDIST	Not interfaced
SUNLINSOL_SPGMR	`fsunlinsol_spgmr_mod`
SUNLINSOL_SPFGMR	`fsunlinsol_spfgmr_mod`
SUNLINSOL_SPBCGS	`fsunlinsol_spbcgs_mod`
SUNLINSOL_SPTFQMR	`fsunlinsol_sptfqmr_mod`
SUNLINSOL_PCG	`fsunlinsol_pcg_mof`
SUNNONLINSOL	`fsundials_nonlinearsolver_mod`
SUNNONLINSOL_NEWTON	`fsunnonlinsol_newton_mod`
SUNNONLINSOL_FIXEDPOINT	`fsunnonlinsol_fixedpoint_mod`

7.1.2 Data Types

Generally, the Fortran 2003 type that is equivalent to the C type is what one would expect. Primitive types map to the `iso_c_binding` type equivalent. SUNDIALS generic types map to a Fortran derived type. However, the handling of pointer types is not always clear as they can depend on the parameter direction. ref:*Fortran2003.DataTypesTable* presents a summary of the type equivalencies with the parameter direction in mind.

NOTE: Currently, the Fortran 2003 interfaces are only compatible with SUNDIALS builds where the `realtype` is double-precision the `sunindextype` size is 64-bits.

7.1.2.1 Table: C/Fortran-2003 Equivalent Types

C Type	Parameter Direction	Fortran 2003 type
`double`	in, inout, out, return	`real(c_double)`
`int`	in, inout, out, return	`integer(c_int)`
`long`	in, inout, out, return	`integer(c_long)`
`booleantype`	in, inout, out, return	`integer(c_int)`
`realtype`	in, inout, out, return	`real(c_double)`
`sunindextype`	in, inout, out, return	`integer(c_long)`
`double*`	in, inout, out	`real(c_double), dimension(*)`
`double*`	return	`real(c_double), pointer, dimension(:)`
`int*`	in, inout, out	`real(c_int), dimension(*)`
`int*`	return	`real(c_int), pointer, dimension(:)`
`long*`	in, inout, out	`real(c_long), dimension(*)`
`long*`	return	`real(c_long), pointer, dimension(:)`
`realtype*`	in, inout, out	`real(c_double), dimension(*)`
`realtype*`	return	`real(c_double), pointer, dimension(:)`
`sunindextype*`	in, inout, out	`real(c_long), dimension(*)`
`sunindextype*`	return	`real(c_long), pointer, dimension(:)`
`realtype[]`	in, inout, out	`real(c_double), dimension(*)`
`sunindextype[]`	in, inout, out	`integer(c_long), dimension(*)`
`N_Vector`	in, inout, out	`type(N_Vector)`
`N_Vector`	return	`type(N_Vector), pointer`
`SUNMatrix`	in, inout, out	`type(SUNMatrix)`
`SUNMatrix`	return	`type(SUNMatrix), pointer`
`SUNLinearSolver`	in, inout, out	`type(SUNLinearSolver)`
`SUNLinearSolver`	return	`type(SUNLinearSolver), pointer`
`SUNNonlinearSolver`	in, inout, out	`type(SUNNonlinearSolver)`
`SUNNonlinearSolver`	return	`type(SUNNonlinearSolver), pointer`
`FILE*`	in, inout, out, return	`type(c_ptr)`
`void*`	in, inout, out, return	`type(c_ptr)`
`T**`	in, inout, out, return	`type(c_ptr)`
`T***`	in, inout, out, return	`type(c_ptr)`
`T****`	in, inout, out, return	`type(c_ptr)`

7.1.3 Notable Fortran/C usage differences

While the Fortran 2003 interface to SUNDIALS closely follows the C API, some differences are inevitable due to the differences between Fortran and C. In this section, we note the most critical differences. Additionally, section *Data Types* discusses equivalencies of data types in the two languages.

7.1.3.1 Creating generic SUNDIALS objects

In the C API a generic SUNDIALS object, such as an `N_Vector`, is actually a pointer to an underlying C struct. However, in the Fortran 2003 interface, the derived type is bound to the C struct, not the pointer to the struct. E.g., `type(N_Vector)` is bound to the C struct `_generic_N_Vector` not the `N_Vector` type. The consequence of this is that creating and declaring SUNDIALS objects in Fortran is nuanced. This is illustrated in the code snippets below:

C code:

```
N_Vector x;
x = N_VNew_Serial(N);
```

Fortran code:

```
type(N_Vector), pointer :: x
x => FN_VNew_Serial(N)
```

Note that in the Fortran declaration, the vector is a `type(N_Vector), pointer`, and that the pointer assignment operator is then used.

7.1.3.2 Arrays and pointers

Unlike in the C API, in the Fortran 2003 interface, arrays and pointers are treated differently when they are return values versus arguments to a function. Additionally, pointers which are meant to be out parameters, not arrays, in the C API must still be declared as a rank-1 array in Fortran. The reason for this is partially due to the Fortran 2003 standard for C bindings, and partially due to the tool used to generate the interfaces. Regardless, the code snippets below illustrate the differences.

C code:

```
N_Vector x
realtype* xdata;
long int leniw, lenrw;

x = N_VNew_Serial(N);

/* capturing a returned array/pointer */
xdata = N_VGetArrayPointer(x)

/* passing array/pointer to a function */
N_VSetArrayPointer(xdata, x)

/* pointers that are out-parameters */
N_VSpace(x, &leniw, &lenrw);
```

Fortran code:

```
type(N_Vector), pointer :: x
real(c_double), pointer :: xdataptr(:)
real(c_double)          :: xdata(N)
integer(c_long)         :: leniw(1), lenrw(1)

x => FN_VNew_Serial(x)

! capturing a returned array/pointer
xdataptr => FN_VGetArrayPointer(x)
```

```
! passing array/pointer to a function
call FN_VSetArrayPointer(xdata, x)

! pointers that are out-parameters
call FN_VSpace(x, leniw, lenrw)
```

7.1.3.3 Passing procedure pointers and user data

Since functions/subroutines passed to SUNDIALS will be called from within C code, the Fortran procedure must have the attribute `bind(C)` Additionally, when providing them as arguments to a Fortran 2003 interface routine, it is required to convert a procedure's Fortran address to C with the Fortran intrinsic `c_funloc`.

Typically when passing user data to a SUNDIALS function, a user may simply cast some custom data structure as a `void*`. When using the Fortran 2003 interfaces, the same thing can be achieved. Note, the custom data structure *does not* have to be `bind(C)` since it is never accessed on the C side.

C code:

```
MyUserData* udata;
void *cvode_mem;

ierr = CVodeSetUserData(cvode_mem, udata);
```

Fortran code:

```
type(MyUserData) :: udata
type(c_ptr)      :: arkode_mem

ierr = FARKStepSetUserData(arkode_mem, c_loc(udata))
```

On the other hand, Fortran users may instead choose to store problem-specific data, e.g. problem parameters, within modules, and thus do not need the SUNDIALS-provided `user_data` pointers to pass such data back to user-supplied functions. These users should supply the `c_null_ptr` input for user_data arguments to the relevant SUNDIALS functions.

7.1.3.4 Passing NULL to optional parameters

In the SUNDIALS C API some functions have optional parameters that a caller can pass `NULL` to. If the optional parameter is of a type that is equivalent to a Fortran `type(c_ptr)` (see section *Data Types*), then a Fortran user can pass the intrinsic `c_null_ptr`. However, if the optional parameter is of a type that is not equivalent to `type(c_ptr)`, then a caller must provide a Fortran pointer that is dissociated. This is demonstrated in the code example below.

C code:

```
SUNLinearSolver LS;
N_Vector x, b;

! SUNLinSolSolve expects a SUNMatrix or NULL
! as the second parameter.
ierr = SUNLinSolSolve(LS, NULL, x, b);
```

Fortran code:

```
type(SUNLinearSolver), pointer :: LS
type(SUNMatrix), pointer :: A
type(N_Vector), pointer :: x, b

A => null()

! SUNLinSolSolve expects a type(SUNMatrix), pointer
! as the second parameter. Therefore, we cannot
! pass a c_null_ptr, rather we pass a disassociated A.
ierr = FSUNLinSolSolve(LS, A, x, b)
```

7.1.3.5 Working with `N_Vector` arrays

Arrays of `N_Vector` objects are interfaced to Fortran 2003 as opaque `type(c_ptr)`. As such, it is not possible to directly index an array of `N_Vector` objects returned by the `N_Vector` "VectorArray" operations, or packages with sensitivity capabilities. Instead, SUNDIALS provides a utility function `FN_VGetVecAtIndexVectorArray` that can be called for accessing a vector in a vector array. The example below demonstrates this:

C code:

```
N_Vector x;
N_Vector* vecs;

vecs = N_VCloneVectorArray(count, x);
for (int i=0; i < count; ++i)
N_VConst(vecs[i]);
```

Fortran code:

```
type(N_Vector), pointer :: x, xi
type(c_ptr)             :: vecs

vecs = FN_VCloneVectorArray(count, x)
do index, count
xi => FN_VGetVecAtIndexVectorArray(vecs, index)
call FN_VConst(xi)
enddo
```

SUNDIALS also provides the functions `FN_VSetVecAtIndexVectorArray` and `FN_VNewVectorArray` for working with `N_Vector` arrays. These functions are particularly useful for users of the Fortran interface to the `NVECTOR_MANYVECTOR` or `NVECTOR_MPIMANYVECTOR` when creating the subvector array. Both of these functions along with `FN_VGetVecAtIndexVectorArray` are further described in Chapter *NVECTOR Utility Functions*.

7.1.3.6 Providing file pointers

Expert SUNDIALS users may notice that there are a few advanced functions in the SUNDIALS C API which take a `FILE*` argument. Since there is no portable way to convert between a Fortran file descriptor and a C file pointer, SUNDIALS provides two utility functions for creating a `FILE*` and destroying it. These functions are defined in the module `fsundials_futils_mod`.

`function FSUNDIALSFileOpen` (*filename*, *mode*)
The function allocates a `FILE*` by calling the C function `fopen` with the provided filename and I/O mode.

The function argument `filename` is the full path to the file and has the type `character(kind=C_CHAR, len=*)`.

The function argument `mode` has the type `character(kind=C_CHAR, len=*)`. The string begins with one of the following characters:

- "r" - open text file for reading
- "r+" - open text file for reading and writing
- "w" - truncate text file to zero length or create it for writing
- "w+" - open text file for reading or writing, create it if it does not exist
- "a" - open for appending, see documentation of `fopen` for your system/compiler
- "a+ - open for reading and appending, see documentation for `fopen` for your system/compiler

The function returns a `type(C_PTR)` which holds a C `FILE*`.

subroutine FSUNDIALSFileClose(*fp*)

The function deallocates a C `FILE*` by calling the C function `fclose` with the provided pointer.

The function argument `fp` has the type `type(c_ptr)` and should be the C `FILE*` obtained from `fopen`.

7.1.4 Important notes on portability

The SUNDIALS Fortran 2003 interface *should* be compatible with any compiler supporting the Fortran 2003 ISO standard. However, it has only been tested and confirmed to be working with GNU Fortran 4.9+ and Intel Fortran 18.0.1+.

Upon compilation of SUNDIALS, Fortran module (`.mod`) files are generated for each Fortran 2003 interface. These files are highly compiler specific, and thus it is almost always necessary to compile a consuming application with the same compiler used to generate the modules.

7.2 FARKODE, an Interface Module for FORTRAN Applications

The FARKODE interface module is a package of C functions which support the use of the ARKStep time-stepping module for the solution of ODE systems

$$M\,\dot{y} = f^E(t,y) + f^I(t,y),$$

in a mixed Fortran/C setting. While ARKode is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in Fortran. We assume only minimal Fortran capabilities; specifically that the Fortran compiler support full Fortran77 functionality (although more modern standards are similarly supported). This package provides the necessary interfaces to ARKODE for the majority of supplied serial and parallel NVECTOR implementations.

7.2.1 Important note on portability

In this package, the names of the interface functions, and the names of the Fortran user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h`. The mapping defined by `F77_FUNC` in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

By "name-mangling", we mean that due to the case-independent nature of the Fortran language, Fortran compilers convert all subroutine and object names to use either all lower-case or all upper-case characters, and append either zero, one or two underscores as a prefix or suffix the the name. For example, the Fortran subroutine `MyFunction()` will be changed to one of `myfunction`, `MYFUNCTION`, `myfunction__`, `MYFUNCTION_`, and so on, depending on the Fortran compiler used.

SUNDIALS determines this name-mangling scheme at configuration time (see *ARKode Installation Procedure*).

7.2.2 Fortran Data Types

Throughout this documentation, we will refer to data types according to their usage in C. The equivalent types to these may vary, depending on your computer architecture and on how SUNDIALS was compiled (see *ARKode Installation Procedure*). A Fortran user should first determine the equivalent types for their architecture and compiler, and then take care that all arguments passed through this Fortran/C interface are declared of the appropriate type.

Integers: SUNDIALS uses `int`, `long int` and `sunindextype` types. As discussed in *ARKode Installation Procedure*, at compilation SUNDIALS allows the configuration of the 'index' type, that accepts values of 32-bit signed and 64-bit signed. This choice dictates the size of a SUNDIALS `sunindextype` variable.

- `int` – equivalent to an `INTEGER` or `INTEGER*4` in Fortran
- `long int` – this will depend on the computer architecture:
  - 32-bit architecture – equivalent to an `INTEGER` or `INTEGER*4` in Fortran
  - 64-bit architecture – equivalent to an `INTEGER*8` in Fortran
- `sunindextype` – this will depend on the SUNDIALS configuration:
  - 32-bit – equivalent to an `INTEGER` or `INTEGER*4` in Fortran
  - 64-bit – equivalent to an `INTEGER*8` in Fortran

Real numbers: As discussed in *ARKode Installation Procedure*, at compilation SUNDIALS allows the configuration option `--with-precision`, that accepts values of `single`, `double` or `extended` (the default is `double`). This choice dictates the size of a `realtype` variable. The corresponding Fortran types for these `realtype` sizes are:

- `single` – equivalent to a `REAL` or `REAL*4` in Fortran
- `double` – equivalent to a `DOUBLE PRECISION` or `REAL*8` in Fortran
- `extended` – equivalent to a `REAL*16` in Fortran

We note that when SUNDIALS is compiled with Fortran interfaces enabled, a file `sundials/sundials_fconfig.h` is placed in the installation's `include` directory, containing information about the Fortran types that correspond to the C types of the configured SUNDIALS installation. This file may be "included" by Fortran routines, as long as the compiler supports the Fortran90 standard (or higher), as shown in the ARKode example programs `ark_bruss.f90`, `ark_bruss1D_FEM_klu.f90` and `fark_heat2D.f90`.

Details on the Fortran interface to ARKode are provided in the following sub-sections:

7.2.2.1 FARKODE routines

In this section, we list the full set of user-callable functions comprising the FARKODE solver interface. For each function, we list the corresponding ARKStep functions, to provide a mapping between the two solver interfaces. Further documentation on each FARKODE function is provided in the following sections, *Usage of the FARKODE interface module*, *FARKODE optional output*, *Usage of the FARKROOT interface to rootfinding* and *Usage of the FARKODE interface to built-in preconditioners*. Additionally, all Fortran and C functions below are hyperlinked to their definitions in the documentation, for simplified access.

Interface to the NVECTOR modules

- `FNVINITS()` (defined by NVECTOR_SERIAL) interfaces to *N_VNewEmpty_Serial()*.
- `FNVINITP()` (defined by NVECTOR_PARALLEL) interfaces to *N_VNewEmpty_Parallel()*.
- `FNVINITOMP()` (defined by NVECTOR_OPENMP) interfaces to *N_VNewEmpty_OpenMP()*.
- `FNVINITPTS()` (defined by NVECTOR_PTHREADS) interfaces to *N_VNewEmpty_Pthreads()*.
- `FNVINITPH()` (defined by NVECTOR_PARHYP) interfaces to *N_VNewEmpty_ParHyp()*.

Interface to the SUNMATRIX modules

- `FSUNBANDMATINIT()` (defined by SUNMATRIX_BAND) interfaces to *SUNBandMatrix()*.
- `FSUNDENSEMATINIT()` (defined by SUNMATRIX_DENSE) interfaces to *SUNDenseMatrix()*.
- `FSUNSPARSEMATINIT()` (defined by SUNMATRIX_SPARSE) interfaces to *SUNSparseMatrix()*.

Interface to the SUNLINSOL modules

- `FSUNBANDLINSOLINIT()` (defined by SUNLINSOL_BAND) interfaces to *SUNLinSol_Band()*.
- `FSUNDENSELINSOLINIT()` (defined by SUNLINSOL_DENSE) interfaces to *SUNLinSol_Dense()*.
- `FSUNKLUINIT()` (defined by SUNLINSOL_KLU) interfaces to *SUNLinSol_KLU()*.
- `FSUNKLUREINIT()` (defined by SUNLINSOL_KLU) interfaces to `SUNLinSol_KLUReinit()`.
- `FSUNLAPACKBANDINIT()` (defined by SUNLINSOL_LAPACKBAND) interfaces to *SUNLinSol_LapackBand()*.
- `FSUNLAPACKDENSEINIT()` (defined by SUNLINSOL_LAPACKDENSE) interfaces to *SUNLinSol_LapackDense()*.
- `FSUNPCGINIT()` (defined by SUNLINSOL_PCG) interfaces to *SUNLinSol_PCG()*.
- `FSUNSPBCGSINIT()` (defined by SUNLINSOL_SPBCGS) interfaces to *SUNLinSol_SPBCGS()*.
- `FSUNSPFGMRINIT()` (defined by SUNLINSOL_SPFGMR) interfaces to *SUNLinSol_SPFGMR()*.
- `FSUNSPGMRINIT()` (defined by SUNLINSOL_SPGMR) interfaces to *SUNLinSol_SPGMR()*.
- `FSUNSPTFQMRINIT()` (defined by SUNLINSOL_SPTFQMR) interfaces to *SUNLinSol_SPTFQMR()*.
- `FSUNSUPERLUMTINIT()` (defined by SUNLINSOL_SUPERLUMT) interfaces to *SUNLinSol_SuperLUMT()*.

Interface to the SUNNONLINSOL modules

- `FSUNNEWTONINIT()` (defined by SUNNONLINSOL_NEWTON) interfaces to *SUNNonlinSol_Newton()*.
- `FSUNNEWTONSETMAXITERS()` (defined by SUNNONLINSOL_NEWTON) interfaces to *SUNNonlinSolSetMaxIters()* for a SUNNONLINSOL_NEWTON object.
- `FSUNFIXEDPOINTINIT()` (defined by SUNNONLINSOL_FIXEDPOINT) interfaces to *SUNNonlinSol_Newton()*.

- `FSUNFIXEDPOINTSETMAXITERS()` (defined by SUNNONLINSOL_FIXEDPOINT) interfaces to *SUNNonlinSolSetMaxIters()* for a SUNNONLINSOL_FIXEDPOINT object.

Interface to the main ARKODE module

- *FARKMALLOC()* interfaces to *ARKStepCreate()* and *ARKStepSetUserData()*, as well as one of *ARKStepSStolerances()* or *ARKStepSVtolerances()*.
- *FARKREINIT()* interfaces to *ARKStepReInit()*.
- *FARKRESIZE()* interfaces to *ARKStepResize()*.
- *FARKSETIIN()* and *FARKSETRIN()* interface to the ARKStepSet* and ARKStepSet* functions (see *Optional input functions*).
- *FARKEWTSET()* interfaces to *ARKStepWFtolerances()*.
- *FARKADAPTSET()* interfaces to *ARKStepSetAdaptivityFn()*.
- *FARKEXPSTABSET()* interfaces to *ARKStepSetStabilityFn()*.
- *FARKSETERKTABLE()* interfaces to *ARKStepSetTables()*.
- *FARKSETIRKTABLE()* interfaces to *ARKStepSetTables()*.
- *FARKSETARKTABLES()* interfaces to *ARKStepSetTables()*.
- *FARKSETRESTOLERANCE()* interfaces to either *ARKStepResStolerance()* and *ARKStepResVtolerance()*
- *FARKODE()* interfaces to *ARKStepEvolve()*, the ARKStepGet* functions (see *Optional output functions*), and to the optional output functions for the selected linear solver module (see *Optional output functions*).
- *FARKDKY()* interfaces to the interpolated output function *ARKStepGetDky()*.
- *FARKGETERRWEIGHTS()* interfaces to *ARKStepGetErrWeights()*.
- *FARKGETESTLOCALERR()* interfaces to *ARKStepGetEstLocalErrors()*.
- *FARKFREE()* interfaces to *ARKStepFree()*.

Interface to the system nonlinear solver interface

- *FARKNLSINIT()* interfaces to *ARKStepSetNonlinearSolver()*.

Interface to the system linear solver interfaces

- *FARKLSINIT()* interfaces to *ARKStepSetLinearSolver()*.
- *FARKDENSESETJAC()* interfaces to *ARKStepSetJacFn()*.
- *FARKBANDSETJAC()* interfaces to *ARKStepSetJacFn()*.
- *FARKSPARSESETJAC()* interfaces to *ARKStepSetJacFn()*.
- *FARKLSSETEPSLIN()* interfaces to *ARKStepSetEpsLin()*.
- *FARKLSSETJAC()* interfaces to *ARKStepSetJacTimes()*.
- *FARKLSSETPREC()* interfaces to *ARKStepSetPreconditioner()*.

Interface to the mass matrix linear solver interfaces

- *FARKLSMASSINIT()* interfaces to *ARKStepSetMassLinearSolver()*.
- *FARKDENSESETMASS()* interfaces to *ARKStepSetMassFn()*.
- *FARKBANDSETMASS()* interfaces to *ARKStepSetMassFn()*.
- *FARKSPARSESETMASS()* interfaces to *ARKStepSetMassFn()*.
- *FARKLSSETMASSEPSLIN()* interfaces to *ARKStepSetMassEpsLin()*.
- *FARKLSSETMASS()* interfaces to *ARKStepSetMassTimes()*.
- *FARKLSSETMASSPREC()* interfaces to *ARKStepSetMassPreconditioner()*.

User-supplied routines

As with the native C interface, the FARKODE solver interface requires user-supplied functions to specify the ODE problem to be solved. In contrast to the case of direct use of ARKStep, and of most Fortran ODE solvers, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. As a result, whether using a purely implicit, purely explicit, or mixed implicit-explicit solver, routines for both $f^E(t,y)$ and $f^I(t,y)$ must be provided by the user (though either of which may do nothing):

FARKODE routine (FORTRAN, user-supplied)	ARKStep interface function type
FARKIFUN()	*ARKRhsFn()*
FARKEFUN()	*ARKRhsFn()*

In addition, as with the native C interface a user may provide additional routines to assist in the solution process. Each of the following user-supplied routines is activated by calling the specified "activation" routine, with the exception of *FARKSPJAC()* which is required whenever a sparse matrix solver is used:

FARKODE routine (FORTRAN, user-supplied)	ARKStep interface function type	FARKODE "activation" routine
FARKDJAC()	*ARKLsJacFn()*	*FARKDENSESETJAC()*
FARKBJAC()	*ARKLsJacFn()*	*FARKBANDSETJAC()*
FARKSPJAC()	*ARKLsJacFn()*	*FARKSPARSESETJAC()*
FARKDMASS()	*ARKLsMassFn()*	*FARKDENSESETMASS()*
FARKBMASS()	*ARKLsMassFn()*	*FARKBANDSETMASS()*
FARKSPMASS()	*ARKLsMassFn()*	*FARKSPARSESETMASS()*
FARKPSET()	*ARKLsPrecSetupFn()*	*FARKLSSETPREC()*
FARKPSOL()	*ARKLsPrecSolveFn()*	*FARKLSSETPREC()*
FARKJTSETUP()	*ARKLsJacTimesSetupFn()*	*FARKLSSETJAC()*
FARKJTIMES()	*ARKLsJacTimesVecFn()*	*FARKLSSETJAC()*
FARKMASSPSET()	*ARKLsMassPrecSetupFn()*	*FARKLSSETMASSPREC()*
FARKMASSPSOL()	*ARKLsMassPrecSolveFn()*	*FARKLSSETMASSPREC()*
FARKMTSETUP()	*ARKLsMassTimesSetupFn()*	*FARKLSSETMASS()*
FARKMTIMES()	*ARKLsMassTimesVecFn()*	*FARKLSSETMASS()*
FARKEWT()	*ARKEwtFn()*	*FARKEWTSET()*
FARKADAPT()	*ARKAdaptFn()*	*FARKADAPTSET()*
FARKEXPSTAB()	*ARKExpStabFn()*	*FARKEXPSTABSET()*

7.2.2.2 Usage of the FARKODE interface module

The usage of FARKODE requires calls to a variety of interface functions, depending on the method options selected, and two or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized separately below. Some details are omitted, and the user is referred to the description of the corresponding C interface ARKStep functions for complete information on the arguments of any given user-callable interface routine, or of a given user-supplied function called by an interface function. The usage of FARKODE for rootfinding and with preconditioner modules is described in later subsections.

Right-hand side specification

The user must in all cases supply the following Fortran routines:

subroutine FARKIFUN(*T*, *Y*, *YDOT*, *IPAR*, *RPAR*, *IER*)
Sets the *YDOT* array to $f^I(t, y)$, the implicit portion of the right-hand side of the ODE system, as function of the independent variable $T = t$ and the array of dependent state variables $Y = y$.

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing state variables.
- *YDOT* (`realtype`, output) – array containing state derivatives.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 success, >0 recoverable error, <0 unrecoverable error).

subroutine FARKEFUN(*T*, *Y*, *YDOT*, *IPAR*, *RPAR*, *IER*)
Sets the *YDOT* array to $f^E(t, y)$, the explicit portion of the right-hand side of the ODE system, as function of the independent variable $T = t$ and the array of dependent state variables $Y = y$.

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing state variables.
- *YDOT* (`realtype`, output) – array containing state derivatives.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 success, >0 recoverable error, <0 unrecoverable error).

For purely explicit problems, although the routine *FARKIFUN()* must exist, it will never be called, and may remain empty. Similarly, for purely implicit problems, *FARKEFUN()* will never be called and must exist and may remain empty.

NVECTOR module initialization

If using one of the NVECTOR modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FNVINITS(4, NEQ, IER)
CALL FNVINITP(COMM, 4, NLOCAL, NGLOBAL, IER)
CALL FNVINITOMP(4, NEQ, NUM_THREADS, IER)
```

```
CALL FNVINITPTS(4, NEQ, NUM_THREADS, IER)
CALL FNVINITPH(COMM, 4, NLOCAL, NGLOBAL, IER)
```

in which the specific arguments are as described in the appropriate section of the Chapter *Vector Data Structures*.

SUNMATRIX module initialization

In the case of using either an implicit or ImEx method, the solution of each Runge-Kutta stage may involve the solution of linear systems related to the Jacobian $J = \frac{\partial f^I}{\partial y}$ of the implicit portion of the ODE system. If using a Newton iteration with direct SUNLINSOL linear solver module and one of the SUNMATRIX modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FSUNBANDMATINIT(4, N, MU, ML, SMU, IER)
CALL FSUNDENSEMATINIT(4, M, N, IER)
CALL FSUNSPARSEMATINIT(4, M, N, NNZ, SPARSETYPE, IER)
```

in which the specific arguments are as described in the appropriate section of the Chapter *Matrix Data Structures*. Note that these matrix options are usable only in a serial or multi-threaded environment.

As described in the section *Mass matrix solver (ARKStep only)*, in the case of using a problem with a non-identity mass matrix (no matter whether the integrator is implicit, explicit or ImEx), linear systems of the form $Mx = b$ must be solved, where M is the system mass matrix. If these are to be solved with a direct SUNLINSOL linear solver module and one of the SUNMATRIX modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FSUNBANDMASSMATINIT(N, MU, ML, SMU, IER)
CALL FSUNDENSEMASSMATINIT(M, N, IER)
CALL FSUNSPARSEMASSMATINIT(M, N, NNZ, SPARSETYPE, IER)
```

in which the specific arguments are as described in the appropriate section of the Chapter *Matrix Data Structures*, again noting that these are only usable in a serial or multi-threaded environment.

SUNLINSOL module initialization

If using a Newton iteration with one of the SUNLINSOL linear solver modules supplied with SUNDIALS, the user must make a call of the form

```
CALL FSUNBANDLINSOLINIT(4, IER)
CALL FSUNDENSELINSOLINIT(4, IER)
CALL FSUNKLUINIT(4, IER)
CALL FSUNLAPACKBANDINIT(4, IER)
CALL FSUNLAPACKDENSEINIT(4, IER)
CALL FSUNPCGINIT(4, PRETYPE, MAXL, IER)
CALL FSUNSPBCGSINIT(4, PRETYPE, MAXL, IER)
CALL FSUNSPFGMRINIT(4, PRETYPE, MAXL, IER)
CALL FSUNSPGMRINIT(4, PRETYPE, MAXL, IER)
CALL FSUNSPTFQMRINIT(4, PRETYPE, MAXL, IER)
CALL FSUNSUPERLUMTINIT(4, NUM_THREADS, IER)
```

in which the specific arguments are as described in the appropriate section of the Chapter *Description of the SUNLinearSolver module*. Note that the dense, band and sparse solvers are usable only in a serial or multi-threaded environment.

Once one of these has been initialized, its solver parameters may be modified using a call to the functions

```
CALL FSUNKLUSETORDERING(4, ORD_CHOICE, IER)
CALL FSUNSUPERLUMTSETORDERING(4, ORD_CHOICE, IER)
CALL FSUNPCGSETPRECTYPE(4, PRETYPE, IER)
CALL FSUNPCGSETMAXL(4, MAXL, IER)
CALL FSUNSPBCGSSETPRECTYPE(4, PRETYPE, IER)
CALL FSUNSPBCGSSETMAXL(4, MAXL, IER)
CALL FSUNSPFGMRSETGSTYPE(4, GSTYPE, IER)
CALL FSUNSPFGMRSETPRECTYPE(4, PRETYPE, IER)
CALL FSUNSPGMRSETGSTYPE(4, GSTYPE, IER)
CALL FSUNSPGMRSETPRECTYPE(4, PRETYPE, IER)
CALL FSUNSPTFQMRSETPRECTYPE(4, PRETYPE, IER)
CALL FSUNSPTFQMRSETMAXL(4, MAXL, IER)
```

where again the call sequences are described in the appropriate sections of the Chapter *Description of the SUNLinearSolver module*.

Similarly, in the case of using one of the SUNLINSOL linear solver modules supplied with SUNDIALS to solve a problem with a non-identity mass matrix, the user must make a call of the form

```
CALL FSUNMASSBANDLINSOLINIT(IER)
CALL FSUNMASSDENSELINSOLINIT(IER)
CALL FSUNMASSKLUINIT(IER)
CALL FSUNMASSLAPACKBANDINIT(IER)
CALL FSUNMASSLAPACKDENSEINIT(IER)
CALL FSUNMASSPCGINIT(PRETYPE, MAXL, IER)
CALL FSUNMASSSPBCGSINIT(PRETYPE, MAXL, IER)
CALL FSUNMASSSPFGMRINIT(PRETYPE, MAXL, IER)
CALL FSUNMASSSPGMRINIT(PRETYPE, MAXL, IER)
CALL FSUNMASSSPTFQMRINIT(PRETYPE, MAXL, IER)
CALL FSUNMASSSUPERLUMTINIT(NUM_THREADS, IER)
```

in which the specific arguments are as described in the appropriate section of the Chapter *Description of the SUNLinearSolver module*.

Once one of these has been initialized, its solver parameters may be modified using a call to the functions

```
CALL FSUNMASSKLUSETORDERING(ORD_CHOICE, IER)
CALL FSUNMASSSUPERLUMTSETORDERING(ORD_CHOICE, IER)
CALL FSUNMASSPCGSETPRECTYPE(PRETYPE, IER)
CALL FSUNMASSPCGSETMAXL(MAXL, IER)
CALL FSUNMASSSPBCGSSETPRECTYPE(PRETYPE, IER)
CALL FSUNMASSSPBCGSSETMAXL(MAXL, IER)
CALL FSUNMASSSPFGMRSETGSTYPE(GSTYPE, IER)
CALL FSUNMASSSPFGMRSETPRECTYPE(PRETYPE, IER)
CALL FSUNMASSSPGMRSETGSTYPE(GSTYPE, IER)
CALL FSUNMASSSPGMRSETPRECTYPE(PRETYPE, IER)
CALL FSUNMASSSPTFQMRSETPRECTYPE(PRETYPE, IER)
CALL FSUNMASSSPTFQMRSETMAXL(MAXL, IER)
```

where again the call sequences are described in the appropriate sections of the Chapter *Description of the SUNLinearSolver module*.

SUNNONLINSOL module initialization

If using a non-default nonlinear solver method, the user must make a call of the form

```
CALL FSUNNEWTONINIT(4, IER)
CALL FSUNFIXEDPOINTINIT(4, M, IER)
```

in which the specific arguments are as described in the appropriate section of the Chapter *Description of the SUNNonlinearSolver Module*.

Once one of these has been initialized, its solver parameters may be modified using a call to the functions

```
CALL FSUNNEWTONSETMAXITERS(4, MAXITERS, IER)
CALL FSUNFIXEDPOINTSETMAXITERS(4, MAXITERS, IER)
```

where again the call sequences are described in the appropriate sections of the Chapter *Description of the SUNNonlinearSolver Module*.

Problem specification

To set various problem and solution parameters and allocate internal memory, the user must call `FARKMALLOC()`.

subroutine FARKMALLOC(*T0*, *Y0*, *IMEX*, *IATOL*, *RTOL*, *ATOL*, *IOUT*, *ROUT*, *IPAR*, *RPAR*, *IER*)
Initializes the Fortran interface to the ARKStep solver, providing interfaces to the C routines `ARKStepCreate()` and `ARKStepSetUserData()`, as well as one of `ARKStepSStolerances()` or `ARKStepSVtolerances()`.

Arguments:

- *T0* (`realtype`, input) – initial value of t.
- *Y0* (`realtype`, input) – array of initial conditions.
- *IMEX* (`int`, input) – flag denoting basic integration method: 0 = implicit, 1 = explicit, 2 = ImEx.
- *IATOL* (`int`, input) – type for absolute tolerance input *ATOL*: 1 = scalar, 2 = array, 3 = user-supplied function; the user must subsequently call `FARKEWTSET()` and supply a routine `FARKEWT()` to compute the error weight vector.
- *RTOL* (`realtype`, input) – scalar relative tolerance.
- *ATOL* (`realtype`, input) – scalar or array absolute tolerance.
- *IOUT* (`long int`, input/output) – array of length 29 for integer optional outputs.
- *ROUT* (`realtype`, input/output) – array of length 6 for real optional outputs.
- *IPAR* (`long int`, input/output) – array of user integer data, which will be passed unmodified to all user-provided routines.
- *RPAR* (`realtype`, input/output) – array with user real data, which will be passed unmodified to all user-provided routines.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Notes: Modifications to the user data arrays *IPAR* and *RPAR* inside a user-provided routine will be propagated to all subsequent calls to such routines. The optional outputs associated with the main ARKStep integrator are listed in *Table: Optional FARKODE integer outputs* and *Table: Optional FARKODE real outputs*, in the section *FARKODE optional output*.

As an alternative to providing tolerances in the call to `FARKMALLOC()`, the user may provide a routine to compute the error weights used in the WRMS norm evaluations. If supplied, it must have the following form:

subroutine FARKEWT(*Y*, *EWT*, *IPAR*, *RPAR*, *IER*)
It must set the positive components of the error weight vector *EWT* for the calculation of the WRMS norm of *Y*.

Arguments:

- *Y* (`realtype`, input) – array containing state variables.
- *EWT* (`realtype`, output) – array containing the error weight vector.
- *IPAR* (`long int`, input) – array containing the integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing the real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

If the *FARKEWT()* routine is provided, then, following the call to *FARKMALLOC()*, the user must call the function *FARKEWTSET()*.

subroutine FARKEWTSET(*FLAG*, *IER*)
Informs FARKODE to use the user-supplied *FARKEWT()* function.

Arguments:

- *FLAG* (`int`, input) – flag, use "1" to denoting to use *FARKEWT()*.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Setting optional inputs

Unlike ARKStep's C interface, that provides separate functions for setting each optional input, FARKODE uses only three functions, that accept keywords to specify which optional input should be set to the provided value. These routines are *FARKSETIIN()*, *FARKSETRIN()*, and *FARKSETVIN()* and are further described below.

subroutine FARKSETIIN(*KEY*, *IVAL*, *IER*)
Specification routine to pass optional integer inputs to the *FARKODE()* solver.

Arguments:

- *KEY* (quoted string, input) – which optional input is set (see *Table: Keys for setting FARKODE integer optional inputs*).
- *IVAL* (`long int`, input) – the integer input value to be used.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Table: Keys for setting FARKODE integer optional inputs

Key	ARKStep routine
ORDER	*ARKStepSetOrder()*
DENSE_ORDER	*ARKStepSetDenseOrder()*
LINEAR	*ARKStepSetLinear()*
NONLINEAR	*ARKStepSetNonlinear()*
EXPLICIT	*ARKStepSetExplicit()*
IMPLICIT	*ARKStepSetImplicit()*
IMEX	*ARKStepSetImEx()*
IRK_TABLE_NUM	*ARKStepSetTableNum()*
ERK_TABLE_NUM	*ARKStepSetTableNum()*
ARK_TABLE_NUM *(a)*	*ARKStepSetTableNum()*
MAX_NSTEPS	*ARKStepSetMaxNumSteps()*
HNIL_WARNS	*ARKStepSetMaxHnilWarns()*
PREDICT_METHOD	*ARKStepSetPredictorMethod()*
MAX_ERRFAIL	*ARKStepSetMaxErrTestFails()*
MAX_CONVFAIL	*ARKStepSetMaxConvFails()*
MAX_NITERS	*ARKStepSetMaxNonlinIters()*
ADAPT_SMALL_NEF	*ARKStepSetSmallNumEFails()*
LSETUP_MSBP	*ARKStepSetLSetupFrequency()*
MAX_CONSTR_FAIL	*ARKStepSetMaxNumConstrFails()*

(a) When setting ARK_TABLE_NUM, pass in *IVAL* as an array of length 2, specifying the IRK table number first, then the ERK table number. The integer specifiers for each table may be found in the section *Appendix: ARKode Constants*, or in the ARKode header files arkode_butcher_dirk.h and arkode_butcher_erk.h.

subroutine FARKSETRIN(*KEY*, *RVAL*, *IER*)

Specification routine to pass optional real inputs to the *FARKODE()* solver.

Arguments:

- *KEY* (quoted string, input) – which optional input is set (see *Table: Keys for setting FARKODE real optional inputs*).
- *RVAL* (realtype, input) – the real input value to be used.
- *IER* (int, output) – return flag (0 success, $\neq 0$ failure).

Table: Keys for setting FARKODE real optional inputs

Key	ARKStep routine
INIT_STEP	*ARKStepSetInitStep()*
MAX_STEP	*ARKStepSetMaxStep()*
MIN_STEP	*ARKStepSetMinStep()*
STOP_TIME	*ARKStepSetStopTime()*
NLCONV_COEF	*ARKStepSetNonlinConvCoef()*
ADAPT_CFL	*ARKStepSetCFLFraction()*
ADAPT_SAFETY	*ARKStepSetSafetyFactor()*
ADAPT_BIAS	*ARKStepSetErrorBias()*
ADAPT_GROWTH	*ARKStepSetMaxGrowth()*
ADAPT_ETAMX1	*ARKStepSetMaxFirstGrowth()*
ADAPT_BOUNDS	*ARKStepSetFixedStepBounds()*
ADAPT_ETAMXF	*ARKStepSetMaxEFailGrowth()*
ADAPT_ETACF	*ARKStepSetMaxCFailGrowth()*
NONLIN_CRDOWN	*ARKStepSetNonlinCRDown()*
NONLIN_RDIV	*ARKStepSetNonlinRDiv()*
LSETUP_DGMAX	*ARKStepSetDeltaGammaMax()*
FIXED_STEP	*ARKStepSetFixedStep()*

subroutine FARKSETVIN(*KEY*, *VVAL*, *IER*)

Specification routine to pass optional vector inputs to the *FARKODE()* solver.

Arguments:

- *KEY* (quoted string, input) – which optional input is set (see *Table: Keys for setting FARKODE vector optional inputs*).
- *VVAL* (realtype*, input) – the input vector of real values to be used.
- *IER* (int, output) – return flag (0 success, $\neq$ 0 failure).

Table: Keys for setting FARKODE vector optional inputs

Key	ARKStep routine
CONSTR_VEC	*ARKStepSetConstraints()*

If a user wishes to reset all of the options to their default values, they may call the routine *FARKSETDEFAULTS()*.

subroutine FARKSETDEFAULTS(*IER*)

Specification routine to reset all FARKODE optional inputs to their default values.

Arguments:

- *IER* (int, output) – return flag (0 success, $\neq$ 0 failure).

Optional advanced FARKODE inputs

FARKODE supplies additional routines to specify optional advanced inputs to the *ARKStepEvolve()* solver. These are summarized below, and the user is referred to their C routine counterparts for more complete information.

subroutine FARKSETERKTABLE(*S*, *Q*, *P*, *C*, *A*, *B*, *BEMBED*, *IER*)
Interface to the routine *ARKStepSetTables()*.

Arguments:

- *S* (int, input) – number of stages in the table.
- *Q* (int, input) – global order of accuracy of the method.
- *P* (int, input) – global order of accuracy of the embedding.
- *C* (realtype, input) – array of length *S* containing the stage times.
- *A* (realtype, input) – array of length *S***S* containing the ERK coefficients (stored in row-major, "C", order).
- *B* (realtype, input) – array of length *S* containing the solution coefficients.
- *BEMBED* (realtype, input) – array of length *S* containing the embedding coefficients.
- *IER* (int, output) – return flag (0 success, $\neq 0$ failure).

subroutine FARKSETIRKTABLE(*S*, *Q*, *P*, *C*, *A*, *B*, *BEMBED*, *IER*)
Interface to the routine *ARKStepSetTables()*.

Arguments:

- *S* (int, input) – number of stages in the table.
- *Q* (int, input) – global order of accuracy of the method.
- *P* (int, input) – global order of accuracy of the embedding.
- *C* (realtype, input) – array of length *S* containing the stage times.
- *A* (realtype, input) – array of length *S***S* containing the IRK coefficients (stored in row-major, "C", order).
- *B* (realtype, input) – array of length *S* containing the solution coefficients.
- *BEMBED* (realtype, input) – array of length *S* containing the embedding coefficients.
- *IER* (int, output) – return flag (0 success, $\neq 0$ failure).

subroutine FARKSETARKTABLES(*S*, *Q*, *P*, *CI*, *CE*, *AI*, *AE*, *BI*, *BE*, *B2I*, *B2E*, *IER*)
Interface to the routine *ARKStepSetTables()*.

Arguments:

- *S* (int, input) – number of stages in the table.
- *Q* (int, input) – global order of accuracy of the method.
- *P* (int, input) – global order of accuracy of the embedding.
- *CI* (realtype, input) – array of length *S* containing the implicit stage times.
- *CE* (realtype, input) – array of length *S* containing the explicit stage times.
- *AI* (realtype, input) – array of length *S***S* containing the IRK coefficients (stored in row-major, "C", order).
- *AE* (realtype, input) – array of length *S***S* containing the ERK coefficients (stored in row-major, "C", order).
- *BI* (realtype, input) – array of length *S* containing the implicit solution coefficients.
- *BE* (realtype, input) – array of length *S* containing the explicit solution coefficients.

- *B2I* (`realtype`, input) – array of length *S* containing the implicit embedding coefficients.
- *B2E* (`realtype`, input) – array of length *S* containing the explicit embedding coefficients.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

subroutine FARKSETRESTOLERANCE (*IATOL*, *ATOL*, *IER*)
Interface to the routines *ARKStepResStolerance()* and *ARKStepResVtolerance()*.

Arguments:

- *IATOL* (`int`, input) – type for absolute residual tolerance input *ATOL*: 1 = scalar, 2 = array.
- *ATOL* (`realtype`, input) – scalar or array absolute residual tolerance.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Additionally, a user may set the accuracy-based step size adaptivity strategy (and it's associated parameters) through a call to *FARKSETADAPTIVITYMETHOD()*, as described below.

subroutine FARKSETADAPTIVITYMETHOD (*IMETHOD*, *IDEFAULT*, *IPQ*, *PARAMS*, *IER*)
Specification routine to set the step size adaptivity strategy and parameters within the *FARKODE()* solver. Interfaces with the C routine *ARKStepSetAdaptivityMethod()*.

Arguments:

- *IMETHOD* (`int`, input) – choice of adaptivity method.
- *IDEFAULT* (`int`, input) – flag denoting whether to use default parameters (1) or that customized parameters will be supplied (1).
- *IPQ* (`int`, input) – flag denoting whether to use the embedding order of accuracy (0) or the method order of accuracy (1) within step adaptivity algorithm.
- *PARAMS* (`realtype`, input) – array of 3 parameters to be used within the adaptivity strategy.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Lastly, the user may provide functions to aid/replace those within ARKStep for handling adaptive error control and explicit stability. The former of these is designed for advanced users who wish to investigate custom step adaptivity approaches as opposed to using any of those built-in to ARKStep. In ARKStep's C/C++ interface, this would be provided by a function of type *ARKAdaptFn()*; in the Fortran interface this is provided through the user-supplied function:

subroutine FARKADAPT (*Y*, *T*, *H1*, *H2*, *H3*, *E1*, *E2*, *E3*, *Q*, *P*, *HNEW*, *IPAR*, *RPAR*, *IER*)
It must set the new step size *HNEW* based on the three previous steps (*H1*, *H2*, *H3*) and the three previous error estimates (*E1*, *E2*, *E3*).

Arguments:

- *Y* (`realtype`, input) – array containing state variables.
- *T* (`realtype`, input) – current value of the independent variable.
- *H1* (`realtype`, input) – current step size.
- *H2* (`realtype`, input) – previous step size.
- *H3* (`realtype`, input) – previous-previous step size.
- *E1* (`realtype`, input) – estimated temporal error in current step.
- *E2* (`realtype`, input) – estimated temporal error in previous step.
- *E3* (`realtype`, input) – estimated temporal error in previous-previous step.
- *Q* (`int`, input) – global order of accuracy for RK method.

- *P* (`int`, input) – global order of accuracy for RK embedded method.
- *HNEW* (`realtype`, output) – array containing the error weight vector.
- *IPAR* (`long int`, input) – array containing the integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing the real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

This routine is enabled by a call to the activation routine:

subroutine FARKADAPTSET(*FLAG*, *IER*)
Informs FARKODE to use the user-supplied *FARKADAPT()* function.

Arguments:

- *FLAG* (`int`, input) – flag, use "1" to denoting to use *FARKADAPT()*, or use "0" to denote a return to the default adaptivity strategy.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Note: The call to *FARKADAPTSET()* must occur *after* the call to *FARKMALLOC()*.

Similarly, if either an explicit or mixed implicit-explicit integration method is to be employed, the user may specify a function to provide the maximum explicitly-stable step for their problem. Again, in the C/C++ interface this would be a function of type *ARKExpStabFn()*, while in ARKStep's Fortran interface this must be given through the user-supplied function:

subroutine FARKEXPSTAB(*Y*, *T*, *HSTAB*, *IPAR*, *RPAR*, *IER*)
It must set the maximum explicitly-stable step size, *HSTAB*, based on the current solution, *Y*.

Arguments:

- *Y* (`realtype`, input) – array containing state variables.
- *T* (`realtype`, input) – current value of the independent variable.
- *HSTAB* (`realtype`, output) – maximum explicitly-stable step size.
- *IPAR* (`long int`, input) – array containing the integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing the real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

This routine is enabled by a call to the activation routine:

subroutine FARKEXPSTABSET(*FLAG*, *IER*)
Informs FARKODE to use the user-supplied *FARKEXPSTAB()* function.

Arguments:

- *FLAG* (`int`, input) – flag, use "1" to denoting to use *FARKEXPSTAB()*, or use "0" to denote a return to the default error-based stability strategy.
- *IER* (`int`, output) – return flag (0 success, $\neq 0$ failure).

Note: The call to *FARKEXPSTABSET()* must occur *after* the call to *FARKMALLOC()*.

Nonlinear solver module specification

To use a non-default nonlinear solver algorithm, then after it has been initialized in step *SUNNONLINSOL module initialization* above, the user of FARKODE must attach it to ARKSTEP by calling the `FARKNLSINIT()` routine:

subroutine FARKNLSINIT(*IER*)
Interfaces with the `ARKStepSetNonlinearSolver()` function to specify use of a non-default nonlinear solver module.

Arguments:

- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

System linear solver interface specification

To attach the linear solver (and optionally the matrix) object(s) initialized in steps *SUNMATRIX module initialization* and *SUNLINSOL module initialization* above, the user of FARKODE must initialize the linear solver interface. To attach any SUNLINSOL object (and optional SUNMATRIX object) to ARKStep, following calls to initialize the SUNLINSOL (and SUNMATRIX) object(s) in steps *SUNMATRIX module initialization* and *SUNLINSOL module initialization* above, the user must call the `FARKLSINIT()` routine:

subroutine FARKLSINIT(*IER*)
Interfaces with the `ARKStepSetLinearSolver()` function to attach a linear solver object (and optionally a matrix object) to ARKStep.

Arguments:

- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

Matrix-based linear solvers

As an option when using ARKSTEP with either the SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE linear solver modules, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \frac{\partial f^I}{\partial y}$. If supplied, it must have the following form:

subroutine FARKDJAC(*NEQ*, *T*, *Y*, *FY*, *DJAC*, *H*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)
Interface to provide a user-supplied dense Jacobian approximation function (of type `ARKLsJacFn()`), to be used by the SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE solver modules.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing values of the dependent state variables.
- *FY* (`realtype`, input) – array containing values of the dependent state derivatives.
- *DJAC* (`realtype` of size (NEQ,NEQ), output) – 2D array containing the Jacobian entries.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.

- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: Typically this routine will use only *NEQ*, *T*, *Y*, and *DJAC*. It must compute the Jacobian and store it column-wise in *DJAC*.

If the above routine uses difference quotient approximations, it may need to access the error weight array *EWT* in the calculation of suitable increments. The array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using one of the work arrays as temporary storage for *EWT*. It may also need the unit roundoff, which can be obtained as the optional output *ROUT(6)*, passed from the calling program to this routine using either *RPAR* or a common block.

If the `FARKDJAC()` routine is provided, then, following the call to `FARKLSINIT()`, the user must call the routine `FARKDENSESETJAC()`:

subroutine FARKDENSESETJAC(*FLAG*, *IER*)

Interface to the `ARKStepSetJacFn()` function, specifying to use the user-supplied routine `FARKDJAC()` for the Jacobian approximation.

Arguments:

- *FLAG* (`int`, input) – any nonzero value specifies to use `FARKDJAC()`.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

As an option when using ARKStep with either the SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND linear solver modules, the user may supply a routine that computes a banded approximation of the linear system Jacobian $J = \frac{\partial f^I}{\partial y}$. If supplied, it must have the following form:

subroutine FARKBJAC(*NEQ*, *MU*, *ML*, *MDIM*, *T*, *Y*, *FY*, *BJAC*, *H*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied band Jacobian approximation function (of type `ARKLsJacFn()`), to be used by the SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND solver modules.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *MU* (`long int`, input) – upper half-bandwidth.
- *ML* (`long int`, input) – lower half-bandwidth.
- *MDIM* (`long int`, input) – leading dimension of *BJAC* array.
- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing dependent state variables.
- *FY* (`realtype`, input) – array containing dependent state derivatives.
- *BJAC* (`realtype` of size *(MDIM,NEQ)*, output) – 2D array containing the Jacobian entries.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: Typically this routine will use only *NEQ*, *MU*, *ML*, *T*, *Y*, and *BJAC*. It must load the *MDIM* by *N* array *BJAC* with the Jacobian matrix at the current (t, y) in band form. Store in *BJAC(k,j)* the Jacobian element $J_{i,j}$ with *k = i - j + MU + 1* (or *k = 1, ..., ML+MU+1*) and *j = 1, ..., N*.

If the above routine uses difference quotient approximations, it may need to use the error weight array *EWT* in the calculation of suitable increments. The array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using one of the work arrays as temporary storage for *EWT*. It may also need the unit roundoff, which can be obtained as the optional output *ROUT(6)*, passed from the calling program to this routine using either *RPAR* or a common block.

If the `FARKBJAC()` routine is provided, then, following the call to `FARKLSINIT()`, the user must call the routine `FARKBANDSETJAC()`.

subroutine FARKBANDSETJAC(*FLAG*, *IER*)

Interface to the `ARKStepSetJacFn()` function, specifying to use the user-supplied routine `FARKBJAC()` for the Jacobian approximation.

Arguments:

- *FLAG* (`int`, input) – any nonzero value specifies to use `FARKBJAC()`.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

When using ARKStep with either the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT sparse direct linear solver modules, the user must supply a routine that computes a sparse approximation of the system Jacobian $J = \frac{\partial f^I}{\partial y}$. Both the KLU and SuperLU_MT solvers allow specification of J in either compressed-sparse-column (CSC) format or compressed-sparse-row (CSR) format. The sparse Jacobian approximation function must have the following form:

subroutine FARKSPJAC(*T*, *Y*, *FY*, *N*, *NNZ*, *JDATA*, *JINDEXVALS*, *JINDEXPTRS*, *H*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied sparse Jacobian approximation function (of type `ARKLsJacFn()`), to be used by the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT solver modules.

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing values of the dependent state variables.
- *FY* (`realtype`, input) – array containing values of the dependent state derivatives.
- *N* (`sunindextype`, input) – number of matrix rows and columns in Jacobian.
- *NNZ* (`sunindextype`, input) – allocated length of nonzero storage in Jacobian.
- *JDATA* (`realtype` of size NNZ, output) – nonzero values in Jacobian.
- *JINDEXVALS* (`sunindextype` of size NNZ, output) – row *[CSR: column]* indices for each nonzero Jacobian entry.
- *JINDEXPTRS* (`sunindextype` of size N+1, output) – indices of where each column's *[CSR: row's]* nonzeros begin in data array; last entry points just past end of data values.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: due to the internal storage format of the SUNMATRIX_SPARSE module, the matrix-specific integer parameters and arrays are all of type `sunindextype` – the index precision (32-bit vs 64-bit signed integers) specified during the SUNDIALS build. It is assumed that the user's Fortran codes are constructed to have matching type to how SUNDIALS was installed.

If the above routine uses difference quotient approximations to compute the nonzero entries, it may need to access the error weight array *EWT* in the calculation of suitable increments. The array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using one of the work arrays as temporary storage for *EWT*. It may also need the unit roundoff, which can be obtained as the optional output *ROUT(6)*, passed from the calling program to this routine using either *RPAR* or a common block.

When supplying the `FARKSPJAC()` routine, following the call to `FARKLSINIT()`, the user must call the routine `FARKSPARSESETJAC()`.

subroutine FARKSPARSESETJAC(*IER*)
Interface to the `ARKStepSetJacFn()` function, specifying that the user-supplied routine `FARKSPJAC()` has been provided for the Jacobian approximation.

Arguments:

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

Iterative linear solvers

As described in the section *Linear iteration error control*, a user may adjust the linear solver tolerance scaling factor ϵ_L. Fortran users may adjust this value by calling the function `FARKLSSETEPSLIN()`:

subroutine FARKLSSETEPSLIN(*EPLIFAC*, *IER*)
Interface to the function `ARKStepSetEpsLin()` to specify the linear solver tolerance scale factor ϵ_L for the Newton system linear solver.

This routine must be called *after* `FARKLSINIT()`.

Arguments:

- *EPLIFAC* (`realtype`, input) – value to use for ϵ_L. Passing a value of 0 indicates to use the default value (0.05).
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Optional user-supplied routines `FARKJTSETUP()` and `FARKJTIMES()` may be provided to compute the product of the system Jacobian $J = \frac{\partial f^I}{\partial y}$ and a given vector v. If these are supplied, then following the call to `FARKLSINIT()`, the user must call the `FARKLSSETJAC()` routine with *FLAG* $\neq 0$:

subroutine FARKLSSETJAC(*FLAG*, *IER*)
Interface to the function `ARKStepSetJacTimes()` to specify use of the user-supplied Jacobian-times-vector setup and product functions, `FARKJTSETUP()` and `FARKJTIMES()`, respectively.

This routine must be called *after* `FARKLSINIT()`.

Arguments:

- *FLAG* (`int`, input) – flag denoting use of user-supplied Jacobian-times-vector routines. A nonzero value specifies to use these the user-supplied routines, a zero value specifies not to use these.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Similarly, optional user-supplied routines `FARKPSET()` and `FARKPSOL()` may be provided to perform preconditioning of the iterative linear solver (note: the SUNLINSOL module must have been configured with preconditioning enabled). If these routines are supplied, then following the call to `FARKLSINIT()` the user must call the routine `FARKLSSETPREC()` with *FLAG* $\neq 0$:

subroutine FARKLSSETPREC(*FLAG*, *IER*)
Interface to the function `ARKStepSetPreconditioner()` to specify use of the user-supplied preconditioner setup and solve functions, `FARKPSET()` and `FARKPSOL()`, respectively.

This routine must be called *after* `FARKLSINIT()`.

Arguments:

- *FLAG* (`int`, input) – flag denoting use of user-supplied preconditioning routines. A nonzero value specifies to use these the user-supplied routines, a zero value specifies not to use these.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

With treatment of the linear systems by any of the Krylov iterative solvers, there are four optional user-supplied routines – *FARKJTSETUP()*, *FARKJTIMES()*, *FARKPSET()* and *FARKPSOL()*. The specifications of these functions are given below.

As an option when using iterative linear solvers, the user may supply a routine that computes the product of the system Jacobian $J = \frac{\partial f^I}{\partial y}$ and a given vector v. If supplied, it must have the following form:

subroutine FARKJTIMES(*V*, *FJV*, *T*, *Y*, *FY*, *H*, *IPAR*, *RPAR*, *WORK*, *IER*)

Interface to provide a user-supplied Jacobian-times-vector product approximation function (corresponding to a C interface routine of type *ARKLsJacTimesVecFn()*), to be used by one of the Krylov iterative linear solvers.

Arguments:

- *V* (`realtype`, input) – array containing the vector to multiply.
- *FJV* (`realtype`, output) – array containing resulting product vector.
- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing dependent state variables.
- *FY* (`realtype`, input) – array containing dependent state derivatives.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *WORK* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: Typically this routine will use only *T*, *Y*, *V*, and *FJV*. It must compute the product vector Jv, where v is given in *V*, and the product is stored in *FJV*.

If the user's Jacobian-times-vector product routine requires that any Jacobian related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of this data:

subroutine FARKJTSETUP(*T*, *Y*, *FY*, *H*, *IPAR*, *RPAR*, *IER*)

Interface to setup data for use in a user-supplied Jacobian-times-vector product approximation function (corresponding to a C interface routine of type `ARKLJacTimesSetupFn()`).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – array containing dependent state variables.
- *FY* (`realtype`, input) – array containing dependent state derivatives.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: Typically this routine will use only *T* and *Y*, and store the results in either the arrays *IPAR* and *RPAR*, or in a Fortran module or common block.

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

subroutine FARKPSOL(*T*, *Y*, *FY*, *R*, *Z*, *GAMMA*, *DELTA*, *LR*, *IPAR*, *RPAR*, *VT*, *IER*)
User-supplied preconditioner solve routine (of type *ARKLsPrecSolveFn()*).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – current dependent state variable array.
- *FY* (`realtype`, input) – current dependent state variable derivative array.
- *R* (`realtype`, input) – right-hand side array.
- *Z* (`realtype`, output) – solution array.
- *GAMMA* (`realtype`, input) – Jacobian scaling factor.
- *DELTA* (`realtype`, input) – desired residual tolerance.
- *LR* (`int`, input) – flag denoting to solve the right or left preconditioner system: 1 = left preconditioner, 2 = right preconditioner.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable failure, <0 if a non-recoverable failure).

Notes: Typically this routine will use only *T*, *Y*, *GAMMA*, *R*, *LR*, and *Z*. It must solve the preconditioner linear system $Pz = r$. The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the matrix $M - \gamma J$, where M is the system mass matrix, γ is the input GAMMA, and $J = \frac{\partial f^I}{\partial y}$.

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of the preconditioner:

subroutine FARKPSET(*T*, *Y*, *FY*, *JOK*, *JCUR*, *GAMMA*, *H*, *IPAR*, *RPAR*, *IER*)
User-supplied preconditioner setup routine (of type *ARKLsPrecSetupFn()*).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *Y* (`realtype`, input) – current dependent state variable array.
- *FY* (`realtype`, input) – current dependent state variable derivative array.
- *JOK* (`int`, input) – flag indicating whether Jacobian-related data needs to be recomputed: 0 = recompute, 1 = reuse with the current value of *GAMMA*.
- *JCUR* (`realtype`, output) – return flag to denote if Jacobian data was recomputed (1=yes, 0=no).
- *GAMMA* (`realtype`, input) – Jacobian scaling factor.
- *H* (`realtype`, input) – current step size.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to *FARKMALLOC()*.

- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable failure, <0 if a non-recoverable failure).

Notes: This routine must set up the preconditioner P to be used in the subsequent call to `FARKPSOL()`. The preconditioner (or the product of the left and right preconditioners if using both) should be an approximation to the matrix $M - \gamma J$, where M is the system mass matrix, γ is the input *GAMMA*, and $J = \frac{\partial f^I}{\partial y}$.

Notes:

1. If the user's `FARKJTSETUP()`, `FARKJTIMES()` or `FARKPSET()` routines use difference quotient approximations, they may need to use the error weight array *EWT* and/or the unit roundoff, in the calculation of suitable increments. Also, if `FARKPSOL()` uses an iterative method in its solution, the residual vector $\rho = r - Pz$ of the system should be made less than $\delta =$ *DELTA* in the weighted l2 norm, i.e.

$$\left(\sum_i \left(\rho_i \, EWT_i\right)^2\right)^{1/2} < \delta.$$

2. If needed in `FARKJTSETUP() FARKJTIMES()`, `FARKPSOL()`, or `FARKPSET()`, the error weight array *EWT* can be obtained by calling `FARKGETERRWEIGHTS()` using a user-allocated array as temporary storage for *EWT*.
3. If needed in `FARKJTSETUP() FARKJTIMES()`, `FARKPSOL()`, or `FARKPSET()`, the unit roundoff can be obtained as the optional output *ROUT(6)* (available after the call to `FARKMALLOC()`) and can be passed using either the *RPAR* user data array or a common block.

Mass matrix linear solver interface specification

To attach the mass matrix linear solver (and optionally the mass matrix) object(s) initialized in steps *SUNMATRIX module initialization* and *SUNLINSOL module initialization* above, the user of FARKODE must initialize the mass-matrix linear solver interface. To attach any SUNLINSOL object (and optional SUNMATRIX object) to the mass-matrix solver interface, following calls to initialize the SUNLINSOL (and SUNMATRIX) object(s) in steps *SUNMATRIX module initialization* and *SUNLINSOL module initialization* above, the user must call the `FARKLSMASSINIT()` routine:

subroutine FARKLSMASSINIT(*TIME_DEP*, *IER*)

Interfaces with the `ARKStepSetMassLinearSolver()` function to attach a linear solver object (and optionally a matrix object) to ARKStep's mass-matrix linear solver interface.

Arguments:

- *TIME_DEP* (`int`, input) – flag indicating whether the mass matrix is time-dependent (1) or not (0). *Currently, only values of "0" are supported*
- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

Matrix-based mass matrix linear solvers

When using the mass-matrix linear solver interface with the SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE mass matrix linear solver modules, the user must supply a routine that computes the dense mass matrix M. This routine must have the following form:

subroutine FARKDMASS(*NEQ*, *T*, *DMASS*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied dense mass matrix computation function (of type *ARKLsMassFn()*), to be used by the SUNLINSOL_DENSE or SUNLINSOL_LAPACKDENSE solver modules.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *T* (`realtype`, input) – current value of the independent variable.
- *DMASS* (`realtype` of size (NEQ,NEQ), output) – 2D array containing the mass matrix entries.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: Typically this routine will use only *NEQ*, *T*, and *DMASS*. It must compute the mass matrix and store it column-wise in *DMASS*.

To indicate that the *FARKDMASS()* routine has been provided, then, following the call to *FARKLSMASSINIT()*, the user must call the routine *FARKDENSESETMASS()*:

subroutine FARKDENSESETMASS(*IER*)

Interface to the *ARKStepSetMassFn()* function, specifying to use the user-supplied routine *FARKDMASS()* for the mass matrix calculation.

Arguments:

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

When using the mass-matrix linear solver interface with the SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND mass matrix linear solver modules, the user must supply a routine that computes the banded mass matrix M. This routine must have the following form:

subroutine FARKBMASS(*NEQ*, *MU*, *ML*, *MDIM*, *T*, *BMASS*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied band mass matrix calculation function (of type *ARKLsMassFn()*), to be used by the SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND solver modules.

Arguments:

- *NEQ* (`long int`, input) – size of the ODE system.
- *MU* (`long int`, input) – upper half-bandwidth.
- *ML* (`long int`, input) – lower half-bandwidth.
- *MDIM* (`long int`, input) – leading dimension of *BMASS* array.
- *T* (`realtype`, input) – current value of the independent variable.
- *BMASS* (`realtype` of size *(MDIM,NEQ)*, output) – 2D array containing the mass matrix entries.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *WK1*, *WK2*, *WK3* (`realtype`, input) – array containing temporary workspace of same size as *Y*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: Typically this routine will use only *NEQ*, *MU*, *ML*, *T*, and *BMASS*. It must load the *MDIM* by *N* array *BMASS* with the mass matrix at the current (t) in band form. Store in *BMASS(k,j)* the mass matrix element $M_{i,j}$ with *k = i - j + MU + 1* (or *k = 1, ..., ML+MU+1*) and *j = 1, ..., N*.

To indicate that the *FARKBMASS()* routine has been provided, then, following the call to *FARKLSMASSINIT()*, the user must call the routine *FARKBANDSETMASS()*:

subroutine FARKBANDSETMASS(*IER*)

Interface to the *ARKStepSetMassFn()* function, specifying to use the user-supplied routine *FARKBMASS()* for the mass matrix calculation.

Arguments:

- *IER* (int, output) – return flag (0 if success, $\neq 0$ if an error occurred).

When using the mass-matrix linear solver interface with the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT mass matrix linear solver modules, the user must supply a routine that computes the sparse mass matrix M. Both the KLU and SuperLU_MT solver interfaces support the compressed-sparse-column (CSC) and compressed-sparse-row (CSR) matrix formats. The desired format must have been specified to the FSUNSPARSEMASSMATINIT() function when initializing the sparse mass matrix. The user-provided routine to compute M must have the following form:

subroutine FARKSPMASS(*T*, *N*, *NNZ*, *MDATA*, *MINDEXVALS*, *MINDEXPTRS*, *IPAR*, *RPAR*, *WK1*, *WK2*, *WK3*, *IER*)

Interface to provide a user-supplied sparse mass matrix approximation function (of type *ARKLsMassFn()*), to be used by the SUNLINSOL_KLU or SUNLINSOL_SUPERLUMT solver modules.

Arguments:

- *T* (realtype, input) – current value of the independent variable.
- *N* (sunindextype, input) – number of mass matrix rows and columns.
- *NNZ* (sunindextype, input) – allocated length of nonzero storage in mass matrix.
- *MDATA* (realtype of size NNZ, output) – nonzero values in mass matrix.
- *MINDEXVALS* (sunindextype of size NNZ, output) – row *[CSR: column]* indices for each nonzero mass matrix entry.
- *MINDEXPTRS* (sunindextype of size N+1, output) – indices of where each column's *[CSR: row's]* nonzeros begin in data array; last entry points just past end of data values.
- *IPAR* (long int, input) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (realtype, input) – array containing real user data that was passed to *FARKMALLOC()*.
- *WK1*, *WK2*, *WK3* (realtype, input) – array containing temporary workspace of same size as *Y*.
- *IER* (int, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: due to the internal storage format of the SUNMATRIX_SPARSE module, the matrix-specific integer parameters and arrays are all of type sunindextype – the index precision (32-bit vs 64-bit signed integers) specified during the SUNDIALS build. It is assumed that the user's Fortran codes are constructed to have matching type to how SUNDIALS was installed.

To indicate that the *FARKSPMASS()* routine has been provided, then, following the call to *FARKLSMASSINIT()*, the user must call the routine *FARKSPARSESETMASS()*:

subroutine FARKSPARSESETMASS(*IER*)

Interface to the *ARKStepSetMassFn()* function, specifying that the user-supplied routine *FARKSPMASS()* has been provided for the mass matrix calculation.

Arguments:

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error occurred).

Iterative mass matrix linear solvers

As described in the section *Linear iteration error control*, a user may adjust the linear solver tolerance scaling factor ϵ_L. Fortran users may adjust this value for the mass matrix linear solver by calling the function `FARKLSSETMASSEPSLIN()`:

subroutine FARKLSSETMASSEPSLIN(*EPLIFAC*, *IER*)

Interface to the function `ARKStepSetMassEpsLin()` to specify the linear solver tolerance scale factor ϵ_L for the mass matrix linear solver.

This routine must be called *after* `FARKLSMASSINIT()`.

Arguments:

- *EPLIFAC* (`realtype`, input) – value to use for ϵ_L. Passing a value of 0 indicates to use the default value (0.05).
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

With treatment of the mass matrix linear systems by any of the Krylov iterative solvers, there are two required user-supplied routines, `FARKMTSETUP()` and `FARKMTIMES()`, and there are two optional user-supplied routines, `FARKMASSPSET()` and `FARKMASSPSOL()`. The specifications of these functions are given below.

The required routines when using a Krylov iterative mass matrix linear solver perform setup and computation of the product of the system mass matrix M and a given vector v. The product routine must have the following form:

subroutine FARKMTIMES(*V*, *MV*, *T*, *IPAR*, *RPAR*, *IER*)

Interface to a user-supplied mass-matrix-times-vector product approximation function (corresponding to a C interface routine of type `ARKLsMassTimesVecFn()`), to be used by one of the Krylov iterative linear solvers.

Arguments:

- *V* (`realtype`, input) – array containing the vector to multiply.
- *MV* (`realtype`, output) – array containing resulting product vector.
- *T* (`realtype`, input) – current value of the independent variable.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: Typically this routine will use only *T*, *V*, and *MV*. It must compute the product vector Mv, where v is given in *V*, and the product is stored in *MV*.

If the user's mass-matrix-times-vector product routine requires that any mass matrix data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of this data:

subroutine FARKMTSETUP(*T*, *IPAR*, *RPAR*, *IER*)

Interface to a user-supplied mass-matrix-times-vector setup function (corresponding to a C interface routine of type `ARKLsMassTimesSetupFn()`).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *IPAR* (`long int`, input) – array containing integer user data that was passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input) – array containing real user data that was passed to `FARKMALLOC()`.

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: Typically this routine will use only *T*, and store the results in either the arrays *IPAR* and *RPAR*, or in a Fortran module or common block. If no mass matrix setup is needed, this routine should just set *IER* to 0 and return.

To indicate that these routines have been supplied by the user, then, following the call to *FARKLSMASSINIT()*, the user must call the routine *FARKLSSETMASS()*:

subroutine FARKLSSETMASS(*IER*)

Interface to the function *ARKStepSetMassTimes()* to specify use of the user-supplied mass-matrix-times-vector setup and product functions *FARKMTSETUP()* and *FARKMTIMES()*.

This routine must be called *after* *FARKLSMASSINIT()*.

Arguments:

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Two optional user-supplied preconditioning routines may be supplied to help accelerate convergence of the Krylov mass matrix linear solver. If preconditioning was selected when enabling the Krylov solver (i.e. the solver was set up with *IPRETYPE* $\neq 0$), then the user must also call the routine *FARKLSSETMASSPREC()* with *FLAG* $\neq 0$:

subroutine FARKLSSETMASSPREC(*FLAG*, *IER*)

Interface to the function *ARKStepSetMassPreconditioner()* to specify use of the user-supplied preconditioner setup and solve functions, *FARKMASSPSET()* and *FARKMASSPSOL()*, respectively.

This routine must be called *after* *FARKLSMASSINIT()*.

Arguments:

- *FLAG* (`int`, input) – flag denoting use of user-supplied preconditioning routines.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

In addition, the user must provide the following two routines to implement the preconditioner setup and solve functions to be used within the solve.

subroutine FARKMASSPSET(*T*, *IPAR*, *RPAR*, *IER*)

User-supplied preconditioner setup routine (of type *ARKLsMassPrecSetupFn()*).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable failure, <0 if a non-recoverable failure).

Notes: This routine must set up the preconditioner P to be used in the subsequent call to *FARKMASSPSOL()*. The preconditioner (or the product of the left and right preconditioners if using both) should be an approximation to the system mass matrix, M.

subroutine FARKMASSPSOL(*T*, *R*, *Z*, *DELTA*, *LR*, *IPAR*, *RPAR*, *IER*)

User-supplied preconditioner solve routine (of type *ARKLsMassPrecSolveFn()*).

Arguments:

- *T* (`realtype`, input) – current value of the independent variable.
- *R* (`realtype`, input) – right-hand side array.

- *Z* (`realtype`, output) – solution array.
- *DELTA* (`realtype`, input) – desired residual tolerance.
- *LR* (`int`, input) – flag denoting to solve the right or left preconditioner system: 1 = left preconditioner, 2 = right preconditioner.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable failure, <0 if a non-recoverable failure).

Notes: Typically this routine will use only *T*, *R*, *LR*, and *Z*. It must solve the preconditioner linear system $Pz = r$. The preconditioner (or the product of the left and right preconditioners if both are nontrivial) should be an approximation to the system mass matrix M.

Notes:

1. If the user's *FARKMASSPSOL()* uses an iterative method in its solution, the residual vector $\rho = r - Pz$ of the system should be made less than $\delta =$ *DELTA* in the weighted l2 norm, i.e.

$$\left(\sum_i (\rho_i \, EWT_i)^2\right)^{1/2} < \delta.$$

2. If needed in *FARKMTIMES()*, *FARKMTSETUP()*, *FARKMASSPSOL()*, or *FARKMASSPSET()*, the error weight array *EWT* can be obtained by calling *FARKGETERRWEIGHTS()* using a user-allocated array as temporary storage for *EWT*.
3. If needed in *FARKMTIMES()*, *FARKMTSETUP()*, *FARKMASSPSOL()*, or *FARKMASSPSET()*, the unit roundoff can be obtained as the optional output *ROUT(6)* (available after the call to *FARKMALLOC()*) and can be passed using either the *RPAR* user data array or a common block.

Problem solution

Carrying out the integration is accomplished by making calls to *FARKODE()*.

subroutine FARKODE(*TOUT*, *T*, *Y*, *ITASK*, *IER*)

Fortran interface to the C routine *ARKStepEvolve()* for performing the solve, along with many of the ARK*Get* routines for reporting on solver statistics.

Arguments:

- *TOUT* (`realtype`, input) – next value of t at which a solution is desired.
- *T* (`realtype`, output) – value of independent variable that corresponds to the output *Y*
- *Y* (`realtype`, output) – array containing dependent state variables on output.
- *ITASK* (`int`, input) – task indicator :
  - 1 = normal mode (overshoot *TOUT* and interpolate)
  - 2 = one-step mode (return after each internal step taken)
  - 3 = normal 'tstop' mode (like 1, but integration never proceeds past *TSTOP*, which must be specified through a preceding call to *FARKSETRIN()* using the key *STOP_TIME*)
  - 4 = one step 'tstop' mode (like 2, but integration never goes past *TSTOP*).

- *IER* (int, output) – completion flag:
  - 0 = success,
  - 1 = tstop return,
  - 2 = root return,
  - values -1, . . . , -10 are failure modes (see `ARKStepEvolve()` and *Appendix: ARKode Constants*).

Notes: The current values of the optional outputs are immediately available in *IOUT* and *ROUT* upon return from this function (see *Table: Optional FARKODE integer outputs* and *Table: Optional FARKODE real outputs*).

A full description of error flags and output behavior of the solver (values filled in for *T* and *Y*) is provided in the description of `ARKStepEvolve()`.

Additional solution output

After a successful return from `FARKODE()`, the routine `FARKDKY()` may be used to obtain a derivative of the solution, of order up to 3, at any t within the last step taken.

subroutine FARKDKY(*T*, *K*, *DKY*, *IER*)
Fortran interface to the C routine `ARKDKY()` for interpolating output of the solution or its derivatives at any point within the last step taken.

Arguments:

- *T* (`realtype`, input) – time at which solution derivative is desired, within the interval $[t_n - h, t_n]$.
- *K* (`int`, input) – derivative order $(0 \leq k \leq 3)$.
- *DKY* (`realtype`, output) – array containing the computed *K*-th derivative of y.
- *IER* (`int`, output) – return flag (0 if success, <0 if an illegal argument).

Problem reinitialization

To re-initialize the ARKStep solver for the solution of a new problem of the same size as one already solved, the user must call `FARKREINIT()`:

subroutine FARKREINIT(*T0*, *Y0*, *IMEX*, *IATOL*, *RTOL*, *ATOL*, *IER*)
Re-initializes the Fortran interface to the ARKStep solver.

Arguments: The arguments have the same names and meanings as those of `FARKMALLOC()`.

Notes: This routine performs no memory allocation, instead using the existing memory created by the previous `FARKMALLOC()` call. The call to specify the linear system solution method may or may not be needed.

Following a call to `FARKREINIT()` if the choice of linear solver is being changed then a user must make a call to create the alternate SUNLINSOL module and then attach it to ARKStep, as shown above. If only linear solver parameters are being modified, then these calls may be made without re-attaching to ARKStep.

Resizing the ODE system

For simulations involving changes to the number of equations and unknowns in the ODE system (e.g. when solving a spatially-adaptive PDE), the `FARKODE()` integrator may be "resized" between integration steps, through calls to the

FARKRESIZE() function, that interfaces with the C routine *ARKStepResize()*. This function modifies ARKStep's internal memory structures to use the new problem size, without destruction of the temporal adaptivity heuristics. It is assumed that the dynamical time scales before and after the vector resize will be comparable, so that all time-stepping heuristics prior to calling FARKRESIZE() remain valid after the call. If instead the dynamics should be re-calibrated, the FARKODE memory structure should be deleted with a call to *FARKFREE()*, and re-created with a call to *FARKMALLOC()*.

subroutine FARKRESIZE(*T0*, *Y0*, *HSCALE*, *ITOL*, *RTOL*, *ATOL*, *IER*)

Re-initializes the Fortran interface to the ARKStep solver for a differently-sized ODE system.

Arguments:

- *T0* (realtype, input) – initial value of the independent variable t.
- *Y0* (realtype, input) – array of dependent-variable initial conditions.
- *HSCALE* (realtype, input) – desired step size scale factor:
  - 1.0 is the default,
  - any value <= 0.0 results in the default.
- *ITOL* (int, input) – flag denoting that a new relative tolerance and vector of absolute tolerances are supplied in the *RTOL* and *ATOL* arguments:
  - 0 = retain the current scalar-valued relative and absolute tolerances, or the user-supplied error weight function, *FARKEWT()*.
  - 1 = *RTOL* contains the new scalar-valued relative tolerance and *ATOL* contains a new array of absolute tolerances.
- *RTOL* (realtype, input) – scalar relative tolerance.
- *ATOL* (realtype, input) – array of absolute tolerances.
- *IER* (int, output) – return flag (0 success, $\neq$ 0 failure).

Notes: This routine performs the opposite set of of operations as *FARKREINIT()*: it does not reinitialize any of the time-step heuristics, but it does perform memory reallocation.

Following a call to *FARKRESIZE()*, the internal data structures for all linear solver and matrix objects will be the incorrect size. Hence, calls must be made to re-create the linear system solver, mass matrix solver, linear system matrix, and mass matrix, followed by calls to attach the updated objects to ARKStep.

If any user-supplied linear solver helper routines were used (Jacobian evaluation, Jacobian-vector product, mass matrix evaluation, mass-matrix-vector product, preconditioning, etc.), then the relevant "set" routines to specify their usage must be called again **following** the re-specification of the linear solver module(s).

Memory deallocation

To free the internal memory created by *FARKMALLOC()*, *FARKLSINIT()*, *FARKLSMASSINIT()*, and the SUNMATRIX, SUNLINSOL and SUNNONLINSOL objects, the user may call *FARKFREE()*, as follows:

subroutine FARKFREE()

Frees the internal memory created by *FARKMALLOC()*.

Arguments: None.

7.2.2.3 FARKODE optional output

We note that the optional inputs to FARKODE have already been described in the section *Setting optional inputs*.

IOUT and ROUT arrays

In the Fortran interface, the optional outputs from the *FARKODE()* solver are accessed not through individual functions, but rather through a pair of user-allocated arrays, *IOUT* (having `long int` type) of dimension at least 36, and *ROUT* (having `realtype` type) of dimension at least 6. These arrays must be allocated by the user program that calls *FARKODE()*, that passes them through the Fortran interface as arguments to *FARKMALLOC()*. Following this call, *FARKODE()* will modify the entries of these arrays to contain all optional output values provided to a Fortran user.

In the following tables, *Table: Optional FARKODE integer outputs* and *Table: Optional FARKODE real outputs*, we list the entries in these arrays by index, naming them according to their role with the main ARKStep solver, and list the relevant ARKStep C/C++ function that is actually called to extract the output value. Similarly, optional integer output values that are specific to the ARKLS linear solver interface are listed in *Table: Optional ARKLS interface outputs*.

For more details on the optional inputs and outputs to ARKStep, see the sections *Optional input functions* and *Optional output functions*.

Table: Optional FARKODE integer outputs

IOUT Index	Optional output	ARKStep function
1	LENRW	*ARKStepGetWorkSpace()*
2	LENIW	*ARKStepGetWorkSpace()*
3	NST	*ARKStepGetNumSteps()*
4	NST_STB	*ARKStepGetNumExpSteps()*
5	NST_ACC	*ARKStepGetNumAccSteps()*
6	NST_ATT	*ARKStepGetNumStepAttempts()*
7	NFE	*ARKStepGetNumRhsEvals()* (num f^E calls)
8	NFI	*ARKStepGetNumRhsEvals()* (num f^I calls)
9	NSETUPS	*ARKStepGetNumLinSolvSetups()*
10	NETF	*ARKStepGetNumErrTestFails()*
11	NNI	*ARKStepGetNumNonlinSolvIters()*
12	NCFN	*ARKStepGetNumNonlinSolvConvFails()*
13	NGE	*ARKStepGetNumGEvals()*

Table: Optional FARKODE real outputs

ROUT Index	Optional output	ARKStep function
1	H0U	*ARKStepGetActualInitStep()*
2	HU	*ARKStepGetLastStep()*
3	HCUR	*ARKStepGetCurrentStep()*
4	TCUR	*ARKStepGetCurrentTime()*
5	TOLSF	*ARKStepGetTolScaleFactor()*
6	UROUND	`UNIT_ROUNDOFF` (see the section *Data Types*)

Table: Optional ARKLS interface outputs

IOUT Index	Optional output	ARKStep function
14	LENRWLS	`ARKLsGetWorkSpace()`
15	LENIWLS	`ARKLsGetWorkSpace()`
16	LSTF	`ARKLsGetLastFlag()`
17	NFELS	`ARKLsGetNumRhsEvals()`
18	NJE	`ARKLsGetNumJacEvals()`
19	NJTS	`ARKLsGetNumJTSetupEvals()`
20	NJTV	`ARKLsGetNumJtimesEvals()`
21	NPE	`ARKLsGetNumPrecEvals()`
22	NPS	`ARKLsGetNumPrecSolves()`
23	NLI	`ARKLsGetNumLinIters()`
24	NCFL	`ARKLsGetNumConvFails()`

Table: Optional ARKLS mass interface outputs

IOUT Index	Optional output	ARKStep function
25	LENRWMS	`ARKLsGetMassWorkSpace()`
26	LENIWMS	`ARKLsGetMassWorkSpace()`
27	LSTMF	`ARKLsGetLastMassFlag()`
28	NMSET	`ARKLsGetNumMassSetups()`
29	NMSOL	`ARKLsGetNumMassSolves()`
30	NMTSET	`ARKLsGetNumMTSetups()`
31	NMMUL	`ARKLsGetNumMassMult()`
32	NMPE	`ARKLsGetNumMassPrecEvals()`
33	NMPS	`ARKLsGetNumMassPrecSolves()`
34	NMLI	`ARKLsGetNumMassIters()`
35	NMCFL	`ARKLsGetNumMassConvFails()`

Table: Optional ARKode constraints outputs

IOUT Index	Optional output	ARKStep function
36	CONSTRFAILS	*`ARKStepGetNumConstrFails()`*

Additional optional output routines

In addition to the optional inputs communicated through FARKSET* calls and the optional outputs extracted from *IOUT* and *ROUT*, the following user-callable routines are available.

To obtain the error weight array *EWT*, containing the multiplicative error weights used in the WRMS norms, the user may call the routine *`FARKGETERRWEIGHTS()`* as follows:

`subroutine FARKGETERRWEIGHTS`(*EWT*, *IER*)

Retrieves the current error weight vector (interfaces with *`ARKStepGetErrWeights()`*).

Arguments:

- *EWT* (`realtype`, output) – array containing the error weight vector.

- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: The array *EWT* must have already been allocated by the user, of the same size as the solution array *Y*.

Similarly, to obtain the estimated local truncation errors, following a successful call to *FARKODE()*, the user may call the routine *FARKGETESTLOCALERR()* as follows:

subroutine FARKGETESTLOCALERR(*ELE*, *IER*)
Retrieves the current local truncation error estimate vector (interfaces with *ARKStepGetEstLocalErrors()*).

Arguments:

- *ELE* (`realtype`, output) – array with the estimated local truncation error vector.
- *IER* (`int`, output) – return flag (0 if success, $\neq 0$ if an error).

Notes: The array *ELE* must have already been allocated by the user, of the same size as the solution array *Y*.

7.2.2.4 Usage of the FARKROOT interface to rootfinding

The FARKROOT interface package allows programs written in Fortran to use the rootfinding feature of the ARKStep solver module. The user-callable functions in FARKROOT, with the corresponding ARKStep functions, are as follows:

- *FARKROOTINIT()* interfaces to *ARKStepRootInit()*,
- *FARKROOTINFO()* interfaces to *ARKStepGetRootInfo()*, and
- *FARKROOTFREE()* interfaces to *ARKStepRootInit()*, freeing memory by calling the initializer with no root functions.

Note that at this time, FARKROOT does not provide support to specify the direction of zero-crossing that is to be monitored. Instead, all roots are considered. However, the actual direction of zero-crossing may be captured by the user through monitoring the sign of any non-zero elements in the array *INFO* returned by *FARKROOTINFO()*.

In order to use the rootfinding feature of ARKStep, after calling *FARKMALLOC()* but prior to calling *FARKODE()*, the user must call *FARKROOTINIT()* to allocate and initialize memory for the FARKROOT module:

subroutine FARKROOTINIT(*NRTFN*, *IER*)
Initializes the Fortran interface to the FARKROOT module.

Arguments:

- *NRTFN* (`int`, input) – total number of root functions.
- *IER* (`int`, output) – return flag (0 success, -1 if ARKStep memory is `NULL`, and -11 if a memory allocation error occurred).

If rootfinding is enabled, the user must specify the functions whose roots are to be found. These rootfinding functions should be implemented in the user-supplied *FARKROOTFN()* subroutine:

subroutine FARKROOTFN(*T*, *Y*, *G*, *IPAR*, *RPAR*, *IER*)
User supplied function implementing the vector-valued function $g(t, y)$ such that the roots of the *NRTFN* components $g_i(t, y) = 0$ are sought.

Arguments:

- *T* (`realtype`, input) – independent variable value t.
- *Y* (`realtype`, input) – dependent variable array y.
- *G* (`realtype`, output) – function value array $g(t, y)$.

- *IPAR* (`long int`, input/output) – integer user data array, the same as the array passed to `FARKMALLOC()`.
- *RPAR* (`realtype`, input/output) – real-valued user data array, the same as the array passed to `FARKMALLOC()`.
- *IER* (`int`, output) – return flag (0 success, < 0 if error).

When making calls to `FARKODE()` to solve the ODE system, the occurrence of a root is flagged by the return value *IER = 2*. In that case, if *NRTFN > 1*, the functions $g_i(t, y)$ which were found to have a root can be identified by calling the routine `FARKROOTINFO()`:

subroutine FARKROOTINFO(*NRTFN*, *INFO*, *IER*)
Initializes the Fortran interface to the FARKROOT module.

Arguments:

- *NRTFN* (`int`, input) – total number of root functions.
- *INFO* (`int`, input/output) – array of length *NRTFN* with root information (must be allocated by the user). For each index, *i = 1, . . . , NRTFN*:
  - *INFO(i) = 1* if $g_i(t, y)$ was found to have a root, and g_i is increasing.
  - *INFO(i) = -1* if $g_i(t, y)$ was found to have a root, and g_i is decreasing.
  - *INFO(i) = 0* otherwise.
- *IER* (`int`, output) – return flag (0 success, < 0 if error).

The total number of calls made to the root function `FARKROOTFN()`, denoted *NGE*, can be obtained from *IOUT(12)*. If the FARKODE/ARKStep memory block is reinitialized to solve a different problem via a call to `FARKREINIT()`, then the counter *NGE* is reset to zero.

Lastly, to free the memory resources allocated by a prior call to `FARKROOTINIT()`, the user must make a call to `FARKROOTFREE()`:

subroutine FARKROOTFREE()
Frees memory associated with the FARKODE rootfinding module.

7.2.2.5 Usage of the FARKODE interface to built-in preconditioners

The FARKODE interface enables usage of the two built-in preconditioning modules ARKBANDPRE and ARKBBDPRE. Details on how these preconditioners work are provided in the section *Preconditioner modules*. In this section, we focus specifically on the Fortran interface to these modules.

Usage of the FARKBP interface to ARKBANDPRE

The FARKBP interface module is a package of C functions which, as part of the FARKODE interface module, support the use of the ARKStep solver with the serial or threaded NVector modules (*The NVECTOR_SERIAL Module*, *The NVECTOR_OPENMP Module* or *The NVECTOR_PTHREADS Module*), and the combination of the ARKBANDPRE preconditioner module (see the section *A serial banded preconditioner module*) with the ARKStep linear solver interface and any of the Krylov iterative linear solvers.

The two user-callable functions in this package, with the corresponding ARKStep function around which they wrap, are:

- `FARKBPINIT()` interfaces to `ARKBandPrecInit()`.
- `FARKBPOPT()` interfaces to the ARKBANDPRE optional output functions, `ARKBandPrecGetWorkSpace()` and `ARKBandPrecGetNumRhsEvals()`.

As with the rest of the FARKODE routines, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `farkbp.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in the section *Usage of the FARKODE interface module* are *italicized*.

1. *Right-hand side specification*
2. *NVECTOR module initialization*
3. SUNLINSOL module initialization

   Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT, supplying an argument to specify that the SUNLINSOL module should utilize left or right preconditioning.
4. *Problem specification*
5. *Set optional inputs*
6. Linear solver interface specification

   First, initialize the ARKStep linear solver interface by calling `FARKLSINIT()`.

   Optionally, to specify that ARKStep should use the supplied `FARKJTIMES()` and `FARKJTSETUP()` routines, the user should call `FARKLSSETJAC()` with FLAG $\neq$ 0, as described in the section *Iterative linear solvers*.

   Then, to initialize the ARKBANDPRE preconditioner, call the routine `FARKBPINIT()`, as follows:

   subroutine FARKBPINIT(*NEQ*, *MU*, *ML*, *IER*)
   Interfaces with the `ARKBandPrecInit()` function to allocate memory and initialize data associated with the ARKBANDPRE preconditioner.

   Arguments:

   - *NEQ* (`long int`, input) – problem size.
   - *MU* (`long int`, input) – upper half-bandwidth of the band matrix that is retained as an approximation of the Jacobian.
   - *ML* (`long int`, input) – lower half-bandwidth of the band matrix approximation to the Jacobian.
   - *IER* (`int`, output) – return flag (0 if success, -1 if a memory failure).
7. *Problem solution*
8. ARKBANDPRE optional outputs

   Optional outputs for ARKStep's linear solver interface are listed in *Table: Optional ARKLS interface outputs*. To obtain the optional outputs associated with the ARKBANDPRE module, the user should call the `FARKBPOPT()`, as specified below:

   subroutine FARKBPOPT(*LENRWBP*, *LENIWBP*, *NFEBP*)
   Interfaces with the ARKBANDPRE optional output functions.

   Arguments:

   - *LENRWBP* (`long int`, output) – length of real preconditioner work space (from `ARKBandPrecGetWorkSpace()`).
   - *LENIWBP* (`long int`, output) – length of integer preconditioner work space, in integer words (from `ARKBandPrecGetWorkSpace()`).

- *NFEBP* (`long int`, output) – number of $f^I(t,y)$ evaluations (from `ARKBandPrecGetNumRhsEvals()`)

9. *Additional solution output*
10. *Problem re-initialization*
11. *Memory deallocation*

    (The memory allocated for the FARKBP module is deallocated automatically by `FARKFREE()`)

Usage of the FARKBBD interface to ARKBBDPRE

The FARKBBD interface module is a package of C functions which, as part of the FARKODE interface module, support the use of the ARKStep solver with the parallel vector module (*The NVECTOR_PARALLEL Module*), and the combination of the ARKBBDPRE preconditioner module (see the section *A parallel band-block-diagonal preconditioner module*) with any of the Krylov iterative linear solvers.

The user-callable functions in this package, with the corresponding ARKStep and ARKBBDPRE functions, are as follows:

- `FARKBBDINIT()` interfaces to `ARKBBDPrecInit()`.
- `FARKBBDREINIT()` interfaces to `ARKBBDPrecReInit()`.
- `FARKBBDOPT()` interfaces to the ARKBBDPRE optional output functions.

In addition to the functions required for general FARKODE usage, the user-supplied functions required by this package are listed in the table below, each with the corresponding interface function which calls it (and its type within ARKBBDPRE or ARKStep).

Table: FARKBBD function mapping

FARKBBD routine (FORTRAN, user-supplied)	ARKStep routine (C, interface)	ARKStep interface function type
`FARKGLOCFN()`	FARKgloc	`ARKLocalFn()`
`FARKCOMMFN()`	FARKcfn	`ARKCommFn()`
`FARKJTIMES()`	FARKJtimes	`ARKLsJacTimesVecFn()`
`FARKJTSETUP()`	FARKJTSetup	`ARKLsJacTimesSetupFn()`

As with the rest of the FARKODE routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in the section *FARKODE routines*, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `farkbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in the section *Usage of the FARKODE interface module* are *italicized*.

1. *Right-hand side specification*
2. *NVECTOR module initialization*
3. SUNLINSOL module initialization

   Initialize one of the iterative SUNLINSOL modules, by calling one of FSUNPCGINIT, FSUNSPBCGSINIT, FSUNSPFGMRINIT, FSUNSPGMRINIT or FSUNSPTFQMRINIT, supplying an argument to specify that the SUNLINSOL module should utilize left or right preconditioning.

4. *Problem specification*
5. *Set optional inputs*

6. Linear solver interface specification

   First, initialize ARKStep's linear solver interface by calling *FARKLSINIT()*.

   Optionally, to specify that ARKStep should use the supplied *FARKJTIMES()* and *FARKJTSETUP()* routines, the user should call *FARKLSSETJAC()* with FLAG $\neq$ 0, as described in the section *Iterative linear solvers*.

   Then, to initialize the ARKBBDPRE preconditioner, call the function *FARKBBDINIT()*, as described below:

   subroutine FARKBBDINIT(*NLOCAL*, *MUDQ*, *MLDQ*, *MU*, *ML*, *DQRELY*, *IER*)
   Interfaces with the *ARKBBDPrecInit()* routine to initialize the ARKBBDPRE preconditioning module.

   Arguments:

   - *NLOCAL* (`long int`, input) – local vector size on this process.
   - *MUDQ* (`long int`, input) – upper half-bandwidth to be used in the computation of the local Jacobian blocks by difference quotients. These may be smaller than the true half-bandwidths of the Jacobian of the local block of g, when smaller values may provide greater efficiency.
   - *MLDQ* (`long int`, input) – lower half-bandwidth to be used in the computation of the local Jacobian blocks by difference quotients.
   - *MU* (`long int`, input) – upper half-bandwidth of the band matrix that is retained as an approximation of the local Jacobian block (may be smaller than *MUDQ*).
   - *ML* (`long int`, input) – lower half-bandwidth of the band matrix that is retained as an approximation of the local Jacobian block (may be smaller than *MLDQ*).
   - *DQRELY* (`realtype`, input) – relative increment factor in y for difference quotients (0.0 indicates to use the default).
   - *IER* (`int`, output) – return flag (0 if success, -1 if a memory failure).

7. *Problem solution*

8. ARKBBDPRE optional outputs

   Optional outputs from the ARKStep linear solver interface are listed in *Table: Optional ARKLS interface outputs*. To obtain the optional outputs associated with the ARKBBDPRE module, the user should call *FARKBBDOPT()*, as specified below:

   subroutine FARKBBDOPT(*LENRWBBD*, *LENIWBBD*, *NGEBBD*)
   Interfaces with the ARKBBDPRE optional output functions.

   Arguments:

   - *LENRWBP* (`long int`, output) – length of real preconditioner work space on this process (from *ARKBBDPrecGetWorkSpace()*).
   - *LENIWBP* (`long int`, output) – length of integer preconditioner work space on this process (from *ARKBBDPrecGetWorkSpace()*).
   - *NGEBBD* (`long int`, output) – number of $g(t, y)$ evaluations (from *ARKBBDPrecGetNumGfnEvals()*) so far.

9. *Additional solution output*

10. Problem re-initialization

    If a sequence of problems of the same size is being solved using the same linear solver in combination with the ARKBBDPRE preconditioner, then the ARKStep package can be re-initialized for the second and subsequent

problems by calling *FARKREINIT()*, following which a call to *FARKBBDREINIT()* may or may not be needed. If the input arguments are the same, no *FARKBBDREINIT()* call is needed.

If there is a change in input arguments other than *MU* or *ML*, then the user program should call *FARKBBDREINIT()* as specified below:

subroutine FARKBBDREINIT(*NLOCAL*, *MUDQ*, *MLDQ*, *DQRELY*, *IER*)
Interfaces with the *ARKBBDPrecReInit()* function to reinitialize the ARKBBDPRE module.

Arguments: The arguments of the same names have the same meanings as in *FARKBBDINIT()*.

However, if the value of *MU* or *ML* is being changed, then a call to *FARKBBDINIT()* must be made instead.

Finally, if there is a change in any of the linear solver inputs, then a call to one of `FSUNSPGMRINIT()`, `FSUNSPBCGSINIT()`, `FSUNSPTFQMRINIT()`, `FSUNSPFGMRINIT()` or `FSUNPCGINIT()`, followed by a call to *FARKLSINIT()* must also be made; in this case the linear solver memory is reallocated.

11. Problem resizing

    If a sequence of problems of different sizes (but with similar dynamical time scales) is being solved using the same linear solver (SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG) in combination with the ARKBBDPRE preconditioner, then the ARKStep package can be re-initialized for the second and subsequent problems by calling *FARKRESIZE()*, following which a call to *FARKBBDINIT()* is required to delete and re-allocate the preconditioner memory of the correct size.

    subroutine FARKBBDREINIT(*NLOCAL*, *MUDQ*, *MLDQ*, *DQRELY*, *IER*)
    Interfaces with the *ARKBBDPrecReInit()* function to reinitialize the ARKBBDPRE module.

    Arguments: The arguments of the same names have the same meanings as in *FARKBBDINIT()*.

    However, if the value of MU or ML is being changed, then a call to *FARKBBDINIT()* must be made instead.

    Finally, if there is a change in any of the linear solver inputs, then a call to one of `FSUNSPGMRINIT()`, `FSUNSPBCGSINIT()`, `FSUNSPTFQMRINIT()`, `FSUNSPFGMRINIT()` or `FSUNPCGINIT()`, followed by a call to *FARKLSINIT()* must also be made; in this case the linear solver memory is reallocated.

12. *Memory deallocation*

    (The memory allocated for the FARKBBD module is deallocated automatically by *FARKFREE()*).

13. User-supplied routines

    The following two routines must be supplied for use with the ARKBBDPRE module:

    subroutine FARKGLOCFN(*NLOC*, *T*, *YLOC*, *GLOC*, *IPAR*, *RPAR*, *IER*)
    User-supplied routine (of type *ARKLocalFn()*) that computes a processor-local approximation $g(t, y)$ to the right-hand side function $f^I(t, y)$.

    Arguments:

    - *NLOC* (`long int`, input) – local problem size.
    - *T* (`realtype`, input) – current value of the independent variable.
    - *YLOC* (`realtype`, input) – array containing local dependent state variables.
    - *GLOC* (`realtype`, output) – array containing local dependent state derivatives.
    - *IPAR* (`long int`, input/output) – array containing integer user data that was passed to *FARKMALLOC()*.
    - *RPAR* (`realtype`, input/output) – array containing real user data that was passed to *FARKMALLOC()*.
    - *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

subroutine FARKCOMMFN(*NLOC*, *T*, *YLOC*, *IPAR*, *RPAR*, *IER*)

User-supplied routine (of type *ARKCommFn()*) that performs all inter-process communication necessary for the execution of the *FARKGLOCFN()* function above, using the input vector *YLOC*.

Arguments:

- *NLOC* (`long int`, input) – local problem size.
- *T* (`realtype`, input) – current value of the independent variable.
- *YLOC* (`realtype`, input) – array containing local dependent state variables.
- *IPAR* (`long int`, input/output) – array containing integer user data that was passed to *FARKMALLOC()*.
- *RPAR* (`realtype`, input/output) – array containing real user data that was passed to *FARKMALLOC()*.
- *IER* (`int`, output) – return flag (0 if success, >0 if a recoverable error occurred, <0 if an unrecoverable error occurred).

Notes: This subroutine must be supplied even if it is not needed, and must return *IER = 0*.

Chapter 8

Butcher Table Data Structure

To store the Butcher table defining a Runge Kutta method ARKode provides the *ARKodeButcherTable* type and several related utilitiy routines. We use the following Butcher table notation (shown for a 3-stage method):

$$\begin{array}{c|c} c & A \\ \hline q & b \\ p & \tilde{b} \end{array} = \begin{array}{c|ccc} c_1 & a_{1,1} & a_{1,2} & a_{1,3} \\ c_2 & a_{2,1} & a_{2,2} & a_{2,3} \\ c_3 & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline q & b_1 & b_2 & b_3 \\ p & \tilde{b}_1 & \tilde{b}_2 & \tilde{b}_3 \end{array}$$

where the method and embedding share stage A and abscissa c values, but use their stages z_i differently through the coefficients b and $\tilde{b}$ to generate methods of orders q (the main method) and p (the embedding, typically $q = p + 1$, though sometimes this is reversed). *ARKodeButcherTable* is defined as

typedef ARKodeButcherTableMem* **ARKodeButcherTable**

where `ARKodeButcherTableMem` is the structure

```
typedef struct ARKodeButcherTableMem {

  int q;
  int p;
  int stages;
  realtype **A;
  realtype *c;
  realtype *b;
  realtype *d;

};
```

where `stages` is the number of stages in the RK method, the variables `q`, `p`, `A`, `c`, and `b` have the same meaning as in the Butcher table above, and `d` is used to store $\tilde{b}$.

8.1 ARKodeButcherTable functions

Function name	Description
ARKodeButcherTable_LoadERK()	Retrieve a given explicit Butcher table by its unique name
ARKodeButcherTable_LoadDIRK()	Retrieve a given implicit Butcher table by its unique name
ARKodeButcherTable_Alloc()	Allocate an empty Butcher table
ARKodeButcherTable_Create()	Create a new Butcher table
ARKodeButcherTable_Copy()	Create a copy of a Butcher table
ARKodeButcherTable_Space()	Get the Butcher table real and integer workspace size
ARKodeButcherTable_Free()	Deallocate a Butcher table
ARKodeButcherTable_Write()	Write the Butcher table to an output file
ARKodeButcherTable_CheckOrder()	Check the order of a Butcher table
ARKodeButcherTable_CheckARKOrder()	Check the order of an ARK pair of Butcher tables

ARKodeButcherTable **ARKodeButcherTable_LoadERK**(int *emethod*)

Retrieves a specified explicit Butcher table. The prototype for this function, as well as the integer names for each provided method, are defined in the header file `arkode/arkode_butcher_erk.h`. For further information on these tables and their corresponding identifiers, see *Appendix: Butcher tables*.

Arguments:

- *emethod* – integer input specifying the given Butcher table.

Return value:

- *ARKodeButcherTable* structure if successful.
- `NULL` pointer if *imethod* was invalid.

ARKodeButcherTable **ARKodeButcherTable_LoadDIRK**(int *imethod*)

Retrieves a specified diagonally-implicit Butcher table. The prototype for this function, as well as the integer names for each provided method, are defined in the header file `arkode/arkode_butcher_dirk.h`. For further information on these tables and their corresponding identifiers, see *Appendix: Butcher tables*.

Arguments:

- *imethod* – integer input specifying the given Butcher table.

Return value:

- *ARKodeButcherTable* structure if successful.
- `NULL` pointer if *imethod* was invalid.

ARKodeButcherTable **ARKodeButcherTable_Alloc**(int *stages*, booleantype *embedded*)

Allocates an empty Butcher table.

Arguments:

- *stages* – the number of stages in the Butcher table.
- *embedded* – flag denoting whether the Butcher table has an embedding (`SUNTRUE`) or not (`SUNFALSE`).

Return value:

- *ARKodeButcherTable* structure if successful.
- `NULL` pointer if *stages* was invalid or an allocation error occured.

ARKodeButcherTable **ARKodeButcherTable_Create**(int *s*, int *q*, int *p*, realtype **c*, realtype **A*, realtype **b*, realtype **d*)

Allocates a Butcher table and fills it with the given values.

Arguments:

- *s* – number of stages in the RK method.
- *q* – global order of accuracy for the RK method.
- *p* – global order of accuracy for the embedded RK method.
- *c* – array (of length *s*) of stage times for the RK method.
- *A* – array of coefficients defining the RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.
- *b* – array of coefficients (of length *s*) defining the time step solution.
- *d* – array of coefficients (of length *s*) defining the embedded solution.

Return value:

- *ARKodeButcherTable* structure if successful.
- `NULL` pointer if *stages* was invalid or an allocation error occured.

Notes: If the method does not have an embedding then *d* should be `NULL` and *p* should be equal to zero.

ARKodeButcherTable **ARKodeButcherTable_Copy**(*ARKodeButcherTable B*)

Creates copy of the given Butcher table.

Arguments:

- *B* – the Butcher table to copy.

Return value:

- *ARKodeButcherTable* structure if successful.
- `NULL` pointer an allocation error occured.

void **ARKodeButcherTable_Space**(*ARKodeButcherTable B*, sunindextype **liw*, sunindextype **lrw*)

Get the real and integer workspace size for a Butcher table.

Arguments:

- *B* – the Butcher table.
- *lenrw* – the number of `realtype` values in the Butcher table workspace.
- *leniw* – the number of integer values in the Butcher table workspace.

Return value:

- *ARK_SUCCESS* if successful.
- *ARK_MEM_NULL* if the Butcher table memory was `NULL`.

void **ARKodeButcherTable_Free**(*ARKodeButcherTable B*)

Deallocate the Butcher table memory.

Arguments:

- *B* – the Butcher table.

void **ARKodeButcherTable_Write**(*ARKodeButcherTable B*, FILE **outfile*)

Write the Butcher table to the provided file pointer.

Arguments:

- *B* – the Butcher table.
- *outfile* – pointer to use for printing the Butcher table.

Notes: The *outfile* argument can be `stdout` or `stderr`, or it may point to a specific file created using `fopen`.

int **ARKodeButcherTable_CheckOrder**(*ARKodeButcherTable B*, int* *q*, int* *p*, FILE* *outfile*)
Determine the analytic order of accuracy for the specified Butcher table. The analytic (necessary) conditions are checked up to order 6. For orders greater than 6 the Butcher simplifying (sufficient) assumptions are used.

Arguments:

- *B* – the Butcher table.
- *q* – the measured order of accuracy for the method.
- *p* – the measured order of accuracy for the embedding; 0 if the method does not have an embedding.
- *outfile* – file pointer for printing results; `NULL` to suppress output.

Return value:

- *0* – success, the measured vales of *q* and *p* match the values of *q* and *p* in the provided Butcher tables.
- *1* – warning, the values of *q* and *p* in the provided Butcher tables are *lower* than the measured values, or the measured values achieve the *maximum order* possible with this function and the values of *q* and *p* in the provided Butcher tables table are higher.
- *-1* – failure, the values of *q* and *p* in the provided Butcher tables are *higher* than the measured values.
- *-2* – failure, the input Butcher table or critical table contents are `NULL`.

Notes: For embedded methods, if the return flags for *q* and *p* would differ, failure takes precedence over warning, which takes precedence over success.

int **ARKodeButcherTable_CheckARKOrder**(*ARKodeButcherTable B1*, *ARKodeButcherTable B2*, int *q, int *p, FILE *outfile)
Determine the analytic order of accuracy (up to order 6) for a specified ARK pair of Butcher tables.

Arguments:

- *B1* – a Butcher table in the ARK pair.
- *B2* – a Butcher table in the ARK pair.
- *q* – the measured order of accuracy for the method.
- *p* – the measured order of accuracy for the embedding; 0 if the method does not have an embedding.
- *outfile* – file pointer for printing results; `NULL` to suppress output.

Return value:

- *0* – success, the measured vales of *q* and *p* match the values of *q* and *p* in the provided Butcher tables.
- *1* – warning, the values of *q* and *p* in the provided Butcher tables are *lower* than the measured values, or the measured values achieve the *maximum order* possible with this function and the values of *q* and *p* in the provided Butcher tables table are higher.
- *-1* – failure, the input Butcher tables or critical table contents are `NULL`.

Notes: For embedded methods, if the return flags for *q* and *p* would differ, warning takes precedence over success.

Chapter 9

ARKODE Features for GPU Accelerated Computing

This chapter is concerned with using GPU-acceleration and ARKODE for the solution of IVPs.

9.1 SUNDIALS GPU Programming Model

In this section, we introduce the SUNDIALS GPU programming model and highlight SUNDIALS GPU features. The model leverages the fact that all of the SUNDIALS packages interact with simulation data either through the shared vector, matrix, and solver APIs (see *Vector Data Structures*, *Matrix Data Structures*, *Description of the SUNLinearSolver module*, and *Description of the SUNNonlinearSolver Module*) or through user-supplied callback functions. Thus, under the model, the overall structure of the user's calling program, and the way users interact with the SUNDIALS packages is similar to using SUNDIALS in CPU-only environments.

Within the SUNDIALS GPU programming model, all control logic executes on the CPU, and all simulation data resides wherever the vector or matrix object dictates as long as SUNDIALS is in control of the program. That is, SUNDIALS will not migrate data (explicitly) from one memory space to another. Except in the most advanced use cases, it is safe to assume that data is kept resident in the GPU-device memory space. The consequence of this is that, when control is passed from the user's calling program to SUNDIALS, simulation data in vector or matrix objects must be up-to-date in the device memory space. Similarly, when control is passed from SUNDIALS to the user's calling program, the user should assume that any simulation data in vector and matrix objects are up-to-date in the device memory space. To put it succinctly, *it is the responsibility of the user's calling program to manage data coherency between the CPU and GPU-device memory spaces* unless unified virtual memory (UVM), also known as managed memory, is being utilized. Typically, the GPU-enabled SUNDIALS modules provide functions to copy data from the host to the device and vice-versa as well as support for unmanaged memory or UVM. In practical terms, the way SUNDIALS handles distinct host and device memory spaces means that *users need to ensure that the user-supplied functions, e.g. the right-hand side function, only operate on simulation data in the device memory space* otherwise extra memory transfers will be required and performance will be poor. The exception to this rule is if some form of hybrid data partitioning (achievable with the *The NVECTOR_MANYVECTOR Module*) is utilized.

SUNDIALS provides many native shared features and modules that are GPU-enabled. Currently, these are primarily limited to the NVIDIA CUDA platform *[CUDA]*, although support for more GPU computing platforms such as AMD ROCm/HIP *[ROCm]* and Intel oneAPI *[oneAPI]*, is an area of active development. Table *List of SUNDIALS GPU-enabled Modules* summarizes the shared SUNDIALS modules that are GPU-enabled, what GPU programming environments they support, and what class of memory they support (unmanaged or UVM). Users may also supply their own GPU-enabled `N_Vector`, `SUNMatrix`, `SUNLinearSolver`, or `SUNNonlinearSolver` implementation, and the capabilties will be leveraged since SUNDIALS operates on data through these APIs.

In addition, SUNDIALS provides *Tools for Memory Management* to support applications which implement their own memory management or memory pooling.

Table 9.1: List of SUNDIALS GPU-enabled Modules

Module	CUDA	ROCm/HIP	oneAPI	Unmanaged Memory	UVM
NVECTOR_CUDA	X			X	X
NVECTOR_RAJA	X			X	X
NVECTOR_OPENMPDEV	X	X[2]	X[2]	X	
SUNMATRIX_CUSPARSE	X			X	X
SUNLINSOL_CUSOLVERSP	X			X	X
SUNLINSOL_SPGMR	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_SPFGMR	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_SPTFQMR	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_SPBCGS	X[1]	X[1]	X[1]	X[1]	X[1]
SUNLINSOL_PCG	X[1]	X[1]	X[1]	X[1]	X[1]
SUNNONLINSOL_NEWTON	X[1]	X[1]	X[1]	X[1]	X[1]
SUNNONLINSOL_FIXEDPOINT	X[1]	X[1]	X[1]	X[1]	X[1]

1. This module inherits support from the NVECTOR module used
2. Support for ROCm/HIP and oneAPI are currently untested.

In addition, note that implicit UVM (i.e. `malloc` returning UVM) is not accounted for.

9.2 Steps for Using GPU Accelerated SUNDIALS

For any SUNDIALS package, the generalized steps a user needs to take to use GPU accelerated SUNDIALS are:

1. Utilize a GPU-enabled vector implementation. Initial data can be loaded on the host, but must be in the device memory space prior to handing control to SUNDIALS.
2. Utilize a GPU-enabled linear solver (if necessary).
3. Utilize a GPU-enabled implementation (if using a matrix-based linear solver).
4. Utilize a GPU-enabled nonlinear solver (if necessary).
5. Write user-supplied functions so that they use data only in the device memory space (again, unless an atypical data partitioning is used). A few examples of these functions are the right-hand side evaluation function, the Jacobian evalution function, or the preconditioner evaulation function. In the context of CUDA and the right-hand side function, one way a user might ensure data is accessed on the device is, for example, calling a CUDA kernel, which does all of the computation, from a CPU function which simply extracts the underlying device data array from the vector object that is passed from SUNDIALS to the user-supplied function.

Users should refer to Table *List of SUNDIALS GPU-enabled Modules* for a list of GPU-enabled native SUNDIALS modules.

Chapter 10

Vector Data Structures

The SUNDIALS library comes packaged with a variety of NVECTOR implementations, designed for simulations in serial, shared-memory parallel, and distributed-memory parallel environments, as well as interfaces to vector data structures used within external linear solver libraries. All native implementations assume that the process-local data is stored contiguously, and they in turn provide a variety of standard vector algebra operations that may be performed on the data.

In addition, SUNDIALS provides a simple interface for generic vectors (akin to a C++ *abstract base class*). All of the major SUNDIALS solvers (CVODE(s), IDA(s), KINSOL, ARKODE) in turn are constructed to only depend on these generic vector operations, making them immediately extensible to new user-defined vector objects. The only exceptions to this rule relate to the dense, banded and sparse-direct linear system solvers, since they rely on particular data storage and access patterns in the NVECTORS used.

10.1 Description of the NVECTOR Modules

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by, and specific to, the particular NVECTOR implementation. Users can provide a custom implementation of the NVECTOR module or use one of four provided within SUNDIALS – a serial and three parallel implementations. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as:

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
   void *content;
   struct _generic_N_Vector_Ops *ops;
};
```

Here, the `_generic_N_Vector_Op` structure is essentially a list of function pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
   N_Vector_ID  (*nvgetvectorid)(N_Vector);
   N_Vector     (*nvclone)(N_Vector);
   N_Vector     (*nvcloneempty)(N_Vector);
```

```
  void          (*nvdestroy)(N_Vector);
  void          (*nvspace)(N_Vector, sunindextype *, sunindextype *);
  realtype*     (*nvgetarraypointer)(N_Vector);
  realtype*     (*nvgetdevicearraypointer)(N_Vector);
  void          (*nvsetarraypointer)(realtype *, N_Vector);
  void*         (*nvgetcommunicator)(N_Vector);
  sunindextype  (*nvgetlength)(N_Vector);
  void          (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void          (*nvconst)(realtype, N_Vector);
  void          (*nvprod)(N_Vector, N_Vector, N_Vector);
  void          (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void            (*nvscale)(realtype, N_Vector, N_Vector);
  void            (*nvabs)(N_Vector, N_Vector);
  void              (*nvinv)(N_Vector, N_Vector);
  void              (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype      (*nvdotprod)(N_Vector, N_Vector);
  realtype      (*nvmaxnorm)(N_Vector);
  realtype      (*nvwrmsnorm)(N_Vector, N_Vector);
  realtype      (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype      (*nvmin)(N_Vector);
  realtype      (*nvwl2norm)(N_Vector, N_Vector);
  realtype      (*nvl1norm)(N_Vector);
  void              (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype   (*nvinvtest)(N_Vector, N_Vector);
  booleantype   (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype      (*nvminquotient)(N_Vector, N_Vector);
  int           (*nvlinearcombination)(int, realtype *, N_Vector *, N_Vector);
  int           (*nvscaleaddmulti)(int, realtype *, N_Vector, N_Vector *, N_Vector *);
  int           (*nvdotprodmulti)(int, N_Vector, N_Vector *,  realtype *);
  int           (*nvlinearsumvectorarray)(int, realtype, N_Vector *, realtype,
                                          N_Vector *, N_Vector *);
  int           (*nvscalevectorarray)(int, realtype *,  N_Vector *, N_Vector *);
  int           (*nvconstvectorarray)(int, realtype, N_Vector *);
  int           (*nvwrmsnomrvectorarray)(int, N_Vector *, N_Vector *, realtype *);
  int           (*nvwrmsnomrmaskvectorarray)(int, N_Vector *, N_Vector *, N_Vector,
                                             realtype *);
  int           (*nvscaleaddmultivectorarray)(int, int, realtype *, N_Vector *,
                                              N_Vector **, N_Vector **);
  int           (*nvlinearcombinationvectorarray)(int, int, realtype *, N_Vector **,
                                                  N_Vector *);
  realtype      (*nvdotprodlocal)(N_Vector, N_Vector);
  realtype      (*nvmaxnormlocal)(N_Vector);
  realtype      (*nvminlocal)(N_Vector);
  realtype      (*nvl1normlocal)(N_Vector);
  booleantype   (*nvinvtestlocal)(N_Vector, N_Vector);
  booleantype   (*nvconstrmasklocal)(N_Vector, N_Vector, N_Vector);
  realtype      (*nvminquotientlocal)(N_Vector, N_Vector);
  realtype      (*nvwsqrsumlocal)(N_Vector, N_Vector);
  realtype      (*nvwsqrsummasklocal(N_Vector, N_Vector, N_Vector);
  int           (*nvbufsize)(N_Vector, sunindextype *);
  int           (*nvbufpack)(N_Vector, void*);
  int           (*nvbufunpack)(N_Vector, void*);
};
```

The generic NVECTOR module defines and implements the vector operations acting on a `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the

scaling of a vector x by a scalar c:

```
void N_VScale(realtype c, N_Vector x, N_Vector z) {
   z->ops->nvscale(c, x, z);
}
```

The subsection *Description of the NVECTOR operations* contains a complete list of all standard vector operations defined by the generic NVECTOR module. The subsections *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations* and *Description of the NVECTOR local reduction operations*, *Description of the NVECTOR exchange operations* list *optional* fused, vector array, local reduction, and exchange operations respectively.

Fused and vector array operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines a fused or vector array operation as NULL, the generic NVECTOR module will automatically call standard vector operations as necessary to complete the desired operation. In all SUNDIALS-provided NVECTOR implementations, all fused and vector array operations are disabled by default. However, these implementations provide additional user-callable functions to enable/disable any or all of the fused and vector array operations. See the following sections for the implementation specific functions to enable/disable operations.

Local reduction operations are similarly intended to reduce parallel communication on distributed memory systems, particularly when NVECTOR objects are combined together within a NVECTOR_MANYVECTOR object (see the section *The NVECTOR_MANYVECTOR Module*). If a particular NVECTOR implementation defines a local reduction operation as NULL, the NVECTOR_MANYVECTOR module will automatically call standard vector reduction operations as necessary to complete the desired operation. All SUNDIALS-provided NVECTOR implementations include these local reduction operations, which may be used as templates for user-defined NVECTOR implementations.

The exchange operations are intended only for use with the XBraid library for parallel-in-time integration and are otherwise unused by SUNDIALS packages.

10.1.1 NVECTOR Utility Functions

The generic NVECTOR module also defines the utility functions N_VCloneVectorArray, N_VCloneVectorArrayEmpty, and N_VDestroyVectorArray. Both clone functions create (by cloning) an array of *count* variables of type N_Vector, each of the same type as an existing N_Vector. Their prototypes are:

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneVectorArrayEmpty(int count, N_Vector w);
```

and their definitions are based on the implementation-specific N_VClone and N_VCloneEmpty operations, respectively. An array of variables of type N_Vector can be destroyed by calling N_VDestroyVectorArray, whose prototype is

```
void N_VDestroyVectorArray(N_Vector *vs, int count);
```

and whose definition is based on the implementation-specific N_VDestroy operation.

Finally, we note that users of the Fortran 2003 interface may be interested in the additional utility functions N_NewVectorArray, N_VGetVecAtIndexVectorArray, and N_VSetVecAtIndexVectorArray. These functions allow a Fortran 2003 user to create an empty vector array, get a vector at an index, and set a vector at an index. There prototypes are given below:

```
N_Vector *N_VNewVectorArray(int count);
N_Vector *N_VGetVecAtIndexVectorArray(N_Vector* vs, int index);
void N_VSetVecAtIndexVectorArray(N_Vector* vs, int index, N_Vector w)
```

10.1.2 Vector Identifications associated with vector kernels supplied with SUNDIALS

Vector ID	Vector type	ID Value
SUNDIALS_NVEC_SERIAL	Serial	0
SUNDIALS_NVEC_PARALLEL	Distributed memory parallel (MPI)	1
SUNDIALS_NVEC_OPENMP	OpenMP shared memory parallel	2
SUNDIALS_NVEC_PTHREADS	PThreads shared memory parallel	3
SUNDIALS_NVEC_PARHYP	*hypre* ParHyp parallel vector	4
SUNDIALS_NVEC_PETSC	PETSc parallel vector	5
SUNDIALS_NVEC_CUDA	CUDA parallel vector	6
SUNDIALS_NVEC_RAJA	RAJA parallel vector	7
SUNDIALS_NVEC_OPENMPDEV	OpenMP parallel vector with device offloading	8
SUNDIALS_NVEC_TRILINOS	Trilinos Tpetra vector	9
SUNDIALS_NVEC_MANYVECTOR	"ManyVector" vector	10
SUNDIALS_NVEC_MPIMANYVECTOR	MPI-enabled "ManyVector" vector	11
SUNDIALS_NVEC_MPIPLUSX	MPI+X vector	12
SUNDIALS_NVEC_CUSTOM	User-provided custom vector	13

10.1.3 Implementing a custom NVECTOR

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

To aid in the creation of custom NVECTOR modules the generic NVECTOR module provides two utility functions *`N_VNewEmpty()`* and *`N_VCopyOps()`*. When used in custom NVECTOR constructors and clone routines these functions will ease the introduction of any new optional vector operations to the NVECTOR API by ensuring only required operations need to be set and all operations are copied when cloning a vector.

N_Vector **`N_VNewEmpty`**`()`
This allocates a new generic `N_Vector` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

Return value: If successful, this function returns an `N_Vector` object. If an error occurs when allocating the object, then this routine will return `NULL`.

void **`N_VFreeEmpty`**(N_Vector *v*)
This routine frees the generic `N_Vector` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is `NULL`, and, if it is not, it will free it as well.

Arguments:

- *v* – an N_Vector object

int **N_VCopyOps**(N_Vector *w*, N_Vector *v*)
This function copies the function pointers in the `ops` structure of `w` into the `ops` structure of `v`.

Arguments:

- *w* – the vector to copy operations from
- *v* – the vector to copy operations to

Return value: If successful, this function returns `0`. If either of the inputs are `NULL` or the `ops` structure of either input is `NULL`, then is function returns a non-zero value.

Each NVECTOR implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in the table below. It is recommended that a user supplied NVECTOR implementation use the `SUNDIALS_NVEC_CUSTOM` identifier.

10.1.3.1 Support for complex-valued vectors

While SUNDIALS itself is written under an assumption of real-valued data, it does provide limited support for complex-valued problems. However, since none of the built-in NVECTOR modules supports complex-valued data, users must provide a custom NVECTOR implementation for this task. Many of the NVECTOR routines described in the subsections *Description of the NVECTOR operations* through *Description of the NVECTOR local reduction operations* above naturally extend to complex-valued vectors; however, some do not. To this end, we provide the following guidance:

- *N_VMin()* and `N_VMinLocal())()` should return the minimum of all *real* components of the vector, i.e., $m = \min_i \text{real}(x_i)$.
- *N_VConst()* (and similarly *N_VConstVectorArray()*) should set the real components of the vector to the input constant, and set all imaginary components to zero, i.e., $z_i = c + 0j,\ i = 0, \ldots, n-1$.
- *N_VAddConst()* should only update the real components of the vector with the input constant, leaving all imaginary components unchanged.
- *N_VWrmsNorm()*, *N_VWrmsNormMask()*, *N_VWSqrSumLocal()* and *N_VWSqrSumMaskLocal()* should assume that all entries of the weight vector `w` and the mask vector `id` are real-valued.
- *N_VDotProd()* should mathematically return a complex number for complex-valued vectors; as this is not possible with SUNDIALS' current `realtype`, this routine should be set to `NULL` in the custom NVECTOR implementation.
- *N_VCompare()*, *N_VConstrMask()*, *N_VMinQuotient()*, *N_VConstrMaskLocal()* and *N_VMinQuotientLocal()* are ill-defined due to the lack of a clear ordering in the complex plane. These routines should be set to `NULL` in the custom NVECTOR implementation.

While many SUNDIALS solver modules may be utilized on complex-valued data, others cannot. Specifically, although both SUNNonlinearSolver_Newton and SUNNonlinearSolver_FixedPoint may be used with any of the IVP solvers (CVODE(S), IDA(S) and ARKode) for complex-valued problems, the Anderson-acceleration feature SUNNonlinearSolver_FixedPoint cannot be used due to its reliance on *N_VDotProd()*. By this same logic, the Anderson acceleration feature within KINSOL also will not work with complex-valued vectors.

Similarly, although each package's linear solver interface (e.g., ARKLS) may be used on complex-valued problems, none of the built-in SUNMatrix or SUNLinearSolver modules work. Hence a complex-valued user should provide a custom SUNLinearSolver (and optionally a custom SUNMatrix) implementation for solving linear systems, and then attach this module as normal to the package's linear solver interface.

Finally, constraint-handling features of each package cannot be used for complex-valued data, due to the issue of ordering in the complex plane discussed above with *N_VCompare()*, *N_VConstrMask()*, *N_VMinQuotient()*, *N_VConstrMaskLocal()* and *N_VMinQuotientLocal()*.

We provide a simple example of a complex-valued example problem, including a custom complex-valued Fortran 2003 NVECTOR module, in the files `examples/arkode/F2003_custom/ark_analytic_complex_f2003.f90`, `examples/arkode/F2003_custom/fnvector_complex_mod.f90`, and `examples/arkode/F2003_custom/test_fnvector_complex_mod.f90`.

10.2 Description of the NVECTOR operations

The standard vector operations defined by the generic `N_Vector` module are defined as follows. For each of these operations, we give the name, usage of the function, and a description of its mathematical operations below.

N_Vector_ID **`N_VGetVectorID`**(N_Vector *w*)
: Returns the vector type identifier for the vector `w`. It is used to determine the vector implementation type (e.g. serial, parallel, ...) from the abstract `N_Vector` interface. Returned values are given in the table, *Vector Identifications associated with vector kernels supplied with SUNDIALS*

  Usage:

```
id = N_VGetVectorID(w);
```

N_Vector **`N_VClone`**(N_Vector *w*)
: Creates a new `N_Vector` of the same type as an existing vector *w* and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.

  Usage:

```
v = N_VClone(w);
```

N_Vector **`N_VCloneEmpty`**(N_Vector *w*)
: Creates a new `N_Vector` of the same type as an existing vector *w* and sets the *ops* field. It does not allocate storage for the new vector's data.

  Usage:

```
v = N_VCloneEmpty(w);
```

void **`N_VDestroy`**(N_Vector *v*)
: Destroys the `N_Vector` *v* and frees memory allocated for its internal data.

  Usage:

```
N_VDestroy(v);
```

void **`N_VSpace`**(N_Vector *v*, sunindextype* *lrw*, sunindextype* *liw*)
: Returns storage requirements for the `N_Vector` *v*: *lrw* contains the number of `realtype` words and *liw* contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.

  Usage:

```
N_VSpace(nvSpec, &lrw, &liw);
```

realtype* **`N_VGetArrayPointer`**(N_Vector *v*)
: Returns a pointer to a `realtype` array from the `N_Vector` *v*. Note that this assumes that the internal data in the `N_Vector` is a contiguous array of `realtype` and is accesible from the CPU.

This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.

Usage:

```
vdata = N_VGetArrayPointer(v);
```

realtype* **N_VGetDeviceArrayPointer**(N_Vector *v*)
Returns a device pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype` and is accessible from the device (e.g., GPU).

This operation is *optional* except when using the GPU-enabled direct linear solvers.

Usage:

```
vdata = N_VGetArrayPointer(v);
```

void **N_VSetArrayPointer**(realtype* *vdata*, N_Vector *v*)
Replaces the data array pointer in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in the `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module.

Usage:

```
N_VSetArrayPointer(vdata,v);
```

void* **N_VGetCommunicator**(N_Vector *v*)
Returns a pointer to the `MPI_Comm` object associated with the vector (if applicable). For MPI-unaware vector implementations, this should return `NULL`.

Usage:

```
commptr = N_VGetCommunicator(v);
```

sunindextype **N_VGetLength**(N_Vector *v*)
Returns the global length (number of 'active' entries) in the NVECTOR *v*. This value should be cumulative across all processes if the vector is used in a parallel environment. If *v* contains additional storage, e.g., for parallel communication, those entries should not be included.

Usage:

```
global_length = N_VGetLength(v);
```

void **N_VLinearSum**(realtype *a*, N_Vector *x*, realtype *b*, N_Vector *y*, N_Vector *z*)
Performs the operation *z* = *ax* + *by*, where *a* and *b* are `realtype` scalars and *x* and *y* are of type `N_Vector`:

$$z_i = ax_i + by_i, \quad i = 0, \ldots, n-1.$$

The output vector *z* can be the same as either of the input vectors (*x* or *y*).

Usage:

```
N_VLinearSum(a, x, b, y, z);
```

void **N_VConst**(realtype *c*, N_Vector *z*)
Sets all components of the `N_Vector` *z* to `realtype` *c*:

$$z_i = c, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VConst(c, z);
```

void **N_VProd** (N_Vector *x*, N_Vector *y*, N_Vector *z*)
Sets the `N_Vector` *z* to be the component-wise product of the `N_Vector` inputs *x* and *y*:

$$z_i = x_i y_i, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VProd(x, y, z);
```

void **N_VDiv** (N_Vector *x*, N_Vector *y*, N_Vector *z*)
Sets the `N_Vector` *z* to be the component-wise ratio of the `N_Vector` inputs *x* and *y*:

$$z_i = \frac{x_i}{y_i}, \quad i = 0, \ldots, n-1.$$

The y_i may not be tested for 0 values. It should only be called with a *y* that is guaranteed to have all nonzero components.

Usage:

```
N_VDiv(x, y, z);
```

void **N_VScale** (realtype *c*, N_Vector *x*, N_Vector *z*)
Scales the `N_Vector` *x* by the `realtype` scalar *c* and returns the result in *z*:

$$z_i = c x_i, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VScale(c, x, z);
```

void **N_VAbs** (N_Vector *x*, N_Vector *z*)
Sets the components of the `N_Vector` *z* to be the absolute values of the components of the `N_Vector` *x*:

$$z_i = |x_i|, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VAbs(x, z);
```

void **N_VInv** (N_Vector *x*, N_Vector *z*)
Sets the components of the `N_Vector` *z* to be the inverses of the components of the `N_Vector` *x*:

$$z_i = 1.0/x_i, \quad i = 0, \ldots, n-1.$$

This routine may not check for division by 0. It should be called only with an *x* which is guaranteed to have all nonzero components.

Usage:

```
N_VInv(x, z);
```

void **N_VAddConst** (N_Vector *x*, realtype *b*, N_Vector *z*)
Adds the `realtype` scalar *b* to all components of *x* and returns the result in the `N_Vector` *z*:

$$z_i = x_i + b, \quad i = 0, \ldots, n-1.$$

Usage:

```
N_VAddConst(x, b, z);
```

realtype **N_VDotProd**(N_Vector *x*, N_Vector *z*)
Returns the value of the dot-product of the `N_Vectors` *x* and *y*:

$$d = \sum_{i=0}^{n-1} x_i y_i.$$

Usage:

```
d = N_VDotProd(x, y);
```

realtype **N_VMaxNorm**(N_Vector *x*)
Returns the value of the l_∞ norm of the `N_Vector` *x*:

$$m = \max_{0 \le i \le n-1} |x_i|.$$

Usage:

```
m = N_VMaxNorm(x);
```

realtype **N_VWrmsNorm**(N_Vector *x*, N_Vector *w*)
Returns the weighted root-mean-square norm of the `N_Vector` *x* with (positive) `realtype` weight vector *w*:

$$m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2\right) / n}$$

Usage:

```
m = N_VWrmsNorm(x, w);
```

realtype **N_VWrmsNormMask**(N_Vector *x*, N_Vector *w*, N_Vector *id*)
Returns the weighted root mean square norm of the `N_Vector` *x* with `realtype` weight vector *w* built using only the elements of *x* corresponding to positive elements of the `N_Vector` *id*:

$$m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i H(id_i))^2\right) / n},$$

where $H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \le 0 \end{cases}$.

Usage:

```
m = N_VWrmsNormMask(x, w, id);
```

realtype **N_VMin**(N_Vector *x*)
Returns the smallest element of the `N_Vector` *x*:

$$m = \min_{0 \le i \le n-1} x_i.$$

Usage:

```
m = N_VMin(x);
```

realtype **N_VWl2Norm** (N_Vector *x*, N_Vector *w*)
Returns the weighted Euclidean l_2 norm of the `N_Vector` *x* with `realtype` weight vector *w*:

$$m = \sqrt{\sum_{i=0}^{n-1} (x_i w_i)^2}.$$

Usage:

```
m = N_VWL2Norm(x, w);
```

realtype **N_VL1Norm** (N_Vector *x*)
Returns the l_1 norm of the `N_Vector` *x*:

$$m = \sum_{i=0}^{n-1} |x_i|.$$

Usage:

```
m = N_VL1Norm(x);
```

void **N_VCompare** (realtype *c*, N_Vector *x*, N_Vector *z*)
Compares the components of the `N_Vector` *x* to the `realtype` scalar *c* and returns an `N_Vector` *z* such that for all $0 \leq i \leq n-1$,

$$z_i = \begin{cases} 1.0 & \text{if } |x_i| \geq c, \\ 0.0 & \text{otherwise} \end{cases}.$$

Usage:

```
N_VCompare(c, x, z);
```

booleantype **N_VInvTest** (N_Vector *x*, N_Vector *z*)
Sets the components of the `N_Vector` *z* to be the inverses of the components of the `N_Vector` *x*, with prior testing for zero values:

$$z_i = 1.0/x_i, \quad i = 0, \ldots, n-1.$$

This routine returns a boolean assigned to `SUNTRUE` if all components of *x* are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

Usage:

```
t = N_VInvTest(x, z);
```

booleantype **N_VConstrMask** (N_Vector *c*, N_Vector *x*, N_Vector *m*)
Performs the following constraint tests based on the values in c_i:

$$\begin{aligned} x_i > 0 \text{ if } c_i = 2, \\ x_i \geq 0 \text{ if } c_i = 1, \\ x_i < 0 \text{ if } c_i = -2, \\ x_i \leq 0 \text{ if } c_i = -1. \end{aligned}$$

There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `SUNFALSE` if any element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector *m*, with elements

equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Usage:

```
t = N_VConstrMask(c, x, m);
```

realtype **N_VMinQuotient** (N_Vector *num*, N_Vector *denom*)

This routine returns the minimum of the quotients obtained by termwise dividing the elements of *n* by the elements in *d*:

$$\min_{i=0,\ldots,n-1} \frac{\text{num}_i}{\text{denom}_i}.$$

A zero element in *denom* will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

Usage:

```
minq = N_VMinQuotient(num, denom);
```

10.2.1 Description of the NVECTOR fused operations

The following fused vector operations are *optional*. These operations are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused vector operations as `NULL`, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

int **N_VLinearCombination** (int *nv*, realtype* c, N_Vector* *X*, N_Vector z)

This routine computes the linear combination of *nv* vectors with n elements:

$$z_i = \sum_{j=0}^{nv-1} c_j x_{j,i}, \quad i = 0, \ldots, n-1,$$

where c is an array of nv scalars, x_j is a vector in the vector array *X*, and z is the output vector. If the output vector z is one of the vectors in *X*, then it *must* be the first vector in the vector array. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VLinearCombination(nv, c, X, z);
```

int **N_VScaleAddMulti** (int *nv*, realtype* c, N_Vector x, N_Vector* Y, N_Vector* Z)

This routine scales and adds one vector to *nv* vectors with n elements:

$$z_{j,i} = c_j x_i + y_{j,i}, \quad j = 0, \ldots, nv-1 \quad i = 0, \ldots, n-1,$$

where c is an array of scalars, x is a vector, y_j is a vector in the vector array Y, and z_j is an output vector in the vector array Z. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VScaleAddMulti(nv, c, x, Y, Z);
```

int **N_VDotProdMulti** (int *nv*, N_Vector *x*, N_Vector* *Y*, realtype* *d*)
This routine computes the dot product of a vector with *nv* vectors having n elements:

$$d_j = \sum_{i=0}^{n-1} x_i y_{j,i}, \quad j = 0, \ldots, nv-1,$$

where *d* is an array of scalars containing the computed dot products, *x* is a vector, and y_j is a vector the vector array *Y*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VDotProdMulti(nv, x, Y, d);
```

10.2.2 Description of the NVECTOR vector array operations

The following vector array operations are also *optional*. As with the fused vector operations, these are intended to increase data reuse, reduce parallel communication on distributed memory systems, and lower the number of kernel launches on systems with accelerators. If a particular NVECTOR implementation defines one of the fused or vector array operations as `NULL`, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

int **N_VLinearSumVectorArray** (int *nv*, realtype *a*, N_Vector *X*, realtype *b*, N_Vector* *Y*, N_Vector* *Z*)
This routine computes the linear sum of two vector arrays of *nv* vectors with n elements:

$$z_{j,i} = a x_{j,i} + b y_{j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where *a* and *b* are scalars, x_j and y_j are vectors in the vector arrays *X* and *Y* respectively, and z_j is a vector in the output vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VLinearSumVectorArray(nv, a, X, b, Y, Z);
```

int **N_VScaleVectorArray** (int *nv*, realtype* *c*, N_Vector* *X*, N_Vector* *Z*)
This routine scales each element in a vector of n elements in a vector array of *nv* vectors by a potentially different constant:

$$z_{j,i} = c_j x_{j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where *c* is an array of scalars, x_j is a vector in the vector array *X*, and z_j is a vector in the output vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VScaleVectorArray(nv, c, X, Z);
```

int **N_VConstVectorArray** (int *nv*, realtype *c*, N_Vector* *Z*)
This routine sets each element in a vector of n elements in a vector array of *nv* vectors to the same value:

$$z_{j,i} = c, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where *c* is a scalar and z_j is a vector in the vector array *Z*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VConstVectorArray(nv, c, Z);
```

int **N_VWrmsNormVectorArray**(int *nv*, N_Vector* *X*, N_Vector* *W*, realtype* *m*)
This routine computes the weighted root mean square norm of each vector in a vector array:

$$m_j = \left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i})^2 \right)^{1/2}, \quad j = 0, \ldots, nv-1,$$

where x_j is a vector in the vector array *X*, w_j is a weight vector in the vector array *W*, and *m* is the output array of scalars containing the computed norms. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VWrmsNormVectorArray(nv, X, W, m);
```

int **N_VWrmsNormMaskVectorArray**(int *nv*, N_Vector* *X*, N_Vector* *W*, N_Vector *id*, realtype* *m*)
This routine computes the masked weighted root mean square norm of each vector in a vector array:

$$m_j = \left(\frac{1}{n} \sum_{i=0}^{n-1} (x_{j,i} w_{j,i} H(id_i))^2 \right)^{1/2}, \quad j = 0, \ldots, nv-1,$$

where $H(id_i) = 1$ for $id_i > 0$ and is zero otherwise, x_j is a vector in the vector array *X*, w_j is a weight vector in the vector array *W*, *id* is the mask vector, and *m* is the output array of scalars containing the computed norms. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VWrmsNormMaskVectorArray(nv, X, W, id, m);
```

int **N_VScaleAddMultiVectorArray**(int *nv*, int *nsum*, realtype* *c*, N_Vector* *X*, N_Vector** *YY*, N_Vector** *ZZ*)
This routine scales and adds a vector array of *nv* vectors to *nsum* other vector arrays:

$$z_{k,j,i} = c_k x_{j,i} + y_{k,j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1, \quad k = 0, \ldots, nsum-1$$

where *c* is an array of scalars, x_j is a vector in the vector array *X*, $y_{k,j}$ is a vector in the array of vector arrays *YY*, and $z_{k,j}$ is an output vector in the array of vector arrays *ZZ*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VScaleAddMultiVectorArray(nv, nsum, c, x, YY, ZZ);
```

int **N_VLinearCombinationVectorArray**(int *nv*, int *nsum*, realtype* *c*, N_Vector** *XX*, N_Vector* *Z*)
This routine computes the linear combination of *nsum* vector arrays containing *nv* vectors:

$$z_{j,i} = \sum_{k=0}^{nsum-1} c_k x_{k,j,i}, \quad i = 0, \ldots, n-1 \quad j = 0, \ldots, nv-1,$$

where *c* is an array of scalars, $x_{k,j}$ is a vector in array of vector arrays *XX*, and $z_{j,i}$ is an output vector in the vector array *Z*. If the output vector array is one of the vector arrays in *XX*, it *must* be the first vector array in *XX*. The operation returns 0 for success and a non-zero value otherwise.

Usage:

```
ier = N_VLinearCombinationVectorArray(nv, nsum, c, XX, Z);
```

10.2.3 Description of the NVECTOR local reduction operations

The following local reduction operations are also *optional*. As with the fused and vector array operations, these are intended to reduce parallel communication on distributed memory systems. If a particular NVECTOR implementation defines one of the local reduction operations as `NULL`, the NVECTOR interface will call one of the above standard vector operations as necessary. As above, for each operation, we give the name, usage of the function, and a description of its mathematical operations below.

realtype **N_VDotProdLocal** (N_Vector *x*, N_Vector *y*)

This routine computes the MPI task-local portion of the ordinary dot product of *x* and *y*:

$$d = \sum_{i=0}^{n_{local}-1} x_i y_i,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Usage:

```
d = N_VDotProdLocal(x, y);
```

realtype **N_VMaxNormLocal** (N_Vector *x*)

This routine computes the MPI task-local portion of the maximum norm of the NVECTOR *x*:

$$m = \max_{0 \le i < n_{local}} |x_i|,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Usage:

```
m = N_VMaxNormLocal(x);
```

realtype **N_VMinLocal** (N_Vector *x*)

This routine computes the smallest element of the MPI task-local portion of the NVECTOR *x*:

$$m = \min_{0 \le i < n_{local}} x_i,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Usage:

```
m = N_VMinLocal(x);
```

realtype **N_VL1NormLocal** (N_Vector *x*)

This routine computes the MPI task-local portion of the l_1 norm of the `N_Vector` *x*:

$$n = \sum_{i=0}^{n_{local}-1} |x_i|,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Usage:

```
n = N_VL1NormLocal(x);
```

realtype **N_VWSqrSumLocal** (N_Vector *x*, N_Vector *w*)

This routine computes the MPI task-local portion of the weighted squared sum of the NVECTOR *x* with weight vector *w*:

$$s = \sum_{i=0}^{n_{local}-1} (x_i w_i)^2,$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Usage:

```
s = N_VWSqrSumLocal(x, w);
```

realtype **N_VWSqrSumMaskLocal** (N_Vector *x*, N_Vector *w*, N_Vector *id*)

This routine computes the MPI task-local portion of the weighted squared sum of the NVECTOR *x* with weight vector *w* built using only the elements of *x* corresponding to positive elements of the NVECTOR *id*:

$$m = \sum_{i=0}^{n_{local}-1} (x_i w_i H(id_i))^2,$$

where

$$H(\alpha) = \begin{cases} 1 & \alpha > 0 \\ 0 & \alpha \leq 0 \end{cases}$$

and n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications).

Usage:

```
s = N_VWSqrSumMaskLocal(x, w, id);
```

booleantype **N_VInvTestLocal** (N_Vector *x*)

This routine sets the MPI task-local components of the NVECTOR *z* to be the inverses of the components of the NVECTOR *x*, with prior testing for zero values:

$$z_i = 1.0/x_i,\ i = 0, \ldots, n_{local} - 1$$

where n_{local} corresponds to the number of components in the vector on this MPI task (or $n_{local} = n$ for MPI-unaware applications). This routine returns a boolean assigned to `SUNTRUE` if all task-local components of *x* are nonzero (successful inversion) and returns `SUNFALSE` otherwise.

Usage:

```
t = N_VInvTestLocal(x);
```

booleantype **N_VConstrMaskLocal** (N_Vector *c*, N_Vector *x*, N_Vector *m*)

This routine performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$, and $x_i =$ anything if $c_i = 0$, for all MPI task-local components of the vectors. This routine returns a boolean assigned to `SUNFALSE` if any task-local element failed the constraint test and assigned to `SUNTRUE` if all passed. It also sets a mask vector *m*, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Usage:

```
t = N_VConstrMaskLocal(c, x, m);
```

realtype **N_VMinQuotientLocal**(N_Vector *num*, N_Vector *denom*)

This routine returns the minimum of the quotients obtained by term-wise dividing num_i by $denom_i$, for all MPI task-local components of the vectors. A zero element in *denom* will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

Usage:

```
minq = N_VMinQuotientLocal(num, denom);
```

10.2.4 Description of the NVECTOR exchange operations

The following vector exchange operations are also *optional* and are intended only for use when interfacing with the XBraid library for parallel-in-time integration. In that setting these operations are required but are otherwise unused by SUNDIALS packages and may be set to `NULL`. For each operation, we give the function signature, a description of the expected behavior, and an example of the function usage.

int **N_VBufSize**(N_Vector *x*, sunindextype **size*)

This routine returns the buffer size need to exchange in the data in the vector *x* between computational nodes.

Usage:

```
flag = N_VBufSize(x, &buf_size)
```

int **N_VBufPack**(N_Vector *x*, void **buf*)

This routine fills the exchange buffer *buf* with the vector data in *x*.

Usage:

```
flag = N_VBufPack(x, &buf)
```

int **N_VBufUnpack**(N_Vector *x*, void **buf*)

This routine unpacks the data in the exchange buffer *buf* into the vector *x*.

Usage:

```
flag = N_VBufUnpack(x, buf)
```

10.3 The NVECTOR_SERIAL Module

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of a `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of data.

```
struct _N_VectorContent_Serial {
   sunindextype length;
   booleantype own_data;
   realtype *data;
};
```

The header file to be included when using this module is `nvector_serial.h`. The installed module library to link to is `libsundials_nvecserial.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

10.3.1 NVECTOR_SERIAL accessor macros

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes the serial version.

NV_CONTENT_S(v)
This macro gives access to the contents of the serial vector `N_Vector` *v*.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` *content* structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

NV_OWN_DATA_S(v)
Access the *own_data* component of the serial `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
```

NV_DATA_S(v)
The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the *data* for the `N_Vector` `v`.

Similarly, the assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
```

NV_LENGTH_S(v)
Access the *length* component of the serial `N_Vector` *v*.

The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the *length* of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the *length* of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
```

NV_Ith_S(v, i)
This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`.

The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to $n-1$ for a vector of length n.

Implementation:

```
#define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
```

10.3.2 NVECTOR_SERIAL functions

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR*

vector array operations, and *Description of the NVECTOR local reduction operations*. Their names are obtained from those in those sections by appending the suffix `_Serial` (e.g. `N_VDestroy_Serial`). All the standard vector operations listed in the section *Description of the NVECTOR operations* with the suffix `_Serial` appended are callable via the Fortran 2003 interface by prepending an `F` (e.g. `FN_VDestroy_Serial`).

The module NVECTOR_SERIAL provides the following additional user-callable routines:

N_Vector **N_VNew_Serial** (sunindextype *vec_length*)
: This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

N_Vector **N_VNewEmpty_Serial** (sunindextype *vec_length*)
: This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Serial** (sunindextype *vec_length*, realtype* *v_data*)
: This function creates and allocates memory for a serial vector with user-provided data array, *v_data*.

  (This function does *not* allocate memory for `v_data` itself.)

N_Vector* **N_VCloneVectorArray_Serial** (int *count*, N_Vector *w*)
: This function creates (by cloning) an array of *count* serial vectors.

N_Vector* **N_VCloneVectorArrayEmpty_Serial** (int *count*, N_Vector *w*)
: This function creates (by cloning) an array of *count* serial vectors, each with an empty (` NULL) data array.

void **N_VDestroyVectorArray_Serial** (N_Vector* *vs*, int *count*)
: This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_Serial()* or with *N_VCloneVectorArrayEmpty_Serial()*.

void **N_VPrint_Serial** (N_Vector *v*)
: This function prints the content of a serial vector to `stdout`.

void **N_VPrintFile_Serial** (N_Vector *v*, FILE **outfile*)
: This function prints the content of a serial vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_SERIAL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_Serial()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_Serial()* will have the default settings for the NVECTOR_SERIAL module.

int **N_VEnableFusedOps_Serial** (N_Vector *v*, booleantype *tf*)
: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the serial vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_Serial** (N_Vector *v*, booleantype *tf*)
: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the serial vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_Serial** (N_Vector *v*, booleantype *tf*)
: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the serial vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableDotProdMulti_Serial** (N_Vector *v*, booleantype *tf*)
: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the serial vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearSumVectorArray_Serial** (N_Vector *v*, booleantype *tf*)
: This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the serial vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleVectorArray_Serial**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the serial vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_Serial**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the serial vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_Serial**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_Serial**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the serial vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMultiVectorArray_Serial**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the serial vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombinationVectorArray_Serial**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the serial vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_S(v) and then access v_data[i] within the loop than it is to use NV_Ith_S(v,i) within the loop.
- *N_VNewEmpty_Serial()*, *N_VMake_Serial()*, and *N_VCloneVectorArrayEmpty_Serial()* set the field *own_data* to SUNFALSE. The functions N_VDestroy_Serial() and *N_VDestroyVectorArray_Serial()* will not attempt to free the pointer data for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same length.

10.3.3 NVECTOR_SERIAL Fortran Interfaces

The NVECTOR_SERIAL module provides a Fortran 2003 module as well as Fortran 77 style interface functions for use from Fortran applications.

10.3.3.1 FORTRAN 2003 interface module

The fnvector_serial_mod Fortran module defines interfaces to all NVECTOR_SERIAL C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading F. For example, the function N_VNew_Serial is interfaced as FN_VNew_Serial.

The Fortran 2003 NVECTOR_SERIAL interface module can be accessed with the use statement, i.e. use fnvector_serial_mod, and linking to the library libsundials_fnvectorserial_mod.lib in addition to the C library. For details on where the library and module file fnvector_serial_mod.mod are installed

see the section *ARKode Installation Procedure*. We note that the module is accessible from the Fortran 2003 SUNDIALS integrators *without* separately linking to the `libsundials_fnvectorserial_mod` library.

10.3.3.2 FORTRAN 77 interface functions

For solvers that include a Fortran 77 interface module, the NVECTOR_SERIAL module also includes a Fortran-callable function `FNVINITS(code, NEQ, IER)`, to initialize this module. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKode); `NEQ` is the problem size (declared so as to match C type `long int`); and `IER` is an error return flag equal 0 for success and -1 for failure.

10.4 The NVECTOR_PARALLEL Module

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of a `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
   sunindextype local_length;
   sunindextype global_length;
   booleantype own_data;
   realtype *data;
   MPI_Comm comm;
};
```

The header file to be included when using this module is `nvector_parallel.h`. The installed module library to link to is `libsundials_nvecparallel.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

10.4.1 NVECTOR_PARALLEL accessor macros

The following seven macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix `_P` in the names denotes the distributed memory parallel version.

NV_CONTENT_P`(v)`
This macro gives access to the contents of the parallel `N_Vector` *v*.

The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` *content* structure of type `struct N_VectorContent_Parallel`.

Implementation:

```
#define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
```

NV_OWN_DATA_P`(v)`
Access the *own_data* component of the parallel `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_P(v)    ( NV_CONTENT_P(v)->own_data )
```

NV_DATA_P`(v)`
The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the *local_data* for the `N_Vector v`.

The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data` into *data*.

Implementation:

```
#define NV_DATA_P(v)        ( NV_CONTENT_P(v)->data )
```

NV_LOCLENGTH_P(v)
The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`.

The call `NV_LOCLENGTH_P(v) = llen_v` sets the *local_length* of `v` to be `llen_v`.

Implementation:

```
#define NV_LOCLENGTH_P(v)   ( NV_CONTENT_P(v)->local_length )
```

NV_GLOBLENGTH_P(v)
The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the *global_length* of the vector `v`.

The call `NV_GLOBLENGTH_P(v) = glen_v` sets the *global_length* of `v` to be `glen_v`.

Implementation:

```
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

NV_COMM_P(v)
This macro provides access to the MPI communicator used by the parallel `N_Vector` *v*.

Implementation:

```
#define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
```

NV_Ith_P(v, i)
This macro gives access to the individual components of the *local_data* array of an `N_Vector`.

The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`.

The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

Here `i` ranges from 0 to $n-1$, where n is the *local_length*.

Implementation:

```
#define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
```

10.4.2 NVECTOR_PARALLEL functions

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*. Their names are obtained from those in those sections by appending the suffix `_Parallel` (e.g. `N_VDestroy_Parallel`). The module NVECTOR_PARALLEL provides the following additional user-callable routines:

N_Vector **N_VNew_Parallel**(MPI_Comm *comm*, sunindextype *local_length*, sunindextype *global_length*)
This function creates and allocates memory for a parallel vector having global length *global_length*, having processor-local length *local_length*, and using the MPI communicator *comm*.

N_Vector **N_VNewEmpty_Parallel**(MPI_Comm *comm*, sunindextype *local_length*, sunindextype *global_length*)
This function creates a new parallel `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Parallel**(MPI_Comm *comm*, sunindextype *local_length*, sunindextype *global_length*, realtype* *v_data*)
This function creates and allocates memory for a parallel vector with user-provided data array.

(This function does *not* allocate memory for `v_data` itself.)

N_Vector* **N_VCloneVectorArray_Parallel**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* parallel vectors.

N_Vector* **N_VCloneVectorArrayEmpty_Parallel**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* parallel vectors, each with an empty (`NULL`) data array.

void **N_VDestroyVectorArray_Parallel**(N_Vector* *vs*, int *count*)
This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_Parallel()* or with *N_VCloneVectorArrayEmpty_Parallel()*.

sunindextype **N_VGetLocalLength_Parallel**(N_Vector *v*)
This function returns the local vector length.

void **N_VPrint_Parallel**(N_Vector *v*)
This function prints the local content of a parallel vector to `stdout`.

void **N_VPrintFile_Parallel**(N_Vector *v*, FILE **outfile*)
This function prints the local content of a parallel vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_PARALLEL module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_Parallel()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_Parallel()* will have the default settings for the NVECTOR_PARALLEL module.

int **N_VEnableFusedOps_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableDotProdMulti_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearSumVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableConstVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the parallel vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parallel vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMultiVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parallel vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombinationVectorArray_Parallel**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parallel vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the local component array via v_data = NV_DATA_P(v) and then access v_data[i] within the loop than it is to use NV_Ith_P(v,i) within the loop.
- *N_VNewEmpty_Parallel()*, *N_VMake_Parallel()*, and *N_VCloneVectorArrayEmpty_Parallel()* set the field *own_data* to SUNFALSE. The routines N_VDestroy_Parallel() and *N_VDestroyVectorArray_Parallel()* will not attempt to free the pointer data for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

10.4.3 NVECTOR_PARALLEL Fortran Interfaces

For solvers that include a Fortran interface module, the NVECTOR_PARALLEL module also includes a Fortran-callable function FNVINITP(COMM, code, NLOCAL, NGLOBAL, IER), to initialize this NVECTOR_PARALLEL module. Here COMM is the MPI communicator, code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKode); NLOCAL and NGLOBAL are the local and global vector sizes, respectively (declared so as to match C type long int); and IER is an error return flag equal 0 for success and -1 for failure.

Note: If the header file sundials_config.h defines SUNDIALS_MPI_COMM_F2C to be 1 (meaning the MPI implementation used to build SUNDIALS includes the MPI_Comm_f2c function), then COMM can be any valid MPI communicator. Otherwise, MPI_COMM_WORLD will be used, so just pass an integer value as a placeholder.

10.5 The NVECTOR_OPENMP Module

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of

length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP, the number of threads used is based on the supplied argument in the vector constructor.

```
struct _N_VectorContent_OpenMP {
  sunindextype length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to be included when using this module is `nvector_openmp.h`. The installed module library to link to is `libsundials_nvecopenmp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries. The Fortran module file to use when using the Fortran 2003 interface to this module is `fnvector_openmp_mod.mod`.

10.5.1 NVECTOR_OPENMP accessor macros

The following six macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix `_OMP` in the names denotes the OpenMP version.

NV_CONTENT_OMP(v)
: This macro gives access to the contents of the OpenMP vector `N_Vector` *v*.

  The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
  ```

NV_OWN_DATA_OMP(v)
: Access the *own_data* component of the OpenMP `N_Vector` *v*.

  Implementation:

  ```
  #define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )
  ```

NV_DATA_OMP(v)
: The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the *data* for the `N_Vector` `v`.

  Similarly, the assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  Implementation:

  ```
  #define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )
  ```

NV_LENGTH_OMP(v)
: Access the *length* component of the OpenMP `N_Vector` *v*.

  The assignment `v_len = NV_LENGTH_OMP(v)` sets `v_len` to be the *length* of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the *length* of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_OMP(v)  ( NV_CONTENT_OMP(v)->length )
```

NV_NUM_THREADS_OMP(v)
Access the *num_threads* component of the OpenMP `N_Vector` *v*.

The assignment `v_threads = NV_NUM_THREADS_OMP(v)` sets `v_threads` to be the *num_threads* of `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the *num_threads* of `v` to be `num_threads_v`.

Implementation:

```
#define NV_NUM_THREADS_OMP(v)  ( NV_CONTENT_OMP(v)->num_threads )
```

NV_Ith_OMP(v, i)
This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

The assignment `r = NV_Ith_OMP(v,i)` sets `r` to be the value of the `i`-th component of `v`.

The assignment `NV_Ith_OMP(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to $n-1$ for a vector of length n.

Implementation:

```
#define NV_Ith_OMP(v,i)  ( NV_DATA_OMP(v)[i] )
```

10.5.2 NVECTOR_OPENMP functions

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*. Their names are obtained from those in those sections by appending the suffix `_OpenMP` (e.g. `N_VDestroy_OpenMP`). All the standard vector operations listed in the section *Description of the NVECTOR operations* with the suffix `_OpenMP` appended are callable via the Fortran 2003 interface by prepending an *F' (e.g. ''FN_VDestroy_OpenMP'*).

The module NVECTOR_OPENMP provides the following additional user-callable routines:

N_Vector **N_VNew_OpenMP**(sunindextype *vec_length*, int *num_threads*)
This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

N_Vector **N_VNewEmpty_OpenMP**(sunindextype *vec_length*, int *num_threads*)
This function creates a new OpenMP `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_OpenMP**(sunindextype *vec_length*, realtype* *v_data*, int *num_threads*)
This function creates and allocates memory for a OpenMP vector with user-provided data array, *v_data*.

(This function does *not* allocate memory for `v_data` itself.)

N_Vector* **N_VCloneVectorArray_OpenMP**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* OpenMP vectors.

N_Vector* **N_VCloneVectorArrayEmpty_OpenMP**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* OpenMP vectors, each with an empty (` NULL) data array.

void **N_VDestroyVectorArray_OpenMP**(N_Vector* *vs*, int *count*)
This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_OpenMP()* or with *N_VCloneVectorArrayEmpty_OpenMP()*.

void **N_VPrint_OpenMP**(N_Vector *v*)
This function prints the content of an OpenMP vector to `stdout`.

void **N_VPrintFile_OpenMP**(N_Vector *v*, FILE **outfile*)
This function prints the content of an OpenMP vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_OPENMP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_OpenMP()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_OpenMP()* will have the default settings for the NVECTOR_OPENMP module.

int **N_VEnableFusedOps_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableDotProdMulti_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearSumVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableConstVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormMaskVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector arrays in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMultiVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombinationVectorArray_OpenMP**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the OpenMP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are

NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_OMP(v) and then access v_data[i] within the loop than it is to use NV_Ith_OMP(v,i) within the loop.
- *N_VNewEmpty_OpenMP()*, *N_VMake_OpenMP()*, and *N_VCloneVectorArrayEmpty_OpenMP()* set the field *own_data* to SUNFALSE. The functions N_VDestroy_OpenMP() and *N_VDestroyVectorArray_OpenMP()* will not attempt to free the pointer data for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_OPENMP implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

10.5.3 NVECTOR_OPENMP Fortran Interfaces

The NVECTOR_OPENMP module provides a Fortran 2003 module as well as Fortran 77 style interface functions for use from Fortran applications.

10.5.3.1 FORTRAN 2003 interface module

The fnvector_openmp_mod Fortran module defines interfaces to all NVECTOR_OPENMP C functions using the intrinsic iso_c_binding module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading F. For example, the function N_VNew_OpenMP is interfaced as FN_VNew_OpenMP.

The Fortran 2003 NVECTOR_OPENMP interface module can be accessed with the use statement, i.e. use fnvector_openmp_mod, and linking to the library libsundials_fnvectoropenmp_mod.lib in addition to the C library. For details on where the library and module file fnvector_openmp_mod.mod are installed see the section *ARKode Installation Procedure*.

10.5.3.2 FORTRAN 77 interface functions

For solvers that include a Fortran 77 interface module, the NVECTOR_OPENMP module also includes a Fortran-callable function FNVINITOMP(code, NEQ, NUMTHREADS, IER), to initialize this module. Here code is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKode); NEQ is the problem size (declared so as to match C type long int); NUMTHREADS is the number of threads; and IER is an error return flag equal 0 for success and -1 for failure.

10.6 The NVECTOR_PTHREADS Module

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, denoted NVECTOR_PTHREADS, defines the *content* field of N_Vector to be a structure containing the length of the vector, a pointer to the beginning of a

contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  sunindextype length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The header file to be included when using this module is `nvector_pthreads.h`. The installed module library to link to is `libsundials_nvecpthreads.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

10.6.1 NVECTOR_PTHREADS accessor macros

The following six macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix `_PT` in the names denotes the Pthreads version.

NV_CONTENT_PT(v)
This macro gives access to the contents of the Pthreads vector `N_Vector` *v*.

The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
```

NV_OWN_DATA_PT(v)
Access the *own_data* component of the Pthreads `N_Vector` *v*.

Implementation:

```
#define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )
```

NV_DATA_PT(v)
The assignment `v_data = NV_DATA_PT(v)` sets `v_data` to be a pointer to the first component of the *data* for the `N_Vector` `v`.

Similarly, the assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )
```

NV_LENGTH_PT(v)
Access the *length* component of the Pthreads `N_Vector` *v*.

The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the *length* of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the *length* of `v` to be `len_v`.

Implementation:

```
#define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )
```

NV_NUM_THREADS_PT(v)
Access the *num_threads* component of the Pthreads `N_Vector` *v*.

The assignment `v_threads = NV_NUM_THREADS_PT(v)` sets `v_threads` to be the *num_threads* of `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the *num_threads* of `v` to be `num_threads_v`.

Implementation:

```
#define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
```

NV_Ith_PT(v, i)
This macro gives access to the individual components of the *data* array of an `N_Vector`, using standard 0-based C indexing.

The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`.

The assignment `NV_Ith_PT(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

Here `i` ranges from 0 to $n-1$ for a vector of length n.

Implementation:

```
#define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
```

10.6.2 NVECTOR_PTHREADS functions

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*. Their names are obtained from those in those sections by appending the suffix `_Pthreads` (e.g. N_VDestroy_Pthreads). All the standard vector operations listed in the section *Description of the NVECTOR operations* are callable via the Fortran 2003 interface by prepending an *F' (e.g. ''FN_VDestroy_Pthreads'*). The module NVECTOR_PTHREADS provides the following additional user-callable routines:

N_Vector **N_VNew_Pthreads**(sunindextype *vec_length*, int *num_threads*)
This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

N_Vector **N_VNewEmpty_Pthreads**(sunindextype *vec_length*, int *num_threads*)
This function creates a new Pthreads `N_Vector` with an empty (`NULL`) data array.

N_Vector **N_VMake_Pthreads**(sunindextype *vec_length*, realtype* *v_data*, int *num_threads*)
This function creates and allocates memory for a Pthreads vector with user-provided data array, *v_data*.

(This function does *not* allocate memory for `v_data` itself.)

N_Vector* **N_VCloneVectorArray_Pthreads**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* Pthreads vectors.

N_Vector* **N_VCloneVectorArrayEmpty_Pthreads**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* Pthreads vectors, each with an empty (`NULL) data array.

void **N_VDestroyVectorArray_Pthreads**(N_Vector* *vs*, int *count*)
This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_Pthreads()* or with *N_VCloneVectorArrayEmpty_Pthreads()*.

void **N_VPrint_Pthreads**(N_Vector *v*)
This function prints the content of a Pthreads vector to `stdout`.

void **N_VPrintFile_Pthreads**(N_Vector *v*, FILE **outfile*)
This function prints the content of a Pthreads vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_PTHREADS module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_Pthreads()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_Pthreads()* will have the default settings for the NVECTOR_PTHREADS module.

int **N_VEnableFusedOps_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombination_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMulti_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMultiVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombinationVectorArray_Pthreads**(N_Vector *v*, booleantype *tf*)
: This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the Pthreads vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- When looping over the components of an N_Vector v, it is more efficient to first obtain the component array via v_data = NV_DATA_PT(v) and then access v_data[i] within the loop than it is to use

`NV_Ith_S(v,i)` within the loop.

- *N_VNewEmpty_Pthreads()*, *N_VMake_Pthreads()*, and *N_VCloneVectorArrayEmpty_Pthreads()* set the field *own_data* to `SUNFALSE`. The functions `N_VDestroy_Pthreads()` and *N_VDestroyVectorArray_Pthreads()* will not attempt to free the pointer data for any `N_Vector` with *own_data* set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the data pointer.
- To maximize efficiency, vector operations in the NVECTOR_PTHREADS implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

10.6.3 NVECTOR_PTHREADS Fortran Interfaces

The NVECTOR_PTHREADS module provides a Fortran 2003 module as well as Fortran 77 style interface functions for use from Fortran applications.

10.6.3.1 FORTRAN 2003 interface module

The `fnvector_pthreads_mod` Fortran module defines interfaces to all NVECTOR_PTHREADS C functions using the intrinsic `iso_c_binding` module which provides a standardized mechanism for interoperating with C. As noted in the C function descriptions above, the interface functions are named after the corresponding C function, but with a leading `F`. For example, the function `N_VNew_Pthreads` is interfaced as `FN_VNew_Pthreads`.

The Fortran 2003 NVECTOR_PTHREADS interface module can be accessed with the `use` statement, i.e. `use fnvector_pthreads_mod`, and linking to the library `libsundials_fnvectorpthreads_mod.lib` in addition to the C library. For details on where the library and module file `fnvector_pthreads_mod.mod` are installed see the section *ARKode Installation Procedure*.

10.6.3.2 FORTRAN 77 interface functions

For solvers that include a Fortran interface module, the NVECTOR_PTHREADS module slso includes a Fortran-callable function `FNVINITPTS(code, NEQ, NUMTHREADS, IER)`, to initialize this module. Here `code` is an input solver id (1 for CVODE, 2 for IDA, 3 for KINSOL, 4 for ARKode); `NEQ` is the problem size (declared so as to match C type `long int`); `NUMTHREADS` is the number of threads; and `IER` is an error return flag equal 0 for success and -1 for failure.

10.7 The NVECTOR_PARHYP Module

The NVECTOR_PARHYP implementation of the NVECTOR module provided with SUNDIALS is a wrapper around HYPRE's ParVector class. Most of the vector kernels simply call HYPRE vector operations. The implementation defines the *content* field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to an object of type `hypre_ParVector`, an MPI communicator, and a boolean flag *own_parvector* indicating ownership of the HYPRE parallel vector object *x*.

```
struct _N_VectorContent_ParHyp {
  sunindextype local_length;
  sunindextype global_length;
  booleantype own_data;
  booleantype own_parvector;
  realtype *data;
```

```
  MPI_Comm comm;
  hypre_ParVector *x;
};
```

The header file to be included when using this module is `nvector_parhyp.h`. The installed module library to link to is `libsundials_nvecparhyp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables. Note that NVECTOR_PARHYP requires SUNDIALS to be built with MPI support.

10.7.1 NVECTOR_PARHYP functions

The NVECTOR_PARHYP module defines implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for `N_VSetArrayPointer` and `N_VGetArrayPointer`, because accessing raw vector data is handled by low-level HYPRE functions. As such, this vector is not available for use with SUNDIALS Fortran interfaces. When access to raw vector data is needed, one should extract the HYPRE vector first, and then use HYPRE methods to access the data. Usage examples of NVECTOR_PARHYP are provided in the `cvAdvDiff_non_ph.c` example programs for CVODE and the `ark_diurnal_kry_ph.c` example program for ARKode.

The names of parhyp methods are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix `_ParHyp` (e.g. `N_VDestroy_ParHyp`). The module NVECTOR_PARHYP provides the following additional user-callable routines:

N_Vector **N_VNewEmpty_ParHyp**(MPI_Comm *comm*, sunindextype *local_length*, sunindextype *global_length*)
: This function creates a new parhyp `N_Vector` with the pointer to the HYPRE vector set to `NULL`.

N_Vector **N_VMake_ParHyp**(hypre_ParVector **x*)
: This function creates an `N_Vector` wrapper around an existing HYPRE parallel vector. It does *not* allocate memory for `x` itself.

hypre_ParVector ***N_VGetVector_ParHyp**(N_Vector *v*)
: This function returns a pointer to the underlying HYPRE vector.

N_Vector* **N_VCloneVectorArray_ParHyp**(int *count*, N_Vector *w*)
: This function creates (by cloning) an array of *count* parhyp vectors.

N_Vector* **N_VCloneVectorArrayEmpty_ParHyp**(int *count*, N_Vector *w*)
: This function creates (by cloning) an array of *count* parhyp vectors, each with an empty (`NULL) data array.

void **N_VDestroyVectorArray_ParHyp**(N_Vector* *vs*, int *count*)
: This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_ParHyp()* or with *N_VCloneVectorArrayEmpty_ParHyp()*.

void **N_VPrint_ParHyp**(N_Vector *v*)
: This function prints the local content of a parhyp vector to `stdout`.

void **N_VPrintFile_ParHyp**(N_Vector *v*, FILE **outfile*)
: This function prints the local content of a parhyp vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_PARHYP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VMake_ParHyp()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from

that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VMake_ParHyp()* will have the default settings for the NVECTOR_PARHYP module.

int **N_VEnableFusedOps_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombination_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMulti_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMultiVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombinationVectorArray_ParHyp**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the parhyp vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an N_Vector_ParHyp v, it is recommended to extract the HYPRE vector via x_vec = N_VGetVector_ParHyp(v) and then access components using appropriate HYPRE functions.
- *N_VNewEmpty_ParHyp()*, *N_VMake_ParHyp()*, and *N_VCloneVectorArrayEmpty_ParHyp()* set the field *own_parvector* to SUNFALSE. The functions N_VDestroy_ParHyp() and *N_VDestroyVectorArray_ParHyp()* will not attempt to delete an underlying HYPRE vector for any

N_Vector with *own_parvector* set to SUNFALSE. In such a case, it is the user's responsibility to delete the underlying vector.

- To maximize efficiency, vector operations in the NVECTOR_PARHYP implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

10.8 The NVECTOR_PETSC Module

The NVECTOR_PETSC module is an NVECTOR wrapper around the PETSc vector. It defines the *content* field of a N_Vector to be a structure containing the global and local lengths of the vector, a pointer to the PETSc vector, an MPI communicator, and a boolean flag *own_data* indicating ownership of the wrapped PETSc vector.

```
struct _N_VectorContent_Petsc {
   sunindextype local_length;
   sunindextype global_length;
   booleantype own_data;
   Vec *pvec;
   MPI_Comm comm;
};
```

The header file to be included when using this module is nvector_petsc.h. The installed module library to link to is libsundials_nvecpetsc.lib where .lib is typically .so for shared libraries and .a for static libraries.

Unlike native SUNDIALS vector types, NVECTOR_PETSC does not provide macros to access its member variables. Note that NVECTOR_PETSC requires SUNDIALS to be built with MPI support.

10.8.1 NVECTOR_PETSC functions

The NVECTOR_PETSC module defines implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for N_VGetArrayPointer and N_VSetArrayPointer. As such, this vector cannot be used with SUNDIALS Fortran interfaces. When access to raw vector data is needed, it is recommended to extract the PETSc vector first, and then use PETSc methods to access the data. Usage examples of NVECTOR_PETSC is provided in example programs for IDA.

The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffice _Petsc (e.g. N_VDestroy_Petsc). The module NVECTOR_PETSC provides the following additional user-callable routines:

N_Vector **N_VNewEmpty_Petsc**(MPI_Comm *comm*, sunindextype *local_length*, sunindextype *global_length*)
: This function creates a new PETSC N_Vector with the pointer to the wrapped PETSc vector set to NULL. It is used by the N_VMake_Petsc and N_VClone_Petsc implementations. It should be used only with great caution.

N_Vector **N_VMake_Petsc**(Vec* *pvec*)
: This function creates and allocates memory for an NVECTOR_PETSC wrapper with a user-provided PETSc vector. It does *not* allocate memory for the vector pvec itself.

Vec ***N_VGetVector_Petsc**(N_Vector *v*)
: This function returns a pointer to the underlying PETSc vector.

N_Vector* **N_VCloneVectorArray_Petsc**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* NVECTOR_PETSC vectors.

N_Vector* **N_VCloneVectorArrayEmpty_Petsc**(int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* NVECTOR_PETSC vectors, each with pointers to PETSc vectors set to `NULL`.

void **N_VDestroyVectorArray_Petsc**(N_Vector* *vs*, int *count*)
This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_Petsc()* or with *N_VCloneVectorArrayEmpty_Petsc()*.

void **N_VPrint_Petsc**(N_Vector *v*)
This function prints the global content of a wrapped PETSc vector to `stdout`.

void **N_VPrintFile_Petsc**(N_Vector *v*, const char *fname[]*)
This function prints the global content of a wrapped PETSc vector to `fname`.

By default all fused and vector array operations are disabled in the NVECTOR_PETSC module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VMake_Petsc()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VMake_Petsc()* will have the default settings for the NVECTOR_PETSC module.

int **N_VEnableFusedOps_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableDotProdMulti_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearSumVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableConstVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the PETSc vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormMaskVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector

arrays in the PETSc vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMultiVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the PETSc vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombinationVectorArray_Petsc**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the PETSc vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an N_Vector_Petsc v, it is recommeded to extract the PETSc vector via

  ```
  x_vec = N_VGetVector_Petsc(v);
  ```

  and then access components using appropriate PETSc functions.

- The functions *N_VNewEmpty_Petsc()*, *N_VMake_Petsc()*, and *N_VCloneVectorArrayEmpty_Petsc()* set the field *own_data* to SUNFALSE. The routines N_VDestroy_Petsc() and *N_VDestroyVectorArray_Petsc()* will not attempt to free the pointer pvec for any N_Vector with *own_data* set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the pvec pointer.

- To maximize efficiency, vector operations in the NVECTOR_PETSC implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

10.9 The NVECTOR_CUDA Module

The NVECTOR_CUDA module is an NVECTOR implementation in the CUDA language. The module allows for SUNDIALS vector kernels to run on NVIDIA GPU devices. It is intended for users who are already familiar with CUDA and GPU programming. Building this vector module requires a CUDA compiler and, by extension, a C++ compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Cuda
{
  sunindextype       length;
  booleantype        own_data;
  realtype*          host_data;
  realtype*          device_data;
  SUNCudaExecPolicy* stream_exec_policy;
  SUNCudaExecPolicy* reduce_exec_policy;
  void*              priv; /* 'private' data */
};

typedef struct _N_VectorContent_Cuda *N_VectorContent_Cuda;
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e. it is in charge of freeing the data), pointers to vector data on the host and the device, pointers to SUNCudaExecPolicy implementations that control how the CUDA kernels are launched for streaming and reduction vector kernels, and a private data structure which holds additonal members that should not be accessed directly.

When instantiated with `N_VNew_Cuda`, the underlying data will be allocated memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the `N_VMake_Cuda` constructor. To use CUDA managed memory, the constructors `N_VNewManaged_Cuda` and `N_VMakeManaged_Cuda` are provided. Details on each of these constructors are provided below.

To use the NVECTOR_CUDA module, include `nvector_cuda.h` and link to the library `libsundials_nveccuda.lib`. The extension, `.lib`, is typically `.so` for shared libraries and `.a` for static libraries.

10.9.1 NVECTOR_CUDA functions

Unlike other native SUNDIALS vector types, the NVECTOR_CUDA module does not provide macros to access its member variables. Instead, user should use the accessor functions:

realtype* **N_VGetHostArrayPointer_Cuda**(N_Vector *v*)
This function returns pointer to the vector data on the host.

realtype* **N_VGetDeviceArrayPointer_Cuda**(N_Vector *v*)
This function returns pointer to the vector data on the device.

booleantype **N_VIsManagedMemory_Cuda**(N_Vector *v*)
This function returns a boolean flag indiciating if the vector data array is in managed memory or not.

The NVECTOR_CUDA module defines implementations of all standard vector operations defined in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for `N_VSetArrayPointer`, and, if using unmanaged memory, `N_VGetArrayPointer`. As such, this vector can only be used with SUNDIALS Fortran interfaces, and the SUNDIALS direct solvers and preconditioners when using managed memory. The NVECTOR_CUDA module provides separate functions to access data on the host and on the device for the unmanaged memory use case. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR_CUDA are provided in example programs for CVODE *[HSR2017]*.

The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix `_Cuda` (e.g. `N_VDestroy_Cuda`). The module NVECTOR_CUDA provides the following additional user-callable routines:

N_Vector **N_VNew_Cuda**(sunindextype *length*)
This function creates and allocates memory for a CUDA `N_Vector`. The vector data array is allocated on both the host and device.

N_Vector **N_VNewManaged_Cuda**(sunindextype *vec_length*)
This function creates and allocates memory for a CUDA `N_Vector`. The vector data array is allocated in managed memory.

N_Vector **N_VNewEmpty_Cuda**(sunindextype *vec_length*)
This function creates a new `N_Vector` wrapper with the pointer to the wrapped CUDA vector set to `NULL`. It is used by *`N_VNew_Cuda()`*, *`N_VMake_Cuda()`*, and `N_VClone_Cuda()` implementations.

N_Vector **N_VMake_Cuda**(sunindextype *vec_length*, realtype **h_vdata*, realtype **d_vdata*)
This function creates a CUDA `N_Vector` with user-supplied vector data arrays for the host and the device.

N_Vector **N_VMakeManaged_Cuda**(sunindextype *vec_length*, realtype **vdata*)
This function creates a CUDA `N_Vector` with a user-supplied managed memory data array.

N_Vector **N_VMakeWithManagedAllocator_Cuda**(sunindextype *length*, void* (**allocfn*)(size_t size),
void (*freefn)(void* ptr))
This function creates a CUDA `N_Vector` with a user-supplied memory allocator. It requires the user to pro-

vide a corresponding free function as well. The memory allocated by the allocator function must behave like CUDA managed memory.

The module NVECTOR_CUDA also provides the following user-callable routines:

void N_VSetKernelExecPolicy_Cuda(N_Vector v,
SUNCudaExecPolicy* stream_exec_policy,
SUNCudaExecPolicy* reduce_exec_policy)

This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction CUDA kernels. By default the vector is setup to use the `SUNCudaThreadDirectExecPolicy` and `SUNCudaBlockReduceExecPolicy`. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the CUDA warp size (32). See section *The SUNCudaExecPolicy Class* below for more information about the `SUNCudaExecPolicy` class.

*Note: All vectors used in a single instance of a {sundials} solver must use the same execution policy. It is **strongly recommended* that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors*

void **N_VSetCudaStream_Cuda**(N_Vector *v*, cudaStream_t **stream*)

DEPRECATED This function will be removed in the next major release, user should utilize the `N_VSetKernelExecPolicy_Cuda` function instead.

This function sets the CUDA stream that all vector kernels will be launched on. By default an NVECTOR_CUDA uses the default CUDA stream.

*Note: All vectors used in a single instance of a {sundials} solver must use the same CUDA stream. It is **strongly recommended* that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors*

realtype* **N_VCopyToDevice_Cuda**(N_Vector *v*)

This function copies host vector data to the device.

realtype* **N_VCopyFromDevice_Cuda**(N_Vector *v*)

This function copies vector data from the device to the host.

void **N_VPrint_Cuda**(N_Vector *v*)

This function prints the content of a CUDA vector to `stdout`.

void **N_VPrintFile_Cuda**(N_Vector *v*, FILE **outfile*)

This function prints the content of a CUDA vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_CUDA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_Cuda()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_Cuda()* will have the default settings for the NVECTOR_CUDA module.

int **N_VEnableFusedOps_Cuda**(N_Vector *v*, booleantype *tf*)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the CUDA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_Cuda**(N_Vector *v*, booleantype *tf*)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the CUDA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_Cuda**(N_Vector *v*, booleantype *tf*)

This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors

fused operation in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMultiVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombinationVectorArray_Cuda**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the CUDA vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- When there is a need to access components of an N_Vector_Cuda, v, it is recommeded to use functions *N_VGetDeviceArrayPointer_Cuda()* or *N_VGetHostArrayPointer_Cuda()*. However, when using managed memory, the function *N_VGetArrayPointer()* may also be used.
- To maximize efficiency, vector operations in the NVECTOR_CUDA implementation that have more than one N_Vector argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same internal representations.

10.9.2 The SUNCudaExecPolicy Class

In order to provide maximum flexibility to users, the CUDA kernel execution parameters used by kernels within SUNDIALS are defined by objects of the sundials::CudaExecPolicy abstract class type (this class can be accessed in the global namespace as SUNCudaExecPolicy). Thus, users may provide custom execution policies that fit the needs of their problem. The sundials::CudaExecPolicy is defined in the header file sundials_cuda_policies.hpp, as follows:

```
class CudaExecPolicy
{
public:
   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
   virtual cudaStream_t stream() const = 0;
   virtual CudaExecPolicy* clone() const = 0;
   virtual ~CudaExecPolicy() {}
};
```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::CudaThreadDirectExecPolicy` (aka in the global namespace as `SUNCudaThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class CudaThreadDirectExecPolicy : public CudaExecPolicy
{
public:
   CudaThreadDirectExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)
      : blockDim_(blockDim), stream_(stream)
   {}

   CudaThreadDirectExecPolicy(const CudaThreadDirectExecPolicy& ex)
      : blockDim_(ex.blockDim_), stream_(ex.stream_)
   {}

   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
   {
      return (numWorkUnits + blockSize() - 1) / blockSize();
   }

   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
   {
      return blockDim_;
   }

   virtual cudaStream_t stream() const
   {
      return stream_;
   }

   virtual CudaExecPolicy* clone() const
   {
      return static_cast<CudaExecPolicy*>(new CudaThreadDirectExecPolicy(*this));
   }

private:
   const cudaStream_t stream_;
   const size_t blockDim_;
};
```

In total, SUNDIALS provides 3 execution policies:

1. `SUNCudaThreadDirectExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)` maps each CUDA thread to a work unit. The number of threads per block (blockDim) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a CUDA stream is provided, it will be used to execute the kernel.

2. `SUNCudaGridStrideExecPolicy(const size_t blockDim, const size_t gridDim, const cudaStream_t stream = 0)` is for kernels that use grid stride loops. The number of threads per block (blockDim) can be set to anything. The number of blocks (gridDim) can be set to anything. If a CUDA stream is provided, it will be used to execute the kernel.
3. `SUNCudaBlockReduceExecPolicy(const size_t blockDim, const cudaStream_t stream = 0)` is for kernels performing a reduction across indvidual thread blocks. The number of threads per block (blockDim) can be set to any valid multiple of the CUDA warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a CUDA stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
SUNCudaThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a `SUNMatrix` and an `N_Vector`) since they do not hold any modifiable state information.

10.10 The NVECTOR_HIP Module

The NVECTOR_HIP module is an NVECTOR implementation using the AMD ROCm HIP library. The module allows for SUNDIALS vector kernels to run on AMD or NVIDIA GPU devices. It is intended for users who are already familiar with HIP and GPU programming. Building this vector module requires the HIP-clang compiler. The vector content layout is as follows:

```
struct _N_VectorContent_Hip
{
  sunindextype       length;
  booleantype        own_data;
  SUNMemory          host_data;
  SUNMemory          device_data;
  SUNHipExecPolicy*  stream_exec_policy;
  SUNHipExecPolicy*  reduce_exec_policy;
  SUNMemoryHelper    mem_helper;
  void*              priv; /* 'private' data */
};

typedef struct _N_VectorContent_Hip *N_VectorContent_Hip;
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e. it is in charge of freeing the data), pointers to vector data on the host and the device, pointers to `SUNHipExecPolicy` implementations that control how the HIP kernels are launched for streaming and reduction vector kernels, and a private data structure which holds additonal members that should not be accessed directly.

When instantiated with *N_VNew_Hip()*, the underlying data will be allocated memory on both the host and the device. Alternatively, a user can provide host and device data arrays by using the *N_VMake_Hip()* constructor. To use HIP managed memory, the constructors *N_VNewManaged_Hip()* and *N_VMakeManaged_Hip()* are provided. Details on each of these constructors are provided below.

To use the NVECTOR_HIP module, include `nvector_hip.h` and link to the library `libsundials_nvechip.lib`. The extension, `.lib`, is typically `.so` for shared libraries and `.a` for static libraries.

10.10.1 NVECTOR_HIP functions

Unlike other native SUNDIALS vector types, the NVECTOR_HIP module does not provide macros to access its member variables. Instead, user should use the accessor functions:

realtype* **N_VGetHostArrayPointer_Hip**(N_Vector *v*)
This function returns pointer to the vector data on the host.

realtype* **N_VGetDeviceArrayPointer_Hip**(N_Vector *v*)
This function returns pointer to the vector data on the device.

booleantype **N_VIsManagedMemory_Hip**(N_Vector *v*)
This function returns a boolean flag indiciating if the vector data array is in managed memory or not.

The NVECTOR_HIP module defines implementations of all standard vector operations defined in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for `N_VSetArrayPointer()`. The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix `_Hip` (e.g. `N_VDestroy_Hip()`). The module NVECTOR_HIP provides the following additional user-callable routines:

N_Vector **N_VNew_Hip**(sunindextype *length*)
This function creates and allocates memory for a HIP `N_Vector`. The vector data array is allocated on both the host and device.

N_Vector **N_VNewManaged_Hip**(sunindextype *vec_length*)
This function creates and allocates memory for a HIP `N_Vector`. The vector data array is allocated in managed memory.

N_Vector **N_VNewEmpty_Hip**(sunindextype *vec_length*)
This function creates a new `N_Vector` wrapper with the pointer to the wrapped HIP vector set to `NULL`. It is used by `N_VNew_Hip()`, `N_VMake_Hip()`, and `N_VClone_Hip()` implementations.

N_Vector **N_VMake_Hip**(sunindextype *vec_length*, realtype **h_vdata*, realtype **d_vdata*)
This function creates a HIP `N_Vector` with user-supplied vector data arrays for the host and the device.

N_Vector **N_VMakeManaged_Hip**(sunindextype *vec_length*, realtype **vdata*)
This function creates a HIP `N_Vector` with a user-supplied managed memory data array.

The module NVECTOR_HIP also provides the following user-callable routines:

void **N_VSetKernelExecPolicy_Hip**(N_Vector *v*, SUNHipExecPolicy* *stream_exec_policy*, SUNHipExecPolicy* *reduce_exec_policy*)
This function sets the execution policies which control the kernel parameters utilized when launching the streaming and reduction HIP kernels. By default the vector is setup to use the `SUNHipThreadDirectExecPolicy` and `SUNHipBlockReduceExecPolicy`. Any custom execution policy for reductions must ensure that the grid dimensions (number of thread blocks) is a multiple of the HIP warp size (32 for NVIDIA GPUs, 64 for AMD GPUs). See section *The SUNHipExecPolicy Class* below for more information about the `SUNHipExecPolicy` class.

*Note: All vectors used in a single instance of a {sundials} solver must use the same execution policy. It is **strongly recommended** that this function is called immediately after constructing the vector, and any subsequent vector be created by cloning to ensure consistent execution policies across vectors*

realtype* **N_VCopyToDevice_Hip**(N_Vector *v*)
This function copies host vector data to the device.

realtype* **N_VCopyFromDevice_Hip**(N_Vector *v*)
This function copies vector data from the device to the host.

void **N_VPrint_Hip**(N_Vector *v*)
This function prints the content of a HIP vector to `stdout`.

void **N_VPrintFile_Hip**(N_Vector *v*, FILE **outfile*)
This function prints the content of a HIP vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_HIP module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_Hip()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_Hip()* will have the default settings for the NVECTOR_HIP module.

int **N_VEnableFusedOps_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableDotProdMulti_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearSumVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableConstVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the WRMS norm operation for vector arrays in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableWrmsNormMaskVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the masked WRMS norm operation for vector arrays in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMultiVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombinationVectorArray_Hip**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the HIP vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

Notes

- When there is a need to access components of an `N_Vector_Hip`, `v`, it is recommeded to use functions *`N_VGetDeviceArrayPointer_Hip()`* or *`N_VGetHostArrayPointer_Hip()`*. However, when using managed memory, the function *`N_VGetArrayPointer()`* may also be used.
- To maximize efficiency, vector operations in the NVECTOR_HIP implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

10.10.2 The `SUNHipExecPolicy` Class

In order to provide maximum flexibility to users, the HIP kernel execution parameters used by kernels within SUNDIALS are defined by objects of the `sundials::HipExecPolicy` abstract class type (this class can be accessed in the global namespace as `SUNHipExecPolicy`). Thus, users may provide custom execution policies that fit the needs of their problem. The `sundials::HipExecPolicy` is defined in the header file `sundials_hip_policies.hpp`, as follows:

```
class HipExecPolicy
{
public:
   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const = 0;
   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const = 0;
   virtual hipStream_t stream() const = 0;
   virtual HipExecPolicy* clone() const = 0;
   virtual ~HipExecPolicy() {}
};
```

To define a custom execution policy, a user simply needs to create a class that inherits from the abstract class and implements the methods. The SUNDIALS provided `sundials::HipThreadDirectExecPolicy` (aka in the global namespace as `SUNHipThreadDirectExecPolicy`) class is a good example of a what a custom execution policy may look like:

```
class HipThreadDirectExecPolicy : public HipExecPolicy
{
public:
   HipThreadDirectExecPolicy(const size_t blockDim, const hipStream_t stream = 0)
      : blockDim_(blockDim), stream_(stream)
   {}

   HipThreadDirectExecPolicy(const HipThreadDirectExecPolicy& ex)
      : blockDim_(ex.blockDim_), stream_(ex.stream_)
   {}

   virtual size_t gridSize(size_t numWorkUnits = 0, size_t blockDim = 0) const
   {
      return (numWorkUnits + blockSize() - 1) / blockSize();
   }

   virtual size_t blockSize(size_t numWorkUnits = 0, size_t gridDim = 0) const
   {
      return blockDim_;
   }

   virtual hipStream_t stream() const
   {
```

```
      return stream_;
   }

   virtual HipExecPolicy* clone() const
   {
      return static_cast<HipExecPolicy*>(new HipThreadDirectExecPolicy(*this));
   }

private:
   const hipStream_t stream_;
   const size_t blockDim_;
};
```

In total, SUNDIALS provides 3 execution policies:

1. `SUNHipThreadDirectExecPolicy(const size_t blockDim, const hipStream_t stream = 0)` maps each HIP thread to a work unit. The number of threads per block (blockDim) can be set to anything. The grid size will be calculated so that there are enough threads for one thread per element. If a HIP stream is provided, it will be used to execute the kernel.
2. `SUNHipGridStrideExecPolicy(const size_t blockDim, const size_t gridDim, const hipStream_t stream = 0)` is for kernels that use grid stride loops. The number of threads per block (blockDim) can be set to anything. The number of blocks (gridDim) can be set to anything. If a HIP stream is provided, it will be used to execute the kernel.
3. `SUNHipBlockReduceExecPolicy(const size_t blockDim, const hipStream_t stream = 0)` is for kernels performing a reduction across indvidual thread blocks. The number of threads per block (blockDim) can be set to any valid multiple of the HIP warp size. The grid size (gridDim) can be set to any value greater than 0. If it is set to 0, then the grid size will be chosen so that there is enough threads for one thread per work unit. If a HIP stream is provided, it will be used to execute the kernel.

For example, a policy that uses 128 threads per block and a user provided stream can be created like so:

```
hipStream_t stream;
hipStreamCreate(&stream);
SUNHipThreadDirectExecPolicy thread_direct(128, stream);
```

These default policy objects can be reused for multiple SUNDIALS data structures (e.g. a `SUNMatrix` and an `N_Vector`) since they do not hold any modifiable state information.

10.11 The NVECTOR_RAJA Module

The NVECTOR_RAJA module is an experimental {nvector} implementation using the RAJA hardware abstraction layer. In this implementation, RAJA allows for SUNDIALS vector kernels to run on AMD or NVIDIA GPU devices. The module is intended for users who are already familiar with RAJA and GPU programming. Building this vector module requires a C++11 compliant compiler and either the NVIDIA CUDA programming environment, or the AMD ROCm HIP programming environment. When using the AMD ROCm HIP environment, the HIP-clang compiler must be utilized. Users can select which backend (CUDA or HIP) to compile with by setting the `SUNDIALS_RAJA_BACKENDS` CMake variable to either CUDA or HIP. Besides the CUDA and HIP backends, RAJA has other backends such as serial, OpenMP, and OpenACC. These backends are not used in this SUNDIALS release.

The vector content layout is as follows:

```
struct _N_VectorContent_Raja
{
```

```
   sunindextype length;
   booleantype  own_data;
   realtype*    host_data;
   realtype*    device_data;
   void*        priv; /* 'private' data */
};
```

The content members are the vector length (size), a boolean flag that signals if the vector owns the data (i.e., it is in charge of freeing the data), pointers to vector data on the host and the device, and a private data structure which holds the memory management type, which should not be accessed directly.

When instantiated with *N_VNew_Raja()*, the underlying data will be allocated on both the host and the device. Alternatively, a user can provide host and device data arrays by using the *N_VMake_Raja()* constructor. To use managed memory, the constructors *N_VNewManaged_Raja()* and N_VMakeManaged_Raja() are provided. Details on each of these constructors are provided below.

The header file to include when using this is nvector_raja.h. The installed module library to link to is libsundials_nveccudaraja.lib when using the CUDA backend and libsundials_nvechipraja.lib when using the HIP backend. The extension .lib is typically .so for shared libraries .a for static libraries.

10.11.1 NVECTOR_RAJA functions

Unlike other native SUNDIALS vector types, the NVECTOR_RAJA module does not provide macros to access its member variables. Instead, user should use the accessor functions:

realtype* **N_VGetHostArrayPointer_Raja**(N_Vector *v*)
This function returns pointer to the vector data on the host.

realtype* **N_VGetDeviceArrayPointer_Raja**(N_Vector *v*)
This function returns pointer to the vector data on the device.

booleantype **N_VIsManagedMemory_Raja**(N_Vector *v*)
This function returns a boolean flag indicating if the vector data is allocated in managed memory or not.

The NVECTOR_RAJA module defines the implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for N_VDotProdMulti, N_VWrmsNormVectorArray, N_VWrmsNormMaskVectorArray as support for arrays of reduction vectors is not yet supported in RAJA. These functions will be added to the NVECTOR_RAJA implementation in the future. Additionally, the operations N_VGetArrayPointer and N_VSetArrayPointer are not implemented by the RAJA vector. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with SUNDIALS direct solvers and preconditioners. The NVECTOR_RAJA module provides separate functions to access data on the host and on the device. It also provides methods for copying from the host to the device and vice versa. Usage examples of NVECTOR_RAJA are provided in some example programs for CVODE *[HSR2017]*.

The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix _Raja (e.g. N_VDestroy_Raja). The module NVECTOR_RAJA provides the following additional user-callable routines:

N_Vector **N_VNew_Raja**(sunindextype *vec_length*)
This function creates and allocates memory for a RAJA N_Vector. The memory is allocated on both the host and the device. Its only argument is the vector length.

N_Vector **N_VNewManaged_Raja**(sunindextype *vec_length*)
This function creates and allocates memory for a RAJA N_Vector. The vector data array is allocated in managed memory.

N_Vector **N_VNewEmpty_Raja**(sunindextype *vec_length*)
This function creates a new `N_Vector` wrapper with the pointer to the wrapped RAJA vector set to `NULL`. It is used by *N_VNew_Raja()*, *N_VMake_Raja()*, and `N_VClone_Raja()` implementations.

N_Vector **N_VMake_Raja**(sunindextype *length*, realtype **vdata*)
This function creates an NVECTOR_RAJA with a user-supplied managed memory data array. This function does not allocate memory for data itself.

realtype* **N_VCopyToDevice_Raja**(N_Vector *v*)
This function copies host vector data to the device.

realtype* **N_VCopyFromDevice_Raja**(N_Vector *v*)
This function copies vector data from the device to the host.

void **N_VPrint_Raja**(N_Vector *v*)
This function prints the content of a RAJA vector to `stdout`.

void **N_VPrintFile_Raja**(N_Vector *v*, FILE **outfile*)
This function prints the content of a RAJA vector to `outfile`.

By default all fused and vector array operations are disabled in the NVECTOR_RAJA module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_Raja()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with *N_VNew_Raja()* will have the default settings for the NVECTOR_RAJA module.

int **N_VEnableFusedOps_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombination_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMulti_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearSumVectorArray_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleVectorArray_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableConstVectorArray_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the const operation for vector arrays in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableScaleAddMultiVectorArray_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector array to multiple vector arrays operation in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

int **N_VEnableLinearCombinationVectorArray_Raja**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination operation for vector arrays in the RAJA vector. The return value is `0` for success and `-1` if the input vector or its `ops` structure are `NULL`.

Notes

- When there is a need to access components of an `N_Vector_Raja`, `v`, it is recommended to use functions *N_VGetDeviceArrayPointer_Raja()* or *N_VGetHostArrayPointer_Raja()*. However, when using managed memory, the function `N_VGetArrayPointer` may also be used.
- To maximize efficiency, vector operations in the NVECTOR_RAJA implementation that have more than one `N_Vector` argument do not check for consistent internal representations of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

10.12 The NVECTOR_OPENMPDEV Module

In situations where a user has access to a device such as a GPU for offloading computation, SUNDIALS provides an NVECTOR implementation using OpenMP device offloading, called NVECTOR_OPENMPDEV.

The NVECTOR_OPENMPDEV implementation defines the *content* field of the `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguousdata array on the host, a pointer to the beginning of a contiguous data array on the device, and a boolean flag `own_data` which specifies the ownership of host and device data arrays.

```
struct _N_VectorContent_OpenMPDEV {
  sunindextype length;
  booleantype own_data;
  realtype *host_data;
  realtype *dev_data;
};
```

The header file to include when using this module is `nvector_openmpdev.h`. The installed module library to link to is `libsundials_nvecopenmpdev.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

10.12.1 NVECTOR_OPENMPDEV accessor macros

The following macros are provided to access the content of an NVECTOR_OPENMPDEV vector.

NV_CONTENT_OMPDEV(v)
This macro gives access to the contents of the NVECTOR_OPENMPDEV vector `N_Vector`.

The assignment `v_cont = NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the NVECTOR_OPENMPDEV content structure.

Implementation:

```
#define NV_CONTENT_OMPDEV(v) ( (N_VectorContent_OpenMPDEV)(v->content) )
```

NV_OWN_DATA_OMPDEV(v)
Access the *own_data* component of the OpenMPDEV `N_Vector` *v*.

The assignment `v_data = NV_DATA_HOST_OMPDEV(v)` sets `v_data` to be a pointer to the first component of the data on the host for the `N_Vector` `v`.

Implementation:

```
#define NV_OWN_DATA_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->own_data )
```

NV_DATA_HOST_OMPDEV(v)
The assignment `NV_DATA_HOST_OMPDEV(v) = v_data` sets the host component array of `v` to be `v_data` by storing the pointer `v_data`.

Implementation:

```
#define NV_DATA_HOST_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->host_data )
```

NV_DATA_DEV_OMPDEV(v)
The assignment `v_dev_data = NV_DATA_DEV_OMPDEV(v)` sets `v_dev_data` to be a pointer to the first component of the data on the device for the `N_Vector v`. The assignment `NV_DATA_DEV_OMPDEV(v) = v_dev_data` sets the device component array of `v` to be `v_dev_data` by storing the pointer `v_dev_data`.

Implementation:

```
#define NV_DATA_DEV_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->dev_data )
```

NV_LENGTH_OMPDEV
Access the *length* component of the OpenMPDEV `N_Vector` *v*.

The assignment `v_len = NV_LENGTH_OMPDEV(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMPDEV(v) = len_v` sets the length of `v` to be `len_v`.

```
#define NV_LENGTH_OMPDEV(v) ( NV_CONTENT_OMPDEV(v)->length )
```

10.12.2 NVECTOR_OPENMPDEV functions

The NVECTOR_OPENMPDEV module defines OpenMP device offloading implementations of all vector operations listed in Tables *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with the SUNDIALS FORTRAN interfaces, nor with the SUNDIALS direct solvers and preconditioners. It also provides methods for copying from the host to the device and vice versa.

The names of the vector operations are obtained from those in tables *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix `_OpenMPDEV` (e.g. `N_VDestroy_OpenMPDEV`). The module NVECTOR_OPENMPDEV provides the following additional user-callable routines:

N_Vector N_VNew_OpenMPDEV(sunindextype vec_length);
This function creates and allocates memory for an NVECTOR_OPENMPDEV `N_Vector`.

N_Vector N_VNewEmpty_OpenMPDEV(sunindextype vec_length);
This function creates a new NVECTOR_OPENMPDEV `N_Vector` with an empty (`NULL`) data array.

N_Vector N_VMake_OpenMPDEV(sunindextype vec_length, realtype *h_vdata, realtype *d_vdata);
This function creates an NVECTOR_OPENMPDEV vector with user-supplied vector data arrays `h_vdata} and ``d_vdata`. This function does not allocate memory for data itself.

N_Vector *N_VCloneVectorArray_OpenMPDEV(int count, N_Vector w);
This function creates (by cloning) an array of `count` NVECTOR_OPENMPDEV vectors.

N_Vector *N_VCloneVectorArrayEmpty_OpenMPDEV(int count, N_Vector w);
This function creates (by cloning) an array of `count` NVECTOR_OPENMPDEV vectors, each with an empty (`NULL`) data array.

void N_VDestroyVectorArray_OpenMPDEV(N_Vector *vs, int count);
This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMPDEV` or with `N_VCloneVectorArrayEmpty_OpenMPDEV`.

realtype *N_VGetHostArrayPointer_OpenMPDEV(N_Vector v);
This function returns a pointer to the host data array.

realtype *N_VGetDeviceArrayPointer_OpenMPDEV(N_Vector v);
This function returns a pointer to the device data array.

void N_VPrint_OpenMPDEV(N_Vector v);
This function prints the content of an NVECTOR_OPENMPDEV vector to `stdout`.

void N_VPrintFile_OpenMPDEV(N_Vector v, FILE *outfile);
This function prints the content of an NVECTOR_OPENMPDEV vector to `outfile`.

void N_VCopyToDevice_OpenMPDEV(N_Vector v);
This function copies the content of an NVECTOR_OPENMPDEV vector's host data array to the device data array.

void N_VCopyFromDevice_OpenMPDEV(N_Vector v);
This function copies the content of an NVECTOR_OPENMPDEV vector's device data array to the host data array.

By default all fused and vector array operations are disabled in the NVECTOR_OPENMPDEV module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with id{N_VNew_OpenMPDEV}, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using id{N_VClone}. This guarantees the new vectors will have the same operations enabled/disabled as cloned vectors inherit the same enable/disable options as the vector they are cloned from while vectors created with id{N_VNew_OpenMPDEV} will have the default settings for the NVECTOR_OPENMPDEV module.

int **N_VEnableFusedOps_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) all fused and vector array operations in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableLinearCombination_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear combination fused operation in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableScaleAddMulti_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale and add a vector to multiple vectors fused operation in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableDotProdMulti_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the multiple dot products fused operation in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableLinearSumVectorArray_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the linear sum operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableScaleVectorArray_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (`SUNTRUE`) or disables (`SUNFALSE`) the scale operation for vector arrays in the NVEC-

TOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableConstVectorArray_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableWrmsNormVectorArray_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

int **N_VEnableWrmsNormMaskVectorArray_OpenMPDEV**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

N_VEnableScaleAddMultiVectorArray_OpenMPDEV(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector array to multiple vector arrays operation in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

N_VEnableLinearCombinationVectorArray_OpenMPDEV(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination operation for vector arrays in the NVECTOR_OPENMPDEV vector. The return value is id{0} for success and id{-1} if the input vector or its id{ops} structure are id{NULL}.

Notes

- When looping over the components of an `N_Vector v`, it is most efficient to first obtain the component array via `h_data = NV_DATA_HOST_OMPDEV(v)` for the host arry or `v_data = NV_DATA_DEV_OMPDEV(v)` for the device array and then access `v_data[i]` within the loop.
- When accessing individual components of an `N_Vector v` on the host remember to first copy the array back from the device with `N_VCopyFromDevice_OpenMPDEV(v)` to ensure the array is up to date.
- `N_VNewEmpty_OpenMPDEV()`, `N_VMake_OpenMPDEV()`, and `N_VCloneVectorArrayEmpty_OpenMPDEV()` set the field *own_data* to `SUNFALSE`. The functions `N_VDestroy_OpenMPDEV()` and `N_VDestroyVectorArray_OpenMPDEV()` will not attempt to free the pointer data for any `N_Vector` with *own_data* set to `SUNFALSE`. In such a case, it is the user's responsibility to deallocate the data pointers.
- To maximize efficiency, vector operations in the NVECTOR_OPENMPDEV implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same length.

10.13 The NVECTOR_TRILINOS Module

The NVECTOR_TRILINOS module is an NVECTOR wrapper around the Trilinos Tpetra vector. The interface to Tpetra is implemented in the `Sundials::TpetraVectorInterface` class. This class simply stores a reference counting pointer to a Tpetra vector and inherits from an empty structure

```
struct _N_VectorContent_Trilinos {};
```

to interface the C++ class with the NVECTOR C code. A pointer to an instance of this class is kept in the *content* field of the `N_Vector` object, to ensure that the Tpetra vector is not deleted for as long as the `N_Vector` object exists.

The Tpetra vector type in the `Sundials::TpetraVectorInterface` class is defined as:

```
typedef Tpetra::Vector<realtype, int, sunindextype> vector_type;
```

The Tpetra vector will use the SUNDIALS-specified `realtype` as its scalar type, `int` as the local ordinal type, and `sunindextype` as the global ordinal type. This type definition will use Tpetra's default node type. Available Kokkos node types in Trilinos 12.14 release are serial (single thread), OpenMP, Pthread, and CUDA. The default node type is selected when building the Kokkos package. For example, the Tpetra vector will use a CUDA node if Tpetra was built with CUDA support and the CUDA node was selected as the default when Tpetra was built.

The header file to include when using this module is `nvector_trilinos.h`. The installed module library to link to is `libsundials_nvectrilinos.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

10.13.1 NVECTOR_TRILINOS functions

The NVECTOR_TRILINOS module defines implementations of all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with the SUNDIALS direct solvers and preconditioners. When access to raw vector data is needed, it is recommended to extract the Trilinos Tpetra vector first, and then use Tpetra vector methods to access the data. Usage examples of NVECTOR_TRILINOS are provided in example programs for IDA.

The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffice `_Trilinos` (e.g. `N_VDestroy_Trilinos`). Vector operations call existing `Tpetra::Vector` methods when available. Vector operations specific to SUNDIALS are implemented as standalone functions in the namespace `Sundials::TpetraVector`, located in the file `SundialsTpetraVectorKernels.hpp`. The module NVECTOR_TRILINOS provides the following additional user-callable routines:

Teuchos::RCP<vector_type> **`N_VGetVector_Trilinos`**(N_Vector *v*)
: This C++ function takes an `N_Vector` as the argument and returns a reference counting pointer to the underlying Tpetra vector. This is a standalone function defined in the global namespace.

N_Vector **`N_VMake_Trilinos`**(Teuchos::RCP<vector_type> *v*)
: This C++ function creates and allocates memory for an NVECTOR_TRILINOS wrapper around a user-provided Tpetra vector. This is a standalone function defined in the global namespace.

Notes

- The template parameter `vector_type` should be set as:

  ```
  typedef Sundials::TpetraVectorInterface::vector_type vector_type
  ```

  This will ensure that data types used in Tpetra vector match those in SUNDIALS.

- When there is a need to access components of an `N_Vector_Trilinos`, `v`, it is recommeded to extract the Trilinos vector object via `x_vec = N_VGetVector_Trilinos(v)` and then access components using the appropriate Trilinos functions.

- The functions `N_VDestroy_Trilinos` and `N_VDestroyVectorArray_Trilinos` only delete the `N_Vector` wrapper. The underlying Tpetra vector object will exist for as long as there is at least one reference to it.

10.14 The NVECTOR_MANYVECTOR Module

The NVECTOR_MANYVECTOR implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate problems with an inherent data partitioning for the solution vector within a computational node. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR_MANYVECTOR. We envision two generic use cases for this implementation:

1. *Heterogenous computational architectures*: for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one serial component based on NVECTOR_SERIAL, another component for GPU accelerators based on NVECTOR_CUDA, and another threaded component based on NVECTOR_OPENMP.
2. *Structure of arrays (SOA) data layouts*: for users who wish to create separate subvectors for each solution component, e.g., in a Navier-Stokes simulation they could have separate subvectors for density, velocities and pressure, which are combined together into a single NVECTOR_MANYVECTOR for the overall "solution".

We note that the above use cases are not mutually exclusive, and the NVECTOR_MANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR_MANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum *required* set of operations. Additionally, NVECTOR_MANYVECTOR sets no limit on the number of subvectors that may be attached (aside from the limitations of using `sunindextype` for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by NVECTOR_MANYVECTOR. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side, DAE residual, or nonlinear solver residual functions.

10.14.1 NVECTOR_MANYVECTOR structure

The NVECTOR_MANYVECTOR implementation defines the *content* field of `N_Vector` to be a structure containing the number of subvectors comprising the ManyVector, the global length of the ManyVector (including all subvectors), a pointer to the beginning of the array of subvectors, and a boolean flag `own_data` indicating ownership of the subvectors that populate `subvec_array`.

```
struct _N_VectorContent_ManyVector {
  sunindextype  num_subvectors;  /* number of vectors attached      */
  sunindextype  global_length;   /* overall manyvector length       */
  N_Vector*     subvec_array;    /* pointer to N_Vector array       */
  booleantype   own_data;        /* flag indicating data ownership  */
};
```

The header file to include when using this module is `nvector_manyvector.h`. The installed module library to link against is `libsundials_nvecmanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

10.14.2 NVECTOR_MANYVECTOR functions

The NVECTOR_MANYVECTOR module implements all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for `N_VGetArrayPointer()`, `N_VSetArrayPointer()`, `N_VScaleAddMultiVectorArray()`, and `N_VLinearCombinationVectorArray()`. As such, this vector cannot be used with the SUNDIALS Fortran-77 interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR_MANYVECTOR module provides functions to access subvectors, whose data may in turn be accessed according to their NVECTOR implementations.

The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix `_ManyVector` (e.g. `N_VDestroy_ManyVector`). The module NVECTOR_MANYVECTOR provides the following additional user-callable routines:

N_Vector **`N_VNew_ManyVector`**(sunindextype *num_subvectors*, N_Vector **vec_array*)
: This function creates a ManyVector from a set of existing NVECTOR objects.

  This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the ManyVector that contains them.

  Upon successful completion, the new ManyVector is returned; otherwise this routine returns `NULL` (e.g., a memory allocation failure occurred).

  Users of the Fortran 2003 interface to this function will first need to use the generic `N\_Vector` utility functions `N_VNewVectorArray`, and `N_VSetVecAtIndexVectorArray` to create the `N_Vector*` argument. This is further explained in Chapter *Working with N_Vector arrays*, and the functions are documented in Chapter *NVECTOR Utility Functions*.

N_Vector **`N_VGetSubvector_ManyVector`**(N_Vector *v*, sunindextype *vec_num*)
: This function returns the *vec_num* subvector from the NVECTOR array.

realtype ***`N_VGetSubvectorArrayPointer_ManyVector`**(N_Vector *v*, sunindextype *vec_num*)
: This function returns the data array pointer for the *vec_num* subvector from the NVECTOR array.

  If the input *vec_num* is invalid, or if the subvector does not support the `N_VGetArrayPointer` operation, then `NULL` is returned.

int **`N_VSetSubvectorArrayPointer_ManyVector`**(realtype **v_data*, N_Vector *v*, sunindextype *vec_num*)
: This function sets the data array pointer for the *vec_num* subvector from the NVECTOR array.

  If the input *vec_num* is invalid, or if the subvector does not support the `N_VSetArrayPointer` operation, then `-1` is returned; otherwise it returns `0`.

sunindextype **`N_VGetNumSubvectors_ManyVector`**(N_Vector *v*)
: This function returns the overall number of subvectors in the ManyVector object.

By default all fused and vector array operations are disabled in the NVECTOR_MANYVECTOR module, except for `N_VWrmsNormVectorArray()` and `N_VWrmsNormMaskVectorArray()`, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with `N_VNew_ManyVector()`, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using `N_VClone()`. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with `N_VNew_ManyVector()` will have the default settings for the NVEC-

TOR_MANYVECTOR module. We note that these routines *do not* call the corresponding routines on subvectors, so those should be set up as desired *before* attaching them to the ManyVector in *N_VNew_ManyVector()*.

int **N_VEnableFusedOps_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombination_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMulti_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_ManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the manyvector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- *N_VNew_ManyVector()* sets the field own_data = SUNFALSE. N_VDestroy_ManyVector() will not attempt to call *N_VDestroy()* on any subvectors contained in the subvector array for any N_Vector with own_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the subvectors.
- To maximize efficiency, arithmetic vector operations in the NVECTOR_MANYVECTOR implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same subvector representations.

10.15 The NVECTOR_MPIMANYVECTOR Module

The NVECTOR_MPIMANYVECTOR implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate problems with an inherent data partitioning for the solution vector, and when using distributed-memory parallel architectures. As such, the MPIManyVector implementation supports all use cases allowed by the

MPI-unaware NVECTOR_MANYVECTOR implementation, as well as partitioning data between nodes in a parallel environment. These data partitions are entirely user-defined, through construction of distinct NVECTOR modules for each component, that are then combined together to form the NVECTOR_MPIMANYVECTOR. We envision three generic use cases for this implementation:

1. *Heterogenous computational architectures (single-node or multi-node)*: for users who wish to partition data on a node between different computing resources, they may create architecture-specific subvectors for each partition. For example, a user could create one MPI-parallel component based on NVECTOR_PARALLEL, another single-node component for GPU accelerators based on NVECTOR_CUDA, and another threaded single-node component based on NVECTOR_OPENMP.
2. *Process-based multiphysics decompositions (multi-node)*: for users who wish to combine separate simulations together, e.g., where one subvector resides on one subset of MPI processes, while another subvector resides on a different subset of MPI processes, and where the user has created a MPI *intercommunicator* to connect these distinct process sets together.
3. *Structure of arrays (SOA) data layouts (single-node or multi-node)*: for users who wish to create separate subvectors for each solution component, e.g., in a Navier-Stokes simulation they could have separate subvectors for density, velocities and pressure, which are combined together into a single NVECTOR_MPIMANYVECTOR for the overall "solution".

We note that the above use cases are not mutually exclusive, and the NVECTOR_MPIMANYVECTOR implementation should support arbitrary combinations of these cases.

The NVECTOR_MPIMANYVECTOR implementation is designed to work with any NVECTOR subvectors that implement the minimum *required* set of operations, however significant performance benefits may be obtained when subvectors additionally implement the optional local reduction operations listed in the section *Description of the NVECTOR local reduction operations*.

Additionally, NVECTOR_MPIMANYVECTOR sets no limit on the number of subvectors that may be attached (aside from the limitations of using `sunindextype` for indexing, and standard per-node memory limitations). However, while this ostensibly supports subvectors with one entry each (i.e., one subvector for each solution entry), we anticipate that this extreme situation will hinder performance due to non-stride-one memory accesses and increased function call overhead. We therefore recommend a relatively coarse partitioning of the problem, although actual performance will likely be problem-dependent.

As a final note, in the coming years we plan to introduce additional algebraic solvers and time integration modules that will leverage the problem partitioning enabled by NVECTOR_MPIMANYVECTOR. However, even at present we anticipate that users will be able to leverage such data partitioning in their problem-defining ODE right-hand side, DAE residual, or nonlinear solver residual functions.

10.15.1 NVECTOR_MPIMANYVECTOR structure

The NVECTOR_MPIMANYVECTOR implementation defines the *content* field of `N_Vector` to be a structure containing the MPI communicator (or `MPI_COMM_NULL` if running on a single-node), the number of subvectors comprising the MPIManyVector, the global length of the MPIManyVector (including all subvectors on all MPI tasks), a pointer to the beginning of the array of subvectors, and a boolean flag `own_data` indicating ownership of the subvectors that populate `subvec_array`.

```
struct _N_VectorContent_MPIManyVector {
  MPI_Comm      comm;            /* overall MPI communicator        */
  sunindextype  num_subvectors;  /* number of vectors attached      */
  sunindextype  global_length;   /* overall mpimanyvector length    */
  N_Vector*     subvec_array;    /* pointer to N_Vector array       */
  booleantype   own_data;        /* flag indicating data ownership  */
};
```

The header file to include when using this module is `nvector_mpimanyvector.h`. The installed module library to link against is `libsundials_nvecmpimanyvector.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

Note: If SUNDIALS is configured with MPI disabled, then the MPIManyVector library will not be built. Furthermore, any user codes that include `nvector_mpimanyvector.h` *must* be compiled using an MPI-aware compiler (whether the specific user code utilizes MPI or not). We note that the NVECTOR_MANYVECTOR implementation is designed for ManyVector use cases in an MPI-unaware environment.

10.15.2 NVECTOR_MPIMANYVECTOR functions

The NVECTOR_MPIMANYVECTOR module implements all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, except for *N_VGetArrayPointer()*, *N_VSetArrayPointer()*, *N_VScaleAddMultiVectorArray()*, and *N_VLinearCombinationVectorArray()*. As such, this vector cannot be used with the SUNDIALS Fortran-77 interfaces, nor with the SUNDIALS direct solvers and preconditioners. Instead, the NVECTOR_MPIMANYVECTOR module provides functions to access subvectors, whose data may in turn be accessed according to their NVECTOR implementations.

The names of vector operations are obtained from those in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations* by appending the suffix `_MPIManyVector` (e.g. `N_VDestroy_MPIManyVector`). The module NVECTOR_MPIMANYVECTOR provides the following additional user-callable routines:

N_Vector **N_VNew_MPIManyVector**(sunindextype *num_subvectors*, N_Vector **vec_array*)

This function creates a MPIManyVector from a set of existing NVECTOR objects, under the requirement that all MPI-aware subvectors use the same MPI communicator (this is checked internally). If none of the subvectors are MPI-aware, then this may equivalently be used to describe data partitioning within a single node. We note that this routine is designed to support use cases A and C above.

This routine will copy all `N_Vector` pointers from the input `vec_array`, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns `NULL` (e.g., if two MPI-aware subvectors use different MPI communicators).

Users of the Fortran 2003 interface to this function will first need to use the generic `N\_Vector` utility functions `N_VNewVectorArray`, and `N_VSetVecAtIndexVectorArray` to create the `N_Vector*` argument. This is further explained in Chapter *Working with N_Vector arrays*, and the functions are documented in Chapter *NVECTOR Utility Functions*.

N_Vector **N_VMake_MPIManyVector**(MPI_Comm *comm*, sunindextype *num_subvectors*, N_Vector **vec_array*)

This function creates a MPIManyVector from a set of existing NVECTOR objects, and a user-created MPI communicator that "connects" these subvectors. Any MPI-aware subvectors may use different MPI communicators than the input *comm*. We note that this routine is designed to support any combination of the use cases above.

The input *comm* should be this user-created MPI communicator. This routine will internally call `MPI_Comm_dup` to create a copy of the input `comm`, so the user-supplied `comm` argument need not be retained after the call to *N_VMake_MPIManyVector()*.

If all subvectors are MPI-unaware, then the input *comm* argument should be `MPI_COMM_NULL`, although

in this case, it would be simpler to call *N_VNew_MPIManyVector()* instead, or to just use the NVECTOR_MANYVECTOR module.

This routine will copy all N_Vector pointers from the input *vec_array*, so the user may modify/free that pointer array after calling this function. However, this routine does *not* allocate any new subvectors, so the underlying NVECTOR objects themselves should not be destroyed before the MPIManyVector that contains them.

Upon successful completion, the new MPIManyVector is returned; otherwise this routine returns NULL (e.g., if the input *vec_array* is NULL).

N_Vector **N_VGetSubvector_MPIManyVector**(N_Vector *v*, sunindextype *vec_num*)
This function returns the *vec_num* subvector from the NVECTOR array.

realtype ***N_VGetSubvectorArrayPointer_MPIManyVector**(N_Vector *v*, sunindextype *vec_num*)
This function returns the data array pointer for the *vec_num* subvector from the NVECTOR array.

If the input *vec_num* is invalid, or if the subvector does not support the N_VGetArrayPointer operation, then NULL is returned.

int **N_VSetSubvectorArrayPointer_MPIManyVector**(realtype **v_data*, N_Vector *v*, sunindextype *vec_num*)
This function sets the data array pointer for the *vec_num* subvector from the NVECTOR array.

If the input *vec_num* is invalid, or if the subvector does not support the N_VSetArrayPointer operation, then −1 is returned; otherwise it returns 0.

sunindextype **N_VGetNumSubvectors_MPIManyVector**(N_Vector *v*)
This function returns the overall number of subvectors in the MPIManyVector object.

By default all fused and vector array operations are disabled in the NVECTOR_MPIMANYVECTOR module, except for *N_VWrmsNormVectorArray()* and *N_VWrmsNormMaskVectorArray()*, that are enabled by default. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector. To ensure consistency across vectors it is recommended to first create a vector with *N_VNew_MPIManyVector()* or *N_VMake_MPIManyVector()*, enable/disable the desired operations for that vector with the functions below, and create any additional vectors from that vector using *N_VClone()*. This guarantees that the new vectors will have the same operations enabled/disabled, since cloned vectors inherit those configuration options from the vector they are cloned from, while vectors created with *N_VNew_MPIManyVector()* and *N_VMake_MPIManyVector()* will have the default settings for the NVECTOR_MPIMANYVECTOR module. We note that these routines *do not* call the corresponding routines on subvectors, so those should be set up as desired *before* attaching them to the MPIManyVector in *N_VNew_MPIManyVector()* or *N_VMake_MPIManyVector()*.

int **N_VEnableFusedOps_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) all fused and vector array operations in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearCombination_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear combination fused operation in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleAddMulti_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale and add a vector to multiple vectors fused operation in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableDotProdMulti_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the multiple dot products fused operation in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableLinearSumVectorArray_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the linear sum operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableScaleVectorArray_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the scale operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableConstVectorArray_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the const operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormVectorArray_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the WRMS norm operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

int **N_VEnableWrmsNormMaskVectorArray_MPIManyVector**(N_Vector *v*, booleantype *tf*)
This function enables (SUNTRUE) or disables (SUNFALSE) the masked WRMS norm operation for vector arrays in the MPIManyVector vector. The return value is 0 for success and −1 if the input vector or its ops structure are NULL.

Notes

- *N_VNew_MPIManyVector()* and *N_VMake_MPIManyVector()* set the field own_data = SUNFALSE. N_VDestroy_MPIManyVector() will not attempt to call *N_VDestroy()* on any subvectors contained in the subvector array for any N_Vector with own_data set to SUNFALSE. In such a case, it is the user's responsibility to deallocate the subvectors.
- To maximize efficiency, arithmetic vector operations in the NVECTOR_MPIMANYVECTOR implementation that have more than one N_Vector argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with N_Vector arguments that were all created with the same subvector representations.

10.16 The NVECTOR_MPIPLUSX Module

The NVECTOR_MPIPLUSX implementation of the NVECTOR module provided with SUNDIALS is designed to facilitate the MPI+X paradigm, where X is some form of on-node (local) parallelism (e.g. OpenMP, CUDA). This paradigm is becoming increasingly popular with the rise of heterogeneous computing architectures.

The NVECTOR_MPIPLUSX implementation is designed to work with any NVECTOR that implements the minimum *required* set of operations. However, it is not recommended to use the NVECTOR_PARALLEL, NVECTOR_PARHYP, NVECTOR_PETSC, or NVECTOR_TRILINOS implementations underneath the NVECTOR_MPIPLUSX module since they already provide MPI capabilities.

10.16.1 NVECTOR_MPIPLUSX structure

The NVECTOR_MPIPLUSX implementation is a thin wrapper around the NVECTOR_MPIMANYVECTOR. Accordingly, it adopts the same content structure as defined in the section *NVECTOR_MPIMANYVECTOR structure*.

The header file to include when using this module is nvector_mpiplusx.h. The installed module library to link against is libsundials_nvecmpiplusx.lib where .lib is typically .so for shared libraries and .a for static libraries.

Note: If SUNDIALS is configured with MPI disabled, then the mpiplusx library will not be built. Furthermore, any user codes that include `nvector_mpiplusx.h` *must* be compiled using an MPI-aware compiler.

10.16.2 NVECTOR_MPIPLUSX functions

The NVECTOR_MPIPLUSX module adopts all vector operations listed in the sections *Description of the NVECTOR operations*, *Description of the NVECTOR fused operations*, *Description of the NVECTOR vector array operations*, and *Description of the NVECTOR local reduction operations*, from the NVECTOR_MPIMANYVECTOR (see section *The NVECTOR_MPIMANYVECTOR Module*) except for *N_VGetArrayPointer()*, and *N_VSetArrayPointer()*; the module provides its own implementation of these functions that call the local vector implementations. Therefore, the NVECTOR_MPIPLUSX module implements all of the operations listed in the referenced sections except for *N_VScaleAddMultiVectorArray()*, and *N_VLinearCombinationVectorArray()*. Accordingly, it's compatibility with the SUNDIALS Fortran-77 interface, and with the SUNDIALS direct solvers and preconditioners depends on the local vector implementation.

The module NVECTOR_MPIPLUSX provides the following additional user-callable routines:

N_Vector **N_VMake_MPIPlusX**(MPI_Comm *comm*, N_Vector **local_vector*)
: This function creates a MPIPlusX vector from an exisiting local (i.e. on node) NVECTOR object, and a user-created MPI communicator.

  The input *comm* should be this user-created MPI communicator. This routine will internally call `MPI_Comm_dup` to create a copy of the input `comm`, so the user-supplied `comm` argument need not be retained after the call to *N_VMake_MPIPlusX()*.

  This routine will copy the NVECTOR pointer to the input `local_vector`, so the underlying local NVECTOR object should not be destroyed before the mpiplusx that contains it.

  Upon successful completion, the new MPIPlusX is returned; otherwise this routine returns `NULL` (e.g., if the input *local_vector* is `NULL`).

N_Vector **N_VGetLocal_MPIPlusX**(N_Vector *v*)
: This function returns the local vector underneath the MPIPlusX NVECTOR.

realtype ***N_VGetArrayPointer_MPIPlusX**(N_Vector *v*)
: This function returns the data array pointer for the local vector.

  If the local vector does not support the `N_VGetArrayPointer` operation, then `NULL` is returned.

void **N_VSetArrayPointer_MPIPlusX**(realtype **v_data*, N_Vector *v*)
: This function sets the data array pointer for the local vector if the local vector implements the `N_VGetArrayPointe()` operation.

The NVECTOR_MPIPLUSX module does not implement any fused or vector array operations. Instead users should enable/disable fused operations on the local vector.

Notes

- *N_VMake_MPIPlusX()* sets the field `own_data = SUNFALSE` and `N_VDestroy_MPIPlusX()` will not call *N_VDestroy()* on the local vector. In this a case, it is the user's responsibility to deallocate the local vector.
- To maximize efficiency, arithmetic vector operations in the NVECTOR_MPIPLUSX implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same subvector representations.

10.17 NVECTOR Examples

There are NVECTOR examples that may be installed for each implementation: serial, parallel, OpenMP, and Pthreads. Each implementation makes use of the functions in `test_nvector.c`. These example functions show simple usage of the NVECTOR family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.

The following is a list of the example functions in `test_nvector.c`:

- `Test_N_VClone`: Creates clone of vector and checks validity of clone.
- `Test_N_VCloneEmpty`: Creates clone of empty vector and checks validity of clone.
- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VGetLength`: Compares self-reported length to calculated length.
- `Test_N_VGetCommunicator`: Compares self-reported communicator to the one used in constructor; or for MPI-unaware vectors it ensures that NULL is reported.
- `Test_N_VLinearSum` Case 1a: Test y = x + y
- `Test_N_VLinearSum` Case 1b: Test y = -x + y
- `Test_N_VLinearSum` Case 1c: Test y = ax + y
- `Test_N_VLinearSum` Case 2a: Test x = x + y
- `Test_N_VLinearSum` Case 2b: Test x = x - y
- `Test_N_VLinearSum` Case 2c: Test x = x + by
- `Test_N_VLinearSum` Case 3: Test z = x + y
- `Test_N_VLinearSum` Case 4a: Test z = x - y
- `Test_N_VLinearSum` Case 4b: Test z = -x + y
- `Test_N_VLinearSum` Case 5a: Test z = x + by
- `Test_N_VLinearSum` Case 5b: Test z = ax + y
- `Test_N_VLinearSum` Case 6a: Test z = -x + by
- `Test_N_VLinearSum` Case 6b: Test z = ax - y
- `Test_N_VLinearSum` Case 7: Test z = a(x + y)
- `Test_N_VLinearSum` Case 8: Test z = a(x - y)
- `Test_N_VLinearSum` Case 9: Test z = ax + by
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: z = x * y
- `Test_N_VDiv`: Test vector division: z = x / y
- `Test_N_VScale`: Case 1: scale: x = cx
- `Test_N_VScale`: Case 2: copy: z = x
- `Test_N_VScale`: Case 3: negate: z = -x

- `Test_N_VScale`: Case 4: combination: z = cx
- `Test_N_VAbs`: Create absolute value of vector.
- `Test_N_VAddConst`: add constant vector: z = c + x
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate the max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate the weighted root mean square.
- `Test_N_VWrmsNormMask`: Create vector of known values, find and validate the weighted root mean square using all elements except one.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test z[i] = 1 / x[i]
- `Test_N_VConstrMask`: Test mask of vector x with vector c.
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.
- `Test_N_VLinearCombination`: Case 1a: Test x = a x
- `Test_N_VLinearCombination`: Case 1b: Test z = a x
- `Test_N_VLinearCombination`: Case 2a: Test x = a x + b y
- `Test_N_VLinearCombination`: Case 2b: Test z = a x + b y
- `Test_N_VLinearCombination`: Case 3a: Test x = x + a y + b z
- `Test_N_VLinearCombination`: Case 3b: Test x = a x + b y + c z
- `Test_N_VLinearCombination`: Case 3c: Test w = a x + b y + c z
- `Test_N_VScaleAddMulti`: Case 1a: y = a x + y
- `Test_N_VScaleAddMulti`: Case 1b: z = a x + y
- `Test_N_VScaleAddMulti`: Case 2a: Y[i] = c[i] x + Y[i], i = 1,2,3
- `Test_N_VScaleAddMulti`: Case 2b: Z[i] = c[i] x + Y[i], i = 1,2,3
- `Test_N_VDotProdMulti`: Case 1: Calculate the dot product of two vectors
- `Test_N_VDotProdMulti`: Case 2: Calculate the dot product of one vector with three other vectors in a vector array.
- `Test_N_VLinearSumVectorArray`: Case 1: z = a x + b y
- `Test_N_VLinearSumVectorArray`: Case 2a: Z[i] = a X[i] + b Y[i]
- `Test_N_VLinearSumVectorArray`: Case 2b: X[i] = a X[i] + b Y[i]
- `Test_N_VLinearSumVectorArray`: Case 2c: Y[i] = a X[i] + b Y[i]
- `Test_N_VScaleVectorArray`: Case 1a: y = c y
- `Test_N_VScaleVectorArray`: Case 1b: z = c y
- `Test_N_VScaleVectorArray`: Case 2a: Y[i] = c[i] Y[i]
- `Test_N_VScaleVectorArray`: Case 2b: Z[i] = c[i] Y[i]

- `Test_N_VScaleVectorArray`: Case 1a: z = c
- `Test_N_VScaleVectorArray`: Case 1b: Z[i] = c
- `Test_N_VWrmsNormVectorArray`: Case 1a: Create a vector of know values, find and validate the weighted root mean square norm.
- `Test_N_VWrmsNormVectorArray`: Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each.
- `Test_N_VWrmsNormMaskVectorArray`: Case 1a: Create a vector of know values, find and validate the weighted root mean square norm using all elements except one.
- `Test_N_VWrmsNormMaskVectorArray`: Case 1b: Create a vector array of three vectors of know values, find and validate the weighted root mean square norm of each using all elements except one.
- `Test_N_VScaleAddMultiVectorArray`: Case 1a: y = a x + y
- `Test_N_VScaleAddMultiVectorArray`: Case 1b: z = a x + y
- `Test_N_VScaleAddMultiVectorArray`: Case 2a: Y[j][0] = a[j] X[0] + Y[j][0]
- `Test_N_VScaleAddMultiVectorArray`: Case 2b: Z[j][0] = a[j] X[0] + Y[j][0]
- `Test_N_VScaleAddMultiVectorArray`: Case 3a: Y[0][i] = a[0] X[i] + Y[0][i]
- `Test_N_VScaleAddMultiVectorArray`: Case 3b: Z[0][i] = a[0] X[i] + Y[0][i]
- `Test_N_VScaleAddMultiVectorArray`: Case 4a: Y[j][i] = a[j] X[i] + Y[j][i]
- `Test_N_VScaleAddMultiVectorArray`: Case 4b: Z[j][i] = a[j] X[i] + Y[j][i]
- `Test_N_VLinearCombinationVectorArray`: Case 1a: x = a x
- `Test_N_VLinearCombinationVectorArray`: Case 1b: z = a x
- `Test_N_VLinearCombinationVectorArray`: Case 2a: x = a x + b y
- `Test_N_VLinearCombinationVectorArray`: Case 2b: z = a x + b y
- `Test_N_VLinearCombinationVectorArray`: Case 3a: x = a x + b y + c z
- `Test_N_VLinearCombinationVectorArray`: Case 3b: w = a x + b y + c z
- `Test_N_VLinearCombinationVectorArray`: Case 4a: X[0][i] = c[0] X[0][i]
- `Test_N_VLinearCombinationVectorArray`: Case 4b: Z[i] = c[0] X[0][i]
- `Test_N_VLinearCombinationVectorArray`: Case 5a: X[0][i] = c[0] X[0][i] + c[1] X[1][i]
- `Test_N_VLinearCombinationVectorArray`: Case 5b: Z[i] = c[0] X[0][i] + c[1] X[1][i]
- `Test_N_VLinearCombinationVectorArray`: Case 6a: X[0][i] = X[0][i] + c[1] X[1][i] + c[2] X[2][i]
- `Test_N_VLinearCombinationVectorArray`: Case 6b: X[0][i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]
- `Test_N_VLinearCombinationVectorArray`: Case 6c: Z[i] = c[0] X[0][i] + c[1] X[1][i] + c[2] X[2][i]
- `Test_N_VDotProdLocal`: Calculate MPI task-local portion of the dot product of two vectors.
- `Test_N_VMaxNormLocal`: Create vector with known values, find and validate the MPI task-local portion of the max norm.
- `Test_N_VMinLocal`: Create vector, find and validate the MPI task-local min.
- `Test_N_VL1NormLocal`: Create vector, find and validate the MPI task-local portion of the L1 norm.

- `Test_N_VWSqrSumLocal`: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors.
- `Test_N_VWSqrSumMaskLocal`: Create vector of known values, find and validate the MPI task-local portion of the weighted squared sum of two vectors, using all elements except one.
- `Test_N_VInvTestLocal`: Test the MPI task-local portion of z[i] = 1 / x[i]
- `Test_N_VConstrMaskLocal`: Test the MPI task-local portion of the mask of vector x with vector c.
- `Test_N_VMinQuotientLocal`: Fill two vectors with known values. Calculate and validate the MPI task-local minimum quotient.

10.18 NVECTOR functions required by ARKode

In the table below, we list the vector functions in the `N_Vector` module that are called within the ARKode package. The table also shows, for each function, which ARKode module uses the function. The ARKSTEP and ERKSTEP columns show function usage within the main time-stepping modules and the shared ARKode infrastructure, while the remaining columns show function usage within the ARKLS linear solver interface, the ARKBANDPRE and ARKBBDPRE preconditioner modules, and the FARKODE module.

Note that since FARKODE is built on top of ARKode, and therefore requires the same `N_Vector` routines, in the FARKODE column we only list the routines that the FARKODE interface directly utilizes.

Note that for ARKLS we only list the `N_Vector` routines used directly by ARKLS, each `SUNLinearSolver` module may have additional requirements that are not listed here. In addition, specific `SUNNonlinearSolver` modules attached to ARKode may have additional `N_Vector` requirements. For additional requirements by specific `SUNLinearSolver` and `SUNNonlinearSolver` modules, please see the accompanying sections *Description of the SUNLinearSolver module* and *Description of the SUNNonlinearSolver Module*.

At this point, we should emphasize that the user does not need to know anything about ARKode's usage of vector functions in order to use ARKode. Instead, this information is provided primarily for users interested in constructing a custom `N_Vector` module. We note that a number of `N_Vector` functions from the section *Description of the NVECTOR Modules* are not listed in the above table. Therefore a user-supplied `N_Vector` module for ARKode could safely omit these functions from their implementation (although some may be needed by `SUNNonlinearSolver` or `SUNLinearSolver` modules).

Routine	ARK-STEP	ERK-STEP	ARKLS	ARKBAND-PRE	ARKBBD-PRE	FARKODE
N_VGetLength			X			
N_VAbs	X	X				
N_VAddConst	X	X				
N_VClone	X	X	X			
N_VCloneEmpty						X
N_VConst	X	X	X			X
N_VDestroy	X	X	X			X
N_VDiv	X	X				
N_VGetArrayPointer			X[1]	X	X	X
N_VInv	X	X				
N_VLinearSum	X	X	X			
N_VMaxNorm	X	X				
N_VMin	X	X				X
N_VScale	X	X	X	X	X	
N_VSetArrayPointer			X[1]			X
N_VSpace[2]	X	X	X	X	X	
N_VWrmsNorm	X	X	X	X	X	
N_VLinearCombination[3]	X	X				
N_VMinQuotient[5]	X	X				
N_VConstrMask[5]	X	X				
N_VCompare[5]	X	X				

1. This is only required with dense or band matrix-based linear solver modules, where the default difference-quotient Jacobian approximation is used.

2. The *N_VSpace()* function is only informational, and will only be called if provided by the N_Vector implementation.

3. The *N_VLinearCombination()* function is in fact optional; if it is not supplied then *N_VLinearSum()* will be used instead.

4. The *N_VGetLength()* function is only required when an iterative or matrix iterative SUNLinearSolver module is used.

5. The functions *N_VMinQuotient()*, *N_VConstrMask()*, and *N_VCompare()* are only used when inequality constraints are enabled and may be omitted if this feature is not used.

Chapter 11

Matrix Data Structures

The SUNDIALS library comes packaged with a variety of SUNMatrix implementations, designed for simulations requiring direct linear solvers for problems in serial or shared-memory parallel environments. SUNDIALS additionally provides a simple interface for generic matrices (akin to a C++ *abstract base class*). All of the major SUNDIALS packages (CVODE(s), IDA(s), KINSOL, ARKODE), are constructed to only depend on these generic matrix operations, making them immediately extensible to new user-defined matrix objects. For each of the SUNDIALS-provided matrix types, SUNDIALS also provides at least two SUNLinearSolver implementations that factor these matrix objects and use them in the solution of linear systems.

11.1 Description of the SUNMATRIX Modules

For problems that involve direct methods for solving linear systems, the SUNDIALS solvers not only operate on generic vectors, but also on generic matrices (of type SUNMatrix), through a set of operations defined by the particular SUNMATRIX implementation. Users can provide their own specific implementation of the SUNMATRIX module, particularly in cases where they provide their own N_Vector and/or linear solver modules, and require matrices that are compatible with those implementations. Alternately, we provide three SUNMATRIX implementations: dense, banded, and sparse. The generic operations are described below, and descriptions of the implementations provided with SUNDIALS follow.

The generic SUNMatrix type has been modeled after the object-oriented style of the generic N_Vector type. Specifically, a generic SUNMatrix is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the matrix, and an *ops* field pointing to a structure with generic matrix operations. The type SUNMatrix is defined as:

```
typedef struct _generic_SUNMatrix *SUNMatrix;

struct _generic_SUNMatrix {
    void *content;
    struct _generic_SUNMatrix_Ops *ops;
};
```

Here, the _generic_SUNMatrix_Ops structure is essentially a list of function pointers to the various actual matrix operations, and is defined as

```
struct _generic_SUNMatrix_Ops {
  SUNMatrix_ID (*getid)(SUNMatrix);
  SUNMatrix    (*clone)(SUNMatrix);
  void         (*destroy)(SUNMatrix);
  int          (*zero)(SUNMatrix);
```

```
  int           (*copy)(SUNMatrix, SUNMatrix);
  int           (*scaleadd)(realtype, SUNMatrix, SUNMatrix);
  int           (*scaleaddi)(realtype, SUNMatrix);
  int           (*matvecsetup)(SUNMatrix);
  int           (*matvec)(SUNMatrix, N_Vector, N_Vector);
  int           (*space)(SUNMatrix, long int*, long int*);
};
```

The generic SUNMATRIX module defines and implements the matrix operations acting on a `SUNMatrix`. These routines are nothing but wrappers for the matrix operations defined by a particular SUNMATRIX implementation, which are accessed through the *ops* field of the `SUNMatrix` structure. To illustrate this point we show below the implementation of a typical matrix operation from the generic SUNMATRIX module, namely `SUNMatZero`, which sets all values of a matrix `A` to zero, returning a flag denoting a successful/failed operation:

```
int SUNMatZero(SUNMatrix A)
{
  return((int) A->ops->zero(A));
}
```

The subsection *Description of the SUNMATRIX operations* contains a complete list of all matrix operations defined by the generic SUNMATRIX module. A particular implementation of the SUNMATRIX module must:

- Specify the *content* field of the `SUNMatrix` object.
- Define and implement a minimal subset of the matrix operations. See the documentation for each SUNDIALS solver to determine which SUNMATRIX operations they require. The list of required operations for use with ARKode is given in the section *SUNMATRIX functions required by ARKode*.

  Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMATRIX module (each with different `SUNMatrix` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a `SUNMatrix` with the new *content* field and with *ops* pointing to the new matrix operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNMatrix` (e.g., a routine to print the *content* for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the content field of the newly defined `SUNMatrix`.

To aid in the creation of custom SUNMATRIX modules the generic SUNMATRIX module provides three utility functions *`SUNMatNewEmpty()`*, *`SUNMatCopyOps()`*, and *`SUNMatFreeEmpty()`*. When used in custom SUNMATRIX constructors and clone routines these functions will ease the introduction of any new optional matrix operations to the SUNMATRIX API by ensuring only required operations need to be set and all operations are copied when cloning a matrix.

SUNMatrix **`SUNMatNewEmpty`**`()`
: This function allocates a new generic `SUNMatrix` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

  Return value: If successful, this function returns a `SUNMatrix` object. If an error occurs when allocating the object, then this routine will return `NULL`.

int **`SUNMatCopyOps`**`(`SUNMatrix *A*, SUNMatrix *B*`)`
: This function copies the function pointers in the `ops` structure of `A` into the `ops` structure of `B`.

  Arguments:

  - *A* – the matrix to copy operations from.

- *B* – the matrix to copy operations to.

Return value: If successful, this function returns `0`. If either of the inputs are `NULL` or the `ops` structure of either input is `NULL`, then is function returns a non-zero value.

void **`SUNMatFreeEmpty`**(SUNMatrix *A*)

This routine frees the generic `SUNMatrix` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is `NULL`, and, if it is not, it will free it as well.

Arguments:

- *A* – a SUNMatrix object

Each SUNMATRIX implementation included in SUNDIALS has a unique identifier specified in enumeration and shown in the table below. It is recommended that a user-supplied SUNMATRIX implementation use the `SUNMATRIX_CUSTOM` identifier.

11.1.1 Identifiers associated with matrix kernels supplied with SUNDIALS

Matrix ID	Matrix type	ID Value
SUNMATRIX_DENSE	Dense $M \times N$ matrix	0
SUNMATRIX_BAND	Band $M \times M$ matrix	1
SUNMATRIX_SPARSE	Sparse (CSR or CSC) $M \times N$ matrix	2
SUNMATRIX_SLUNRLOC	SUNMatrix wrapper for SuperLU_DIST SuperMatrix	3
SUNMATRIX_CUSTOM	User-provided custom matrix	4

11.2 Description of the SUNMATRIX operations

For each of the `SUNMatrix` operations, we give the name, usage of the function, and a description of its mathematical operations below.

SUNMatrix_ID **`SUNMatGetID`**(SUNMatrix *A*)

Returns the type identifier for the matrix *A*. It is used to determine the matrix implementation type (e.g. dense, banded, sparse,...) from the abstract `SUNMatrix` interface. This is used to assess compatibility with SUNDIALS-provided linear solver implementations. Returned values are given in the Table *Identifiers associated with matrix kernels supplied with SUNDIALS*

Usage:

```
id = SUNMatGetID(A);
```

SUNMatrix **`SUNMatClone`**(SUNMatrix *A*)

Creates a new `SUNMatrix` of the same type as an existing matrix *A* and sets the *ops* field. It does not copy the matrix, but rather allocates storage for the new matrix.

Usage:

```
B = SUNMatClone(A);
```

void **`SUNMatDestroy`**(SUNMatrix *A*)

Destroys the `SUNMatrix` *A* and frees memory allocated for its internal data.

Usage:

```
SUNMatDestroy(A);
```

int **SUNMatSpace**(SUNMatrix *A*, long int **lrw*, long int **liw*)
Returns the storage requirements for the matrix *A*. *lrw* contains the number of realtype words and *liw* contains the number of integer words. The return value denotes success/failure of the operation.

This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied `SUNMatrix` module if that information is not of interest.

Usage:

```
ier = SUNMatSpace(A, &lrw, &liw);
```

int **SUNMatZero**(SUNMatrix *A*)
Zeros all entries of the `SUNMatrix` *A*. The return value is an integer flag denoting success/failure of the operation:

$$A_{i,j} = 0, \quad i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Usage:

```
ier = SUNMatZero(A);
```

int **SUNMatCopy**(SUNMatrix *A*, SUNMatrix *B*)
Performs the operation *B = A* for all entries of the matrices *A* and *B*. The return value is an integer flag denoting success/failure of the operation:

$$B_{i,j} = A_{i,j}, \quad i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Usage:

```
ier = SUNMatCopy(A,B);
```

SUNMatScaleAdd(realtype *c*, SUNMatrix *A*, SUNMatrix *B*)
Performs the operation *A = cA + B*. The return value is an integer flag denoting success/failure of the operation:

$$A_{i,j} = cA_{i,j} + B_{i,j}, \quad i = 1, \ldots, m, \ j = 1, \ldots, n.$$

Usage:

```
ier = SUNMatScaleAdd(c, A, B);
```

SUNMatScaleAddI(realtype *c*, SUNMatrix *A*)
Performs the operation *A = cA + I*. The return value is an integer flag denoting success/failure of the operation:

$$A_{i,j} = cA_{i,j} + \delta_{i,j}, \quad i, j = 1, \ldots, n.$$

Usage:

```
ier = SUNMatScaleAddI(c, A);
```

SUNMatMatvecSetup(SUNMatrix *A*)
Performs any setup necessary to perform a matrix-vector product. The return value is an integer flag denoting success/failure of the operation. It is useful for SUNMatrix implementations which need to prepare the matrix itself, or communication structures before performing the matrix-vector product.

Usage:

```
ier = SUNMatMatvecSetup(A);
```

SUNMatMatvec(SUNMatrix *A*, N_Vector *x*, N_Vector *y*)
Performs the matrix-vector product *y* = *Ax*. It should only be called with vectors *x* and *y* that are compatible with the matrix *A* – both in storage type and dimensions. The return value is an integer flag denoting success/failure of the operation:

$$y_i = \sum_{j=1}^{n} A_{i,j} x_j, \quad i = 1, \ldots, m.$$

Usage:

```
ier = SUNMatMatvec(A, x, y);
```

11.2.1 SUNMatrix return codes

The functions provided to SUNMatrix modules within the SUNDIALS-provided SUNMatrix implementations utilize a common set of return codes, listed below. These adhere to a common pattern: 0 indicates success, a negative value indicates a failure. Aside from this pattern, the actual values of each error code are primarily to provide additional information to the user in case of a SUNMatrix failure.

- `SUNMAT_SUCCESS` (0) – successful call
- `SUNMAT_ILL_INPUT` (-1) – an illegal input has been provided to the function
- `SUNMAT_MEM_FAIL` (-2) – failed memory access or allocation
- `SUNMAT_OPERATION_FAIL` (-3) – a SUNMatrix operation returned nonzero
- `SUNMAT_MATVEC_SETUP_REQUIRED` (-4) – the `SUNMatMatvecSetup` routine needs to be called prior to calling `SUNMatMatvec`

11.3 Compatibility of SUNMATRIX types

We note that not all `SUNMatrix` types are compatible with all `N_Vector` types provided with SUNDIALS. This is primarily due to the need for compatibility within the `SUNMatMatvec` routine; however, compatibility between `SUNMatrix` and `N_Vector` implementations is more crucial when considering their interaction within `SUNLinearSolver` objects, as will be described in more detail in section *Description of the SUNLinearSolver module*. More specifically, in the Table *SUNDIALS matrix interfaces and vector implementations that can be used for each* we show the matrix interfaces available as `SUNMatrix` modules, and the compatible vector implementations.

11.3.1 SUNDIALS matrix interfaces and vector implementations that can be used for each

Linear Solver	Serial	Parallel (MPI)	OpenMP	pThreads	*hypre* Vec.	PETSc Vec.	CUDA	RAJA	User Suppl.
Dense	X		X	X					X
Band	X		X	X					X
Sparse	X		X	X					X
SLUNRloc	X	X	X	X	X	X			X
User supplied	X	X	X	X	X	X	X	X	X

11.4 The SUNMATRIX_DENSE Module

The dense implementation of the SUNMatrix module provided with SUNDIALS, SUNMATRIX_DENSE, defines the *content* field of SUNMatrix to be the following structure:

```
struct _SUNMatrixContent_Dense {
  sunindextype M;
  sunindextype N;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
```

These entries of the *content* field contain the following information:

- M - number of rows
- N - number of columns
- data - pointer to a contiguous block of realtype variables. The elements of the dense matrix are stored columnwise, i.e. the $A_{i,j}$ element of a dense SUNMatrix A (with $0 \leq i < M$ and $0 \leq j < N$) may be accessed via data[j*M+i].
- ldata - length of the data array ($= M \cdot N$).
- cols - array of pointers. cols[j] points to the first element of the j-th column of the matrix in the array data. The $A_{i,j}$ element of a dense SUNMatrix A (with $0 \leq i < M$ and $0 \leq j < N$) may be accessed may be accessed via cols[j][i].

The header file to be included when using this module is sunmatrix/sunmatrix_dense.h.

The following macros are provided to access the content of a SUNMATRIX_DENSE matrix. The prefix SM_ in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix _D denotes that these are specific to the *dense* version.

SM_CONTENT_D(A)
: This macro gives access to the contents of the dense SUNMatrix *A*.

  The assignment A_cont = SM_CONTENT_D(A) sets A_cont to be a pointer to the dense SUNMatrix content structure.

  Implementation:

  ```
  #define SM_CONTENT_D(A)    ( (SUNMatrixContent_Dense)(A->content) )
  ```

SM_ROWS_D(A)
: Access the number of rows in the dense SUNMatrix *A*.

  This may be used either to retrieve or to set the value. For example, the assignment A_rows = SM_ROWS_D(A) sets A_rows to be the number of rows in the matrix A. Similarly, the assignment SM_ROWS_D(A) = A_rows sets the number of columns in A to equal A_rows.

  Implementation:

  ```
  #define SM_ROWS_D(A)    ( SM_CONTENT_D(A)->M )
  ```

SM_COLUMNS_D(A)
: Access the number of columns in the dense SUNMatrix *A*.

This may be used either to retrieve or to set the value. For example, the assignment `A_columns = SM_COLUMNS_D(A)` sets `A_columns` to be the number of columns in the matrix `A`. Similarly, the assignment `SM_COLUMNS_D(A) = A_columns` sets the number of columns in `A` to equal `A_columns`

Implementation:

```
#define SM_COLUMNS_D(A)    ( SM_CONTENT_D(A)->N )
```

SM_LDATA_D(A)
Access the total data length in the dense `SUNMatrix` A.

This may be used either to retrieve or to set the value. For example, the assignment `A_ldata = SM_LDATA_D(A)` sets `A_ldata` to be the length of the data array in the matrix `A`. Similarly, the assignment `SM_LDATA_D(A) = A_ldata` sets the parameter for the length of the data array in `A` to equal `A_ldata`.

Implementation:

```
#define SM_LDATA_D(A)    ( SM_CONTENT_D(A)->ldata )
```

SM_DATA_D(A)
This macro gives access to the `data` pointer for the matrix entries.

The assignment `A_data = SM_DATA_D(A)` sets `A_data` to be a pointer to the first component of the data array for the dense `SUNMatrix A`. The assignment `SM_DATA_D(A) = A_data` sets the data array of `A` to be `A_data` by storing the pointer `A_data`.

Implementation:

```
#define SM_DATA_D(A)    ( SM_CONTENT_D(A)->data )
```

SM_COLS_D(A)
This macro gives access to the `cols` pointer for the matrix entries.

The assignment `A_cols = SM_COLS_D(A)` sets `A_cols` to be a pointer to the array of column pointers for the dense `SUNMatrix A`. The assignment `SM_COLS_D(A) = A_cols` sets the column pointer array of `A` to be `A_cols` by storing the pointer `A_cols`.

Implementation:

```
#define SM_COLS_D(A)    ( SM_CONTENT_D(A)->cols )
```

SM_COLUMN_D(A)
This macros gives access to the individual columns of the data array of a dense `SUNMatrix`.

The assignment `col_j = SM_COLUMN_D(A,j)` sets `col_j` to be a pointer to the first entry of the `j`-th column of the $M \times N$ dense matrix `A` (with $0 \leq j < N$). The type of the expression `SM_COLUMN_D(A,j)` is `realtype *`. The pointer returned by the call `SM_COLUMN_D(A,j)` can be treated as an array which is indexed from 0 to `M-1`.

Implementation:

```
#define SM_COLUMN_D(A,j)    ( (SM_CONTENT_D(A)->cols)[j] )
```

SM_ELEMENT_D(A)
This macro gives access to the individual entries of the data array of a dense `SUNMatrix`.

The assignments `SM_ELEMENT_D(A,i,j) = a_ij` and `a_ij = SM_ELEMENT_D(A,i,j)` reference the $A_{i,j}$ element of the $M \times N$ dense matrix `A` (with $0 \leq i < M$ and $0 \leq j < N$).

Implementation:

```
#define SM_ELEMENT_D(A,i,j)  ( (SM_CONTENT_D(A)->cols)[j][i] )
```

The SUNMATRIX_DENSE module defines dense implementations of all matrix operations listed in the section *Description of the SUNMATRIX operations*. Their names are obtained from those in that section by appending the suffix `_Dense` (e.g. `SUNMatCopy_Dense`). The module SUNMATRIX_DENSE provides the following additional user-callable routines:

SUNMatrix **SUNDenseMatrix**(sunindextype *M*, sunindextype *N*)
: This constructor function creates and allocates memory for a dense `SUNMatrix`. Its arguments are the number of rows, `M`, and columns, `N`, for the dense matrix.

void **SUNDenseMatrix_Print**(SUNMatrix *A*, FILE* *outfile*)
: This function prints the content of a dense `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

sunindextype **SUNDenseMatrix_Rows**(SUNMatrix *A*)
: This function returns the number of rows in the dense `SUNMatrix`.

sunindextype **SUNDenseMatrix_Columns**(SUNMatrix *A*)
: This function returns the number of columns in the dense `SUNMatrix`.

sunindextype **SUNDenseMatrix_LData**(SUNMatrix *A*)
: This function returns the length of the data array for the dense `SUNMatrix`.

realtype* **SUNDenseMatrix_Data**(SUNMatrix *A*)
: This function returns a pointer to the data array for the dense `SUNMatrix`.

realtype** **SUNDenseMatrix_Cols**(SUNMatrix *A*)
: This function returns a pointer to the cols array for the dense `SUNMatrix`.

realtype* **SUNDenseMatrix_Column**(SUNMatrix *A*, sunindextype *j*)
: This function returns a pointer to the first entry of the jth column of the dense `SUNMatrix`. The resulting pointer should be indexed over the range `0` to `M-1`.

Notes

- When looping over the components of a dense `SUNMatrix A`, the most efficient approaches are to:
  - First obtain the component array via `A_data = SM_DATA_D(A)` or `A_data = SUNDenseMatrix_Data(A)` and then access `A_data[i]` within the loop.
  - First obtain the array of column pointers via `A_cols = SM_COLS_D(A)` or `A_cols = SUNDenseMatrix_Cols(A)`, and then access `A_cols[j][i]` within the loop.
  - Within a loop over the columns, access the column pointer via `A_colj = SUNDenseMatrix_Column(A,j)` and then to access the entries within that column using `A_colj[i]` within the loop.

  All three of these are more efficient than using `SM_ELEMENT_D(A,i,j)` within a double loop.
- Within the `SUNMatMatvec_Dense` routine, internal consistency checks are performed to ensure that the matrix is called with consistent `N_Vector` implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the SUNMATRIX_DENSE module also includes the Fortran-callable function *FSUNDenseMatInit()* to initialize this SUNMATRIX_DENSE module for a given SUNDIALS solver.

subroutine FSUNDenseMatInit(*CODE*, *M*, *N*, *IER*)
: Initializes a dense `SUNMatrix` structure for use in a SUNDIALS solver.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *M* (`long int`, input) – number of matrix rows.
- *N* (`long int`, input) – number of matrix columns.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *`FSUNDenseMassMatInit()`* initializes this SUNMATRIX_DENSE module for storing the mass matrix.

subroutine FSUNDenseMassMatInit(*M*, *N*, *IER*)
Initializes a dense `SUNMatrix` structure for use as a mass matrix in ARKode.

Arguments:

- *M* (`long int`, input) – number of matrix rows.
- *N* (`long int`, input) – number of matrix columns.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

11.5 The SUNMATRIX_BAND Module

The banded implementation of the `SUNMatrix` module provided with SUNDIALS, SUNMATRIX_BAND, defines the *content* field of `SUNMatrix` to be the following structure:

```
struct _SUNMatrixContent_Band {
  sunindextype M;
  sunindextype N;
  sunindextype mu;
  sunindextype ml;
  sunindextype smu;
  sunindextype ldim;
  realtype *data;
  sunindextype ldata;
  realtype **cols;
};
```

A diagram of the underlying data representation in a banded matrix is shown in Figure *SUNBandMatrix Diagram*. A more complete description of the parts of this *content* field is given below:

- `M` - number of rows
- `N` - number of columns (`N` = `M`)
- `mu` - upper half-bandwidth, $0 \leq$ mu $< N$
- `ml` - lower half-bandwidth, $0 \leq$ ml $< N$
- `smu` - storage upper bandwidth, mu $\leq$ smu $< N$. The LU decomposition routines in the associated SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND modules write the LU factors into the existing storage for the band matrix. The upper triangular factor U, however, may have an upper bandwidth as big as `min(N-1, mu+ml)` because of partial pivoting. The `smu` field holds the upper half-bandwidth allocated for the band matrix.
- `ldim` - leading dimension (ldim $\geq smu + ml + 1$)

- `data` - pointer to a contiguous block of `realtype` variables. The elements of the banded matrix are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. `data` is a pointer to `ldata` contiguous locations which hold the elements within the banded matrix.
- `ldata` - length of the data array ($=$ ldim $\cdot N$)
- `cols` - array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from `smu-mu` (to access the uppermost element within the band in the j-th column) to `smu+ml` (to access the lowest element within the band in the j-th column). Indices from 0 to `smu-mu-1` give access to extra storage elements required by the LU decomposition function. Finally, `cols[j][i-j+smu]` is the (i, j)-th element with $j -$ mu $\leq i \leq j +$ ml.

The header file to be included when using this module is `sunmatrix/sunmatrix_band.h`.

The following macros are provided to access the content of a SUNMATRIX_BAND matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_B` denotes that these are specific to the *banded* version.

SM_CONTENT_B(A)
This macro gives access to the contents of the banded `SUNMatrix` *A*.

The assignment `A_cont = SM_CONTENT_B(A)` sets `A_cont` to be a pointer to the banded `SUNMatrix` content structure.

Implementation:

```
#define SM_CONTENT_B(A)    ( (SUNMatrixContent_Band)(A->content) )
```

SM_ROWS_B(A)
Access the number of rows in the banded `SUNMatrix` *A*.

This may be used either to retrieve or to set the value. For example, the assignment `A_rows = SM_ROWS_B(A)` sets `A_rows` to be the number of rows in the matrix `A`. Similarly, the assignment `SM_ROWS_B(A) = A_rows` sets the number of columns in `A` to equal `A_rows`.

Implementation:

```
#define SM_ROWS_B(A)    ( SM_CONTENT_B(A)->M )
```

SM_COLUMNS_B(A)
Access the number of columns in the banded `SUNMatrix` *A*. As with `SM_ROWS_B`, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_COLUMNS_B(A)    ( SM_CONTENT_B(A)->N )
```

SM_UBAND_B(A)
Access the `mu` parameter in the banded `SUNMatrix` *A*. As with `SM_ROWS_B`, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_UBAND_B(A)    ( SM_CONTENT_B(A)->mu )
```

SM_LBAND_B(A)
Access the `ml` parameter in the banded `SUNMatrix` *A*. As with `SM_ROWS_B`, this may be used either to retrieve or to set the value.

Implementation:

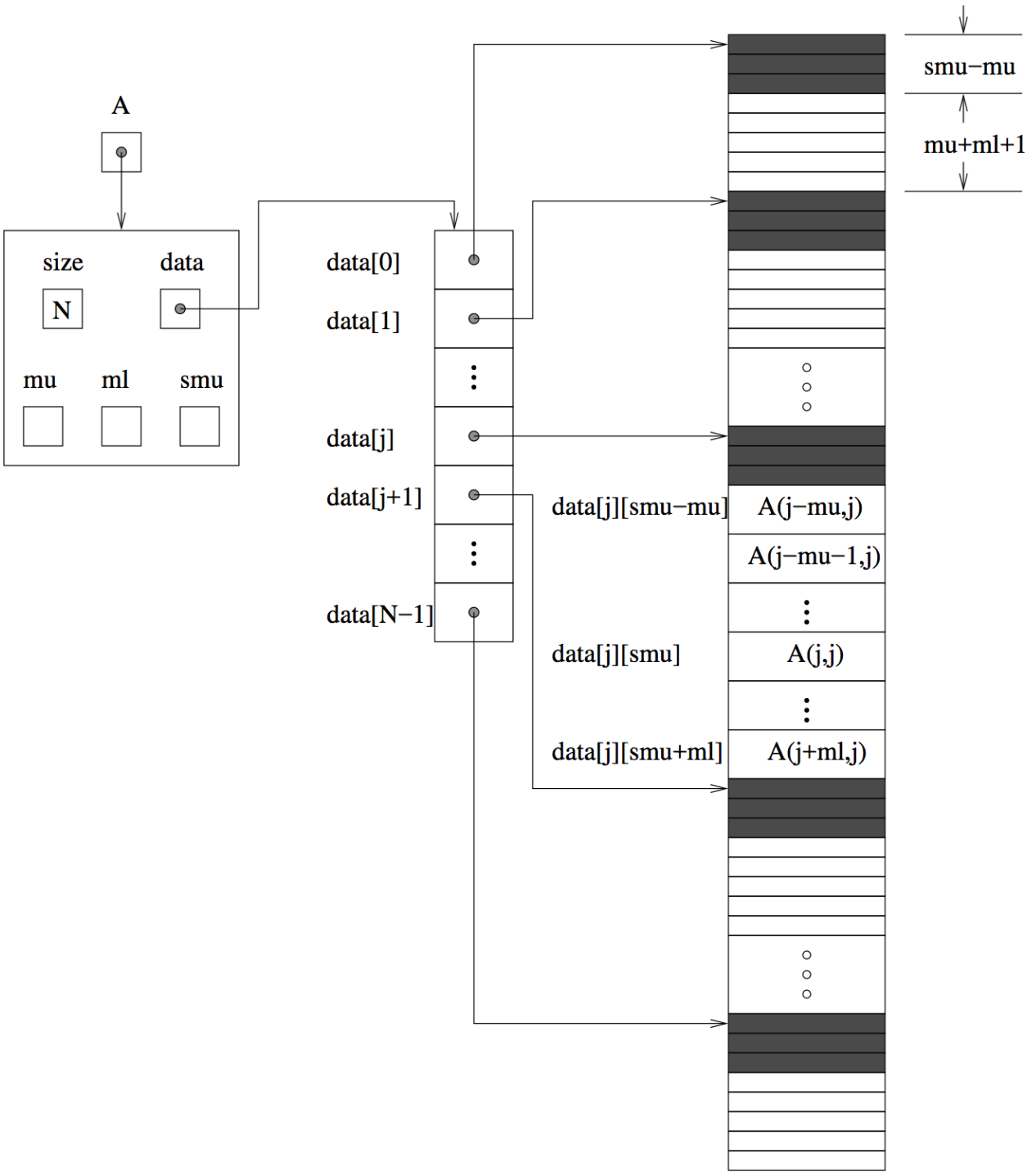


Fig. 11.1: Diagram of the storage for the SUNMATRIX_BAND module. Here `A` is an $N \times N$ band matrix with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to `N-1` and the (i, j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND linear solver.

```
#define SM_LBAND_B(A)    ( SM_CONTENT_B(A)->ml )
```

SM_SUBAND_B(A)
Access the smu parameter in the banded SUNMatrix A. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_SUBAND_B(A)    ( SM_CONTENT_B(A)->smu )
```

SM_LDIM_B(A)
Access the ldim parameter in the banded SUNMatrix A. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_LDIM_B(A)    ( SM_CONTENT_B(A)->ldim )
```

SM_LDATA_B(A)
Access the ldata parameter in the banded SUNMatrix A. As with SM_ROWS_B, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_LDATA_B(A)    ( SM_CONTENT_B(A)->ldata )
```

SM_DATA_B(A)
This macro gives access to the data pointer for the matrix entries.

The assignment A_data = SM_DATA_B(A) sets A_data to be a pointer to the first component of the data array for the banded SUNMatrix A. The assignment SM_DATA_B(A) = A_data sets the data array of A to be A_data by storing the pointer A_data.

Implementation:

```
#define SM_DATA_B(A)    ( SM_CONTENT_B(A)->data )
```

SM_COLS_B(A)
This macro gives access to the cols pointer for the matrix entries.

The assignment A_cols = SM_COLS_B(A) sets A_cols to be a pointer to the array of column pointers for the banded SUNMatrix A. The assignment SM_COLS_B(A) = A_cols sets the column pointer array of A to be A_cols by storing the pointer A_cols.

Implementation:

```
#define SM_COLS_B(A)    ( SM_CONTENT_B(A)->cols )
```

SM_COLUMN_B(A)
This macros gives access to the individual columns of the data array of a banded SUNMatrix.

The assignment col_j = SM_COLUMN_B(A,j) sets col_j to be a pointer to the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \le j \le N-1$. The type of the expression SM_COLUMN_B(A,j) is realtype *. The pointer returned by the call SM_COLUMN_B(A,j) can be treated as an array which is indexed from -mu to ml.

Implementation:

```
#define SM_COLUMN_B(A,j)    ( ((SM_CONTENT_B(A)->cols)[j])+SM_SUBAND_B(A) )
```

SM_ELEMENT_B(A)
This macro gives access to the individual entries of the data array of a banded SUNMatrix.

The assignments SM_ELEMENT_B(A,i,j) = a_ij and a_ij = SM_ELEMENT_B(A,i,j) reference the (i,j)-th element of the $N \times N$ band matrix A, where $0 \le i,j \le N-1$. The location (i,j) should further satisfy $j - \text{mu} \le i \le j + \text{ml}$.

Implementation:

```
#define SM_ELEMENT_B(A,i,j)    ( (SM_CONTENT_B(A)->cols)[j][(i)-(j)+SM_SUBAND_B(A)] )
```

SM_COLUMN_ELEMENT_B(A)
This macro gives access to the individual entries of the data array of a banded SUNMatrix.

The assignments SM_COLUMN_ELEMENT_B(col_j,i,j) = a_ij and a_ij = SM_COLUMN_ELEMENT_B(col_j,i,j) reference the (i,j)-th entry of the band matrix A when used in conjunction with SM_COLUMN_B to reference the j-th column through col_j. The index (i,j) should satisfy $j - \text{mu} \le i \le j + \text{ml}$.

Implementation:

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

The SUNMATRIX_BAND module defines banded implementations of all matrix operations listed in the section *Description of the SUNMATRIX operations*. Their names are obtained from those in that section by appending the suffix _Band (e.g. SUNMatCopy_Band). The module SUNMATRIX_BAND provides the following additional user-callable routines:

SUNMatrix **SUNBandMatrix**(sunindextype *N*, sunindextype *mu*, sunindextype *ml*)
This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, and the upper and lower half-bandwidths of the matrix, mu and ml. The stored upper bandwidth is set to mu+ml to accommodate subsequent factorization in the SUNLINSOL_BAND and SUNLINSOL_LAPACKBAND modules.

SUNMatrix **SUNBandMatrixStorage**(sunindextype *N*, sunindextype *mu*, sunindextype *ml*, sunindextype *smu*)
This constructor function creates and allocates memory for a banded SUNMatrix. Its arguments are the matrix size, N, the upper and lower half-bandwidths of the matrix, mu and ml, and the stored upper bandwidth, smu. When creating a band SUNMatrix, this value should be

- at least min(N-1,mu+ml) if the matrix will be used by the SUNLinSol_Band module;
- exactly equal to mu+ml if the matrix will be used by the SUNLinSol_LapackBand module;
- at least mu if used in some other manner.

Note: it is strongly recommended that users call the default constructor, :c:func:'SUNBandMatrix()', in all standard use cases. This advanced constructor is used internally within SUNDIALS solvers, and is provided to users who require banded matrices for non-default purposes.

void **SUNBandMatrix_Print**(SUNMatrix *A*, FILE* *outfile*)
This function prints the content of a banded SUNMatrix to the output stream specified by outfile. Note: stdout or stderr may be used as arguments for outfile to print directly to standard output or standard error, respectively.

sunindextype **SUNBandMatrix_Rows**(SUNMatrix *A*)
This function returns the number of rows in the banded SUNMatrix.

sunindextype **SUNBandMatrix_Columns**(SUNMatrix *A*)
This function returns the number of columns in the banded SUNMatrix.

sunindextype **SUNBandMatrix_LowerBandwidth**(SUNMatrix *A*)
This function returns the lower half-bandwidth for the banded SUNMatrix.

sunindextype **SUNBandMatrix_UpperBandwidth**(SUNMatrix *A*)
This function returns the upper half-bandwidth of the banded SUNMatrix.

sunindextype **SUNBandMatrix_StoredUpperBandwidth**(SUNMatrix *A*)
This function returns the stored upper half-bandwidth of the banded SUNMatrix.

sunindextype **SUNBandMatrix_LDim**(SUNMatrix *A*)
This function returns the length of the leading dimension of the banded SUNMatrix.

realtype* **SUNBandMatrix_Data**(SUNMatrix *A*)
This function returns a pointer to the data array for the banded SUNMatrix.

realtype** **SUNBandMatrix_Cols**(SUNMatrix *A*)
This function returns a pointer to the cols array for the band SUNMatrix.

realtype* **SUNBandMatrix_Column**(SUNMatrix *A*, sunindextype *j*)
This function returns a pointer to the diagonal entry of the j-th column of the banded SUNMatrix. The resulting pointer should be indexed over the range -mu to ml.

Notes

- When looping over the components of a banded SUNMatrix A, the most efficient approaches are to:
  - First obtain the component array via A_data = SM_DATA_B(A) or A_data = SUNBandMatrix_Data(A) and then access A_data[i] within the loop.
  - First obtain the array of column pointers via A_cols = SM_COLS_B(A) or A_cols = SUNBandMatrix_Cols(A), and then access A_cols[j][i] within the loop.
  - Within a loop over the columns, access the column pointer via A_colj = SUNBandMatrix_Column(A,j) and then to access the entries within that column using SM_COLUMN_ELEMENT_B(A_colj,i,j).

  All three of these are more efficient than using SM_ELEMENT_B(A,i,j) within a double loop.
- Within the SUNMatMatvec_Band routine, internal consistency checks are performed to ensure that the matrix is called with consistent N_Vector implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the SUNMATRIX_BAND module also includes the Fortran-callable function *FSUNBandMatInit()* to initialize this SUNMATRIX_BAND module for a given SUNDIALS solver.

subroutine FSUNBandMatInit(*CODE*, *N*, *MU*, *ML*, *IER*)
Initializes a band SUNMatrix structure for use in a SUNDIALS solver.

Arguments:

- *CODE* (int, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *N* (long int, input) – number of matrix rows (and columns).
- *MU* (long int, input) – upper half-bandwidth.
- *ML* (long int, input) – lower half-bandwidth.
- *IER* (int, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNBandMassMatInit()* initializes this SUNMATRIX_BAND module for storing the mass matrix.

subroutine FSUNBandMassMatInit(*N*, *MU*, *ML*, *IER*)

Initializes a band `SUNMatrix` structure for use as a mass matrix in ARKode.

Arguments:

- *N* (`long int`, input) – number of matrix rows (and columns).
- *MU* (`long int`, input) – upper half-bandwidth.
- *ML* (`long int`, input) – lower half-bandwidth.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

11.6 The SUNMATRIX_CUSPARSE Module

The SUNMATRIX_CUSPARSE implementation of the `SUNMatrix` module provided with SUNDIALS, is an interface to the NVIDIA cuSPARSE matrix for use on NVIDIA GPUs (*[cuSPARSE]*). All data stored by this matrix implementation resides on the GPU at all times.

The header file to be included when using this module is `sunmatrix/sunmatrix_cusparse.h`. The installed library to link to is `libsundials_sunmatrixcusparse.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

11.6.1 SUNMATRIX_CUSPARSE Description

The implementation currently supports the cuSPARSE CSR matrix format described in the cuSPARSE documentation as well as a unique low-storage format for block-diagonal matrices of the form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_1} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_n} \end{bmatrix}$$

where all the block matrices $\mathbf{A_j}$ share the same sparsisty pattern. We will refer to this format as BCSR (not to be confused with the canonical BSR format where each block is stored as dense). In this format, the CSR column indices and row pointers are only stored for the first block and are computed only as necessary for other blocks. This can drastically reduce the amount of storage required compared to the regular CSR format when there is a large number of blocks. This format is well-suited for, and intended to be used with the *The SUNLinSol_cuSolverSp_batchQR Module*.

The SUNMATRIX_CUSPARSE module is experimental and subject to change.

11.6.2 SUNMATRIX_CUSPARSE Functions

The SUNMATRIX_CUSPARSE module defines GPU-enabled sparse implementations of all matrix operations listed in the section *Description of the SUNMATRIX operations* except for the `SUNMatSpace` and `SUNMatMatvecSetup` operations:

- `SUNMatGetID_cuSparse` – returns `SUNMATRIX_CUSPARSE`
- `SUNMatClone_cuSparse`
- `SUNMatDestroy_cuSparse`
- `SUNMatZero_cuSparse`
- `SUNMatCopy_cuSparse`

- SUNMatScaleAdd_cuSparse – performs $A = cA + B$, where A and B must have the same sparsity pattern
- SUNMatScaleAddI_cuSparse – performs $A = cA + I$, where the diagonal of A must be present
- SUNMatMatvec_cuSparse

In addition, the SUNMATRIX_CUSPARSE module defines the following implementation specific functions:

SUNMatrix **SUNMatrix_cuSparse_NewCSR**(int *M*, int *N*, int *NNZ*, cusparseHandle_t *cusp*)
This constructor function creates and allocates memory for a SUMATRIX_CUSPARSE SUNMatrix that uses the CSR storage format. Its arguments are the number of rows and columns of the matrix, M and N, the number of nonzeros to be stored in the matrix, NNZ, and a valid cusparseHandle_t.

SUNMatrix **SUNMatrix_cuSparse_NewBlockCSR**(int *nblocks*, int *blockrows*, int *blockcols*, int *blocknnz*, cusparseHandle_t *cusp*)
This constructor function creates and allocates memory for a SUNMATRIX_CUSPARSE SUNMatrix object that leverages the SUNMAT_CUSPARSE_BCSR storage format to store a block diagonal matrix where each block shares the same sparsity pattern. The blocks must be square. The function arguments are the number of blocks,‘‘nblocks‘‘, the number of rows, blockrows, the number of columns, blockcols, the number of nonzeros in each each block, blocknnz, and a valid cuSPARSE handle.

The ‘‘SUNMAT_CUSPARSE_BCSR‘‘ format currently only supports square matrices..

SUNMatrix **SUNMatrix_cuSparse_MakeCSR**(cusparseMatDescr_t *mat_descr*, int *M*, int *N*, int *NNZ*, int **rowptrs*, int **colind*, realtype **data*, cusparseHandle_t *cusp*)
This constructor function creates a SUNMATRIX_CUSPARSE SUNMatrix object from user provided pointers. Its arguments are a cusparseMatDescr_t that must have index base CUSPARSE_INDEX_BASE_ZERO, the number of rows and columns of the matrix, M and N, the number of nonzeros to be stored in the matrix, NNZ, and a valid cusparseHandle_t.

int **SUNMatrix_cuSparse_Rows**(SUNMatrix *A*)
This function returns the number of rows in the sparse SUNMatrix.

int **SUNMatrix_cuSparse_Columns**(SUNMatrix *A*)
This function returns the number of columns in the sparse SUNMatrix.

int **SUNMatrix_cuSparse_NNZ**(SUNMatrix *A*)
This function returns the number of entries allocated for nonzero storage for the sparse SUNMatrix.

int **SUNMatrix_cuSparse_SparseType**(SUNMatrix *A*)
This function returns the storage type (SUNMAT_CUSPARSE_CSR or SUNMAT_CUSPARSE_BCSR) for the sparse SUNMatrix.

realtype* **SUNMatrix_cuSparse_Data**(SUNMatrix *A*)
This function returns a pointer to the data array for the sparse SUNMatrix.

int* **SUNMatrix_cuSparse_IndexValues**(SUNMatrix *A*)
This function returns a pointer to the index value array for the sparse SUNMatrix: for the CSR format this is an array of column indices for each nonzero entry. For the BCSR format this is an array of the column indices for each nonzero entry in the first block only.

int* **SUNMatrix_cuSparse_IndexPointers**(SUNMatrix *A*)
This function returns a pointer to the index pointer array for the sparse SUNMatrix: for the CSR format this is an array of the locations of the first entry of each row in the data and indexvalues arrays, for the BCSR format this is an array of the locations of each row in the data and indexvalues arrays in the first block only.

int **SUNMatrix_cuSparse_NumBlocks**(SUNMatrix *A*)
This function returns the number of matrix blocks.

int **SUNMatrix_cuSparse_BlockRows**(SUNMatrix *A*)
This function returns the number of rows in a matrix block.

int **SUNMatrix_cuSparse_BlockColumns**(SUNMatrix *A*)
This function returns the number of columns in a matrix block.

int **SUNMatrix_cuSparse_BlockNNZ**(SUNMatrix *A*)
This function returns the number of nonzeros in each matrix block.

realtype* **SUNMatrix_cuSparse_BlockData**(SUNMatrix *A*, int *blockidx*)
This function returns a pointer to the location in the `data` array where the data for the block, `blockidx`, begins. Thus, `blockidx` must be less than `SUNMatrix_cuSparse_NumBlocks(A)`. The first block in the SUNMatrix is index 0, the second block is index 1, and so on.

cusparseMatDescr_t **SUNMatrix_cuSparse_MatDescr**(SUNMatrix *A*)
This function returns the `cusparseMatDescr_t` object associated with the matrix.

int SUNMatrix_cuSparse_CopyToDevice(SUNMatrix A, realtype* h_data, int* h_idxptrs, int* h_idxvals)
This functions copies the matrix information to the GPU device from the provided host arrays. A user may provide `NULL` for any of `h_data`, `h_idxptrs`, or `h_idxvals` to avoid copying that information.

The function returns `SUNMAT_SUCCESS` if the copy operation(s) were successful, or a nonzero error code otherwise.

int **SUNMatrix_cuSparse_CopyFromDevice**(SUNMatrix *A*, realtype* *h_data*, int* *h_idxptrs*, int* *h_idxvals*)
This functions copies the matrix information from the GPU device to the provided host arrays. A user may provide `NULL` for any of `h_data`, `h_idxptrs`, or `h_idxvals` to avoid copying that information. Otherwise:

- The `h_data` array must be at least `SUNMatrix_cuSparse_NNZ(A)*sizeof(realtype)` bytes.
- **The `h_idxptrs` array must be at least** `(SUNMatrix_cuSparse_BlockDim(A)+1)*sizeof(int)` bytes.
- **The `h_idxvals` array must be at least** `(SUNMatrix_cuSparse_BlockNNZ(A))*sizeof(int)` bytes.

The function returns `SUNMAT_SUCCESS` if the copy operation(s) were successful, or a nonzero error code otherwise.

int **SUNMatrix_cuSparse_SetFixedPattern**(SUNMatrix *A*, booleantype *yesno*)
This function changes the behavior of the the `SUNMatZero` operation on the object `A`. By default the matrix sparsity pattern is not considered to be fixed, thus, the `SUNMatZero` operation zeros out all `data` array as well as the `indexvalues` and `indexpointers` arrays. Providing a value of `1` or `SUNTRUE` for the `yesno` argument changes the behavior of `SUNMatZero` on `A` so that only the data is zeroed out, but not the `indexvalues` or `indexpointers` arrays. Providing a value of `0` or `SUNFALSE` for the `yesno` argument is equivalent to the default behavior.

int **SUNMatrix_cuSparse_SetKernelExecPolicy**(SUNMatrix *A*, SUNCudaExecPolicy* *exec_policy*)
This function sets the execution policies which control the kernel parameters utilized when launching the CUDA kernels. By default the matrix is setup to use a policy which tries to leverage the structure of the matrix. See section *The SUNCudaExecPolicy Class* for more information about the `SUNCudaExecPolicy` class.

11.6.3 SUNMATRIX_CUSPARSE Usage Notes

The SUNMATRIX_CUSPARSE module only supports 32-bit indexing, thus SUNDIALS must be built for 32-bit indexing to use this module.

The SUNMATRIX_CUSPARSE module can be used with CUDA streams by calling the cuSPARSE function `cusparseSetStream` on the the `cusparseHandle_t` that is provided to the SUNMATRIX_CUSPARSE constructor.

Warning: When using the SUNMATRIX_CUSPARSE module with a SUNDIALS package (e.g. ARKODE), the stream given to cuSPARSE should be the same stream used for the NVECTOR object that is provided to the package, and the NVECTOR object given to the `SUNMatvec` operation. If different streams are utilized, synchronization issues may occur.

11.7 The SUNMATRIX_SPARSE Module

The sparse implementation of the `SUNMatrix` module provided with SUNDIALS, SUNMATRIX_SPARSE, is designed to work with either *compressed-sparse-column* (CSC) or *compressed-sparse-row* (CSR) sparse matrix formats. To this end, it defines the *content* field of `SUNMatrix` to be the following structure:

```
struct _SUNMatrixContent_Sparse {
  sunindextype M;
  sunindextype N;
  sunindextype NNZ;
  sunindextype NP;
  realtype *data;
  int sparsetype;
  sunindextype *indexvals;
  sunindextype *indexptrs;
  /* CSC indices */
  sunindextype **rowvals;
  sunindextype **colptrs;
  /* CSR indices */
  sunindextype **colvals;
  sunindextype **rowptrs;
};
```

A diagram of the underlying data representation in a sparse matrix is shown in Figure *SUNSparseMatrix Diagram*. A more complete description of the parts of this *content* field is given below:

- `M` - number of rows
- `N` - number of columns
- `NNZ` - maximum number of nonzero entries in the matrix (allocated length of `data` and `indexvals` arrays)
- `NP` - number of index pointers (e.g. number of column pointers for CSC matrix). For CSC matrices `NP=N`, and for CSR matrices `NP=M`. This value is set automatically at construction based the input choice for `sparsetype`.
- `data` - pointer to a contiguous block of `realtype` variables (of length `NNZ`), containing the values of the nonzero entries in the matrix
- `sparsetype` - type of the sparse matrix (`CSC_MAT` or `CSR_MAT`)

- `indexvals` - pointer to a contiguous block of `int` variables (of length `NNZ`), containing the row indices (if CSC) or column indices (if CSR) of each nonzero matrix entry held in `data`
- `indexptrs` - pointer to a contiguous block of `int` variables (of length `NP+1`). For CSC matrices each entry provides the index of the first column entry into the `data` and `indexvals` arrays, e.g. if `indexptr[3]=7`, then the first nonzero entry in the fourth column of the matrix is located in `data[7]`, and is located in row `indexvals[7]` of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the `data` and `indexvals` arrays. For CSR matrices, each entry provides the index of the first row entry into the `data` and `indexvals` arrays.

The following pointers are added to the SUNMATRIX_SPARSE content structure for user convenience, to provide a more intuitive interface to the CSC and CSR sparse matrix data structures. They are set automatically when creating a sparse `SUNMatrix`, based on the sparse matrix storage type.

- `rowvals` - pointer to `indexvals` when `sparsetype` is `CSC_MAT`, otherwise set to `NULL`.
- `colptrs` - pointer to `indexptrs` when `sparsetype` is `CSC_MAT`, otherwise set to `NULL`.
- `colvals` - pointer to `indexvals` when `sparsetype` is `CSR_MAT`, otherwise set to `NULL`.
- `rowptrs` - pointer to `indexptrs` when `sparsetype` is `CSR_MAT`, otherwise set to `NULL`.

For example, the 5×4 matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored as a CSC matrix in this structure as either

```
M = 5;
N = 4;
NNZ = 8;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4};
indexptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
NP = N;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
sparsetype = CSC_MAT;
indexvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
indexptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with $*$ may contain any values). Note in both cases that the final value in `indexptrs` is 8, indicating the total number of nonzero entries in the matrix.

Similarly, in CSR format, the same matrix could be stored as

```
M = 5;
N = 4;
NNZ = 8;
```

```
NP = M;
data = {3.0, 1.0, 3.0, 2.0, 7.0, 1.0, 9.0, 5.0};
sparsetype = CSR_MAT;
indexvals = {1, 2, 0, 3, 1, 0, 3, 3};
indexptrs = {0, 2, 4, 5, 7, 8};
```

The header file to be included when using this module is `sunmatrix/sunmatrix_sparse.h`.

The following macros are provided to access the content of a SUNMATRIX_SPARSE matrix. The prefix `SM_` in the names denotes that these macros are for *SUNMatrix* implementations, and the suffix `_S` denotes that these are specific to the *sparse* version.

SM_CONTENT_S(A)
This macro gives access to the contents of the sparse `SUNMatrix` *A*.

The assignment `A_cont = SM_CONTENT_S(A)` sets `A_cont` to be a pointer to the sparse `SUNMatrix` content structure.

Implementation:

```
#define SM_CONTENT_S(A)    ( (SUNMatrixContent_Sparse)(A->content) )
```

SM_ROWS_S(A)
Access the number of rows in the sparse `SUNMatrix` *A*.

This may be used either to retrieve or to set the value. For example, the assignment `A_rows = SM_ROWS_S(A)` sets `A_rows` to be the number of rows in the matrix *A*. Similarly, the assignment `SM_ROWS_S(A) = A_rows` sets the number of columns in *A* to equal `A_rows`.

Implementation:

```
#define SM_ROWS_S(A)    ( SM_CONTENT_S(A)->M )
```

SM_COLUMNS_S(A)
Access the number of columns in the sparse `SUNMatrix` *A*. As with `SM_ROWS_S`, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_COLUMNS_S(A)    ( SM_CONTENT_S(A)->N )
```

SM_NNZ_S(A)
Access the allocated number of nonzeros in the sparse `SUNMatrix` *A*. As with `SM_ROWS_S`, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_NNZ_S(A)    ( SM_CONTENT_S(A)->NNZ )
```

SM_NP_S(A)
Access the number of index pointers `NP` in the sparse `SUNMatrix` *A*. As with `SM_ROWS_S`, this may be used either to retrieve or to set the value.

Implementation:

```
#define SM_NP_S(A)    ( SM_CONTENT_S(A)->NP )
```

SM_SPARSETYPE_S(A)
Access the sparsity type parameter in the sparse `SUNMatrix` *A*. As with `SM_ROWS_S`, this may be used either to retrieve or to set the value.

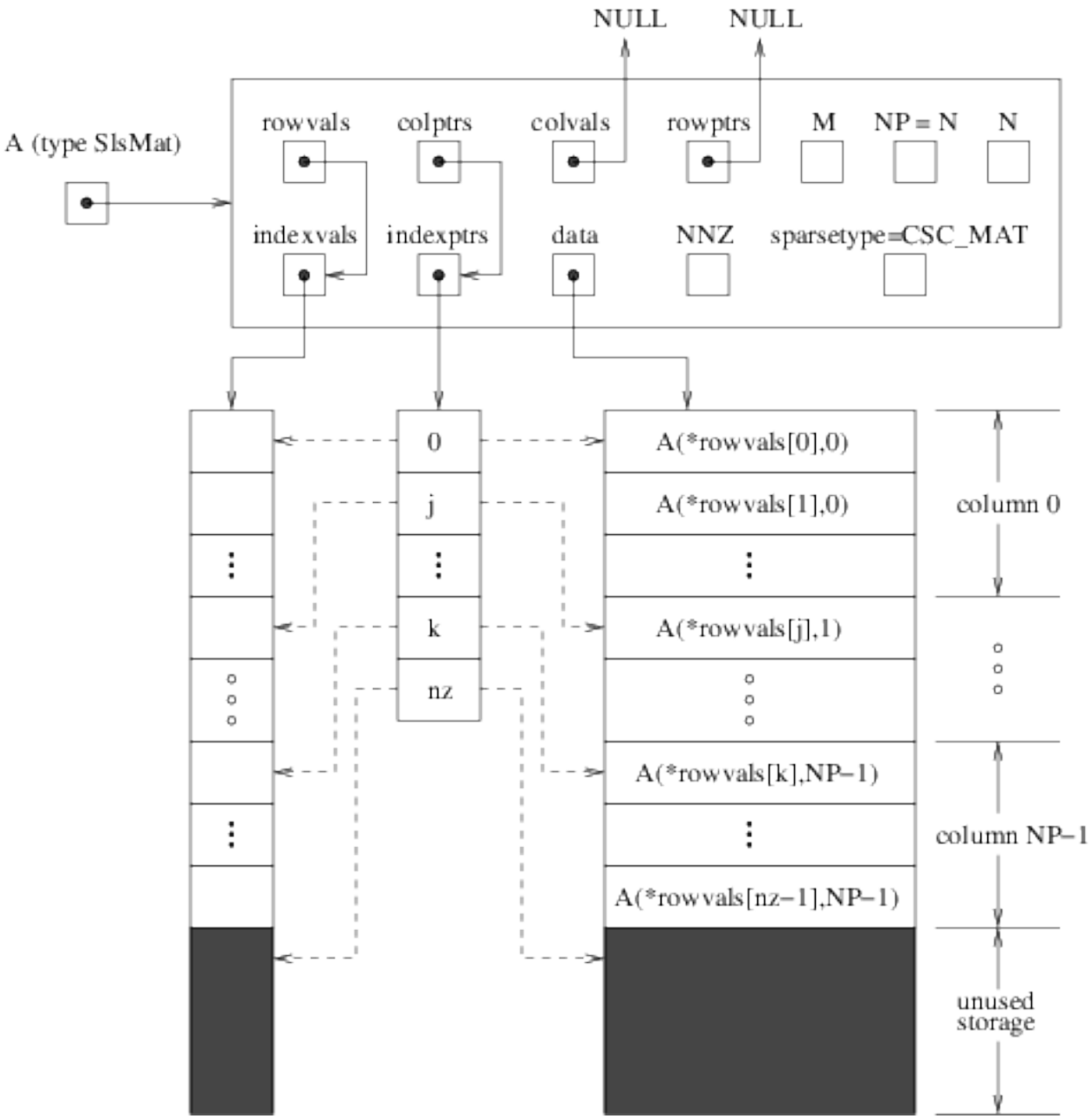


Fig. 11.2: Diagram of the storage for a compressed-sparse-column matrix of type SUNMATRIX_SPARSE: Here `A` is an $M \times N$ sparse CSC matrix with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from `0` to `M-1`, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row `i`, column `j` entry of `A` (again, zero-based) denoted as `A(i,j)`. The `indexptrs` array contains `N+1` entries; the first `N` denote the starting index of each column within the `indexvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although `NNZ` values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `indexvals` indicate extra allocated space.

Implementation:

```
#define SM_SPARSETYPE_S(A)    ( SM_CONTENT_S(A)->sparsetype )
```

SM_DATA_S(A)
This macro gives access to the `data` pointer for the matrix entries.

The assignment `A_data = SM_DATA_S(A)` sets `A_data` to be a pointer to the first component of the data array for the sparse `SUNMatrix A`. The assignment `SM_DATA_S(A) = A_data` sets the data array of *A* to be `A_data` by storing the pointer `A_data`.

Implementation:

```
#define SM_DATA_S(A)    ( SM_CONTENT_S(A)->data )
```

SM_INDEXVALS_S(A)
This macro gives access to the `indexvals` pointer for the matrix entries.

The assignment `A_indexvals = SM_INDEXVALS_S(A)` sets `A_indexvals` to be a pointer to the array of index values (i.e. row indices for a CSC matrix, or column indices for a CSR matrix) for the sparse `SUNMatrix` *A*.

Implementation:

```
#define SM_INDEXVALS_S(A)    ( SM_CONTENT_S(A)->indexvals )
```

SM_INDEXPTRS_S(A)
This macro gives access to the `indexptrs` pointer for the matrix entries.

The assignment `A_indexptrs = SM_INDEXPTRS_S(A)` sets `A_indexptrs` to be a pointer to the array of index pointers (i.e. the starting indices in the data/indexvals arrays for each row or column in CSR or CSC formats, respectively).

Implementation:

```
#define SM_INDEXPTRS_S(A)    ( SM_CONTENT_S(A)->indexptrs )
```

The SUNMATRIX_SPARSE module defines sparse implementations of all matrix operations listed in the section *Description of the SUNMATRIX operations*. Their names are obtained from those in that section by appending the suffix `_Sparse` (e.g. `SUNMatCopy_Sparse`). The module SUNMATRIX_SPARSE provides the following additional user-callable routines:

SUNMatrix **SUNSparseMatrix**(sunindextype *M*, sunindextype *N*, sunindextype *NNZ*, int *sparsetype*)
This constructor function creates and allocates memory for a sparse `SUNMatrix`. Its arguments are the number of rows and columns of the matrix, *M* and *N*, the maximum number of nonzeros to be stored in the matrix, *NNZ*, and a flag *sparsetype* indicating whether to use CSR or CSC format (valid choices are `CSR_MAT` or `CSC_MAT`).

SUNMatrix **SUNSparseFromDenseMatrix**(SUNMatrix *A*, realtype *droptol*, int *sparsetype*)
This constructor function creates a new sparse matrix from an existing SUNMATRIX_DENSE object by copying all values with magnitude larger than *droptol* into the sparse matrix structure.

Requirements:

- *A* must have type `SUNMATRIX_DENSE`
- *droptol* must be non-negative
- *sparsetype* must be either `CSC_MAT` or `CSR_MAT`

The function returns `NULL` if any requirements are violated, or if the matrix storage request cannot be satisfied.

SUNMatrix **SUNSparseFromBandMatrix**(SUNMatrix *A*, realtype *droptol*, int *sparsetype*)
This constructor function creates a new sparse matrix from an existing SUNMATRIX_BAND object by copying all values with magnitude larger than *droptol* into the sparse matrix structure.

Requirements:

- *A* must have type `SUNMATRIX_BAND`
- *droptol* must be non-negative
- *sparsetype* must be either `CSC_MAT` or `CSR_MAT`.

The function returns `NULL` if any requirements are violated, or if the matrix storage request cannot be satisfied.

int **SUNSparseMatrix_Realloc**(SUNMatrix *A*)
This function reallocates internal storage arrays in a sparse matrix so that the resulting sparse matrix has no wasted space (i.e. the space allocated for nonzero entries equals the actual number of nonzeros, `indexptrs[NP]`). Returns 0 on success and 1 on failure (e.g. if the input matrix is not sparse).

void **SUNSparseMatrix_Print**(SUNMatrix *A*, FILE* *outfile*)
This function prints the content of a sparse `SUNMatrix` to the output stream specified by `outfile`. Note: `stdout` or `stderr` may be used as arguments for `outfile` to print directly to standard output or standard error, respectively.

sunindextype **SUNSparseMatrix_Rows**(SUNMatrix *A*)
This function returns the number of rows in the sparse `SUNMatrix`.

sunindextype **SUNSparseMatrix_Columns**(SUNMatrix *A*)
This function returns the number of columns in the sparse `SUNMatrix`.

sunindextype **SUNSparseMatrix_NNZ**(SUNMatrix *A*)
This function returns the number of entries allocated for nonzero storage for the sparse `SUNMatrix`.

sunindextype **SUNSparseMatrix_NP**(SUNMatrix *A*)
This function returns the number of index pointers for the sparse `SUNMatrix` (the `indexptrs` array has `NP+1` entries).

int **SUNSparseMatrix_SparseType**(SUNMatrix *A*)
This function returns the storage type (`CSR_MAT` or `CSC_MAT`) for the sparse `SUNMatrix`.

realtype* **SUNSparseMatrix_Data**(SUNMatrix *A*)
This function returns a pointer to the data array for the sparse `SUNMatrix`.

sunindextype* **SUNSparseMatrix_IndexValues**(SUNMatrix *A*)
This function returns a pointer to index value array for the sparse `SUNMatrix`: for CSR format this is the column index for each nonzero entry, for CSC format this is the row index for each nonzero entry.

sunindextype* **SUNSparseMatrix_IndexPointers**(SUNMatrix *A*)
This function returns a pointer to the index pointer array for the sparse `SUNMatrix`: for CSR format this is the location of the first entry of each row in the `data` and `indexvalues` arrays, for CSC format this is the location of the first entry of each column.

Note: Within the `SUNMatMatvec_Sparse` routine, internal consistency checks are performed to ensure that the matrix is called with consistent `N_Vector` implementations. These are currently limited to: NVECTOR_SERIAL, NVECTOR_OPENMP, NVECTOR_PTHREADS, and NVECTOR_CUDA when using managed memory. As additional compatible vector implementations are added to SUNDIALS, these will be included within this compatibility check.

For solvers that include a Fortran interface module, the SUNMATRIX_SPARSE module also includes the Fortran-callable function `FSUNSparseMatInit()` to initialize this SUNMATRIX_SPARSE module for a given SUNDIALS solver.

subroutine FSUNSparseMatInit(*CODE*, *M*, *N*, *NNZ*, *SPARSETYPE*, *IER*)
Initializes a sparse `SUNMatrix` structure for use in a SUNDIALS solver.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *M* (`long int`, input) – number of matrix rows.
- *N* (`long int`, input) – number of matrix columns.
- *NNZ* (`long int`, input) – amount of nonzero storage to allocate.
- *SPARSETYPE* (`int`, input) – matrix sparsity type (`CSC_MAT` or `CSR_MAT`)
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function `FSUNSparseMassMatInit()` initializes this SUNMATRIX_SPARSE module for storing the mass matrix.

subroutine FSUNSparseMassMatInit(*M*, *N*, *NNZ*, *SPARSETYPE*, *IER*)
Initializes a sparse `SUNMatrix` structure for use as a mass matrix in ARKode.

Arguments:

- *M* (`long int`, input) – number of matrix rows.
- *N* (`long int`, input) – number of matrix columns.
- *NNZ* (`long int`, input) – amount of nonzero storage to allocate.
- *SPARSETYPE* (`int`, input) – matrix sparsity type (`CSC_MAT` or `CSR_MAT`)
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

11.8 The SUNMATRIX_SLUNRLOC Module

The SUNMATRIX_SLUNRLOC implementation of the SUNMatrix module provided with SUNDIALS is an adapter for the `SuperMatrix` structure provided by the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li (*[SuperLUDIST]*, *[GDL2007]*, *[LD2003]*, *[SLUUG1999]*). It is designed to be used with the SuperLU_DIST `SUNLinearSolver` discussed in Section *The SUNLinSol_SuperLUDIST Module*. To this end, it defines the `content` field of `SUNMatrix` to be the following structure:

```
struct _SUNMatrixContent_SLUNRloc {
  booleantype   own_data;
  gridinfo_t    *grid;
  sunindextype  *row_to_proc;
  pdgsmv_comm_t *gsmv_comm;
  SuperMatrix   *A_super;
  SuperMatrix   *ACS_super;
};
```

A more complete description of the this `content` field is given below:

- `own_data` – a flag which indicates if the SUNMatrix is responsible for freeing `A_super`
- `grid` – pointer to the SuperLU_DIST structure that stores the 2D process grid

- row_to_proc – a mapping between the rows in the matrix and the process it resides on; will be NULL until the SUNMatMatvecSetup routine is called
- gsmv_comm – pointer to the SuperLU_DIST structure that stores the communication information needed for matrix-vector multiplication; will be NULL until the SUNMatMatvecSetup routine is called
- A_super – pointer to the underlying SuperLU_DIST SuperMatrix with Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE; must have the full diagonal present to be used with SUNMatScaleAddI routine
- ACS_super – a column-sorted version of the matrix needed to perform matrix-vector multiplication; will be NULL until the routine SUNMatMatvecSetup routine is called

The header file to include when using this module is sunmatrix/sunmatrix_slunrloc.h. The installed module library to link to is libsundials_sunmatrixslunrloc.*lib* where *.lib* is typically .so for shared libraries and .a for static libraries.

11.8.1 SUNMATRIX_SLUNRLOC Functions

The SUNMATRIX_SLUNRLOC module provides the following user-callable routines:

SUNMatrix **SUNMatrix_SLUNRloc**(SuperMatrix **Asuper*, gridinfo_t **grid*)
This constructor function creates and allocates memory for a SUNMatrix_SLUNRloc object. Its arguments are a fully-allocated SuperLU_DIST SuperMatrix with Stype = SLU_NR_loc, Dtype = SLU_D, Mtype = SLU_GE and an initialized SuperLU_DIST 2D process grid structure. It returns a SUNMatrix object if Asuper is compatible else it returns NULL.

void **SUNMatrix_SLUNRloc_Print**(SUNMatrix *A*, FILE **fp*)
This function prints the underlying SuperMatrix content. It is useful for debugging. Its arguments are the SUNMatrix object and a FILE pointer to print to. It returns void.

SuperMatrix* **SUNMatrix_SLUNRloc_SuperMatrix**(SUNMatrix *A*)
This function returns the underlying SuperMatrix of A. Its only argument is the SUNMatrix object to access.

gridinfo_t* **SUNMatrix_SLUNRloc_ProcessGrid**(SUNMatrix *A*)
This function returns the SuperLU_DIST 2D process grid associated with A. Its only argument is the SUNMatrix object to access.

booleantype **SUNMatrix_SLUNRloc_OwnData**(SUNMatrix *A*)
This function returns true if the SUNMatrix object is responsible for freeing the underlying SuperMatrix, otherwise it returns false. Its only argument is the SUNMatrix object to access.

The SUNMATRIX_SLUNRLOC module also defines implementations of all generic SUNMatrix operations listed in Table *Description of the SUNMATRIX operations*:

- SUNMatGetID_SLUNRloc – returns SUNMATRIX_SLUNRLOC
- SUNMatClone_SLUNRloc
- SUNMatDestroy_SLUNRloc
- SUNMatSpace_SLUNRloc – this only returns information for the storage within the matrix interface, i.e. storage for row_to_proc
- SUNMatZero_SLUNRloc
- SUNMatCopy_SLUNRloc
- SUNMatScaleAdd_SLUNRloc – performs $A = cA + B$, where A and B must have the same sparsity pattern

- SUNMatScaleAddI_SLUNRloc – performs $A = cA + I$, where the diagonal of A must be present
- SUNMatMatvecSetup_SLUNRloc – initializes the SuperLU_DIST parallel communication structures needed to perform a matrix-vector product; only needs to be called before the first call to SUNMatMatvec or if the matrix changed since the last setup
- SUNMatMatvec_SLUNRloc

11.9 SUNMATRIX Examples

There are SUNMatrix examples that may be installed for each implementation: dense, banded, and sparse. Each implementation makes use of the functions in test_sunmatrix.c. These example functions show simple usage of the SUNMatrix family of functions. The inputs to the examples depend on the matrix type, and are output to stdout if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in test_sunmatrix.c:

- Test_SUNMatGetID: Verifies the returned matrix ID against the value that should be returned.
- Test_SUNMatClone: Creates clone of an existing matrix, copies the data, and checks that their values match.
- Test_SUNMatZero: Zeros out an existing matrix and checks that each entry equals 0.0.
- Test_SUNMatCopy: Clones an input matrix, copies its data to a clone, and verifies that all values match.
- Test_SUNMatScaleAdd: Given an input matrix A and an input identity matrix I, this test clones and copies A to a new matrix B, computes $B = -B + B$, and verifies that the resulting matrix entries equal 0. Additionally, if the matrix is square, this test clones and copies A to a new matrix D, clones and copies I to a new matrix C, computes $D = D+I$ and $C = C+A$ using SUNMatScaleAdd, and then verifies that $C = D$.
- Test_SUNMatScaleAddI: Given an input matrix A and an input identity matrix I, this clones and copies I to a new matrix B, computes $B = -B + I$ using SUNMatScaleAddI, and verifies that the resulting matrix entries equal 0.
- Test_SUNMatMatvec Given an input matrix A and input vectors x and y such that $y = Ax$, this test has different behavior depending on whether A is square. If it is square, it clones and copies A to a new matrix B, computes $B = 3B + I$ using SUNMatScaleAddI, clones y to new vectors w and z, computes $z = Bx$ using SUNMatMatvec, computes $w = 3y+x$ using N_VLinearSum, and verifies that $w == z$. If A is not square, it just clones y to a new vector z, *computes :math:'z=Ax* using SUNMatMatvec, and verifies that $y = z$.
- Test_SUNMatSpace: verifies that SUNMatSpace can be called, and outputs the results to stdout.

11.10 SUNMATRIX functions required by ARKode

In Table *List of matrix functions usage by ARKode code modules*, we list the matrix functions in the SUNMatrix module used within the ARKode package. The table also shows, for each function, which of the code modules uses the function. The main ARKode time step modules, ARKStep and ERKStep, do not call any SUNMatrix functions directly, so the table columns are specific to the ARKLS interface and the ARKBANDPRE and ARKBBDPRE preconditioner modules. We further note that the ARKLS interface only utilizes these routines when supplied with a *matrix-based* linear solver, i.e. the SUNMatrix object (*J* or *M*) passed to *ARKStepSetLinearSolver()* or *ARKStepSetMassLinearSolver()* was not NULL.

At this point, we should emphasize that the ARKode user does not need to know anything about the usage of matrix functions by the ARKode code modules in order to use ARKode. The information is presented as an implementation detail for the interested reader.

11.10.1 List of matrix functions usage by ARKode code modules

Routine	ARKLS	ARKBANDPRE	ARKBBDPRE
SUNMatGetID	X		
SUNMatClone	X		
SUNMatDestroy	X	X	X
SUNMatZero	X	X	X
SUNMatCopy	X	X	X
SUNMatScaleAddI	X	X	X
SUNMatScaleAdd	1		
SUNMatMatvec	1		
SUNMatMatvecSetup	1,2		
SUNMatSpace	2	2	2

1. These matrix functions are only used for problems involving a non-identity mass matrix.
2. These matrix functions are optionally used, in that these are only called if they are implemented in the `SUNMatrix` module that is being used (i.e. their function pointers are non-`NULL`). If not supplied, these modules will assume that the matrix requires no storage.

We note that both the ARKBANDPRE and ARKBBDPRE preconditioner modules are hard-coded to use the SUNDIALS-supplied band `SUNMatrix` type, so the most useful information above for user-supplied `SUNMatrix` implementations is the column relating to ARKLS requirements.

Chapter 12

Description of the SUNLinearSolver module

For problems that require the solution of linear systems of equations, the SUNDIALS packages operate using generic linear solver modules defined through the SUNLinSol API. This allows SUNDIALS packages to utilize any valid SUNLinSol implementation that provides a set of required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group consists of "set" routines to supply the linear solver object with functions provided by the SUNDIALS package, or for modification of solver parameters. The last group consists of "get" routines for retrieving artifacts (statistics, residual vectors, etc.) from the linear solver. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

The implementations provided with SUNDIALS work in coordination with the SUNDIALS generic `N_Vector` and `SUNMatrix` modules to provide a set of compatible data structures and solvers for the solution of linear systems using direct or iterative (matrix-based or matrix-free) methods. Moreover, advanced users can provide a customized `SUNLinearSolver` implementation to any SUNDIALS package, particularly in cases where they provide their own `N_Vector` and/or `SUNMatrix` modules.

Historically, the SUNDIALS packages have been designed to specifically leverage the use of either *direct linear solvers* or matrix-free, *scaled, preconditioned, iterative linear solvers*. However, matrix-based iterative linear solvers are also supported.

The iterative linear solvers packaged with SUNDIALS leverage scaling and preconditioning, as applicable, to balance error between solution components and to accelerate convergence of the linear solver. To this end, instead of solving the linear system $Ax = b$ directly, these apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{12.1}$$

where

$$\begin{aligned} \tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\ \tilde{b} &= S_1 P_1^{-1} b, \\ \tilde{x} &= S_2 P_2 x, \end{aligned} \tag{12.2}$$

and where

- P_1 is the left preconditioner,
- P_2 is the right preconditioner,
- S_1 is a diagonal matrix of scale factors for $P_1^{-1} b$,
- S_2 is a diagonal matrix of scale factors for $P_2 x$.

SUNDIALS solvers request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\left\|\tilde{b} - \tilde{A}\tilde{x}\right\|_2 < \text{tol}.$$

When provided an iterative SUNLinSol implementation that does not support the scaling matrices S_1 and S_2, SUNDIALS' packages will adjust the value of tol accordingly (see the section *Iterative linear solver tolerance* for more details). In this case, they instead request that iterative linear solvers stop based on the criteria

$$\left\|P_1^{-1}b - P_1^{-1}Ax\right\|_2 < \text{tol}.$$

We note that the corresponding adjustments to tol in this case are non-optimal, in that they cannot balance error between specific entries of the solution x, only the aggregate error in the overall solution vector.

We further note that not all of the SUNDIALS-provided iterative linear solvers support the full range of the above options (e.g., separate left/right preconditioning), and that some of the SUNDIALS packages only utilize a subset of these options. Further details on these exceptions are described in the documentation for each `SUNLinearSolver` implementation, or for each SUNDIALS package.

For users interested in providing their own SUNLinSol module, the following section presents the SUNLinSol API and its implementation beginning with the definition of SUNLinSol functions in sections *SUNLinearSolver core functions* – *SUNLinearSolver get functions*. This is followed by the definition of functions supplied to a linear solver implementation in section *Functions provided by SUNDIALS packages*. The linear solver return codes are described in section *SUNLinearSolver return codes*. The `SUNLinearSolver` type and the generic SUNLinSol module are defined in section *The generic SUNLinearSolver module*. The section *Compatibility of SUNLinearSolver modules* discusses compatibility between the SUNDIALS-provided SUNLinSol modules and SUNMATRIX modules. Section *Implementing a custom SUNLinearSolver module* lists the requirements for supplying a custom SUNLinSol module and discusses some intended use cases. Users wishing to supply their own SUNLinSol module are encouraged to use the SUNLinSol implementations provided with SUNDIALS as a template for supplying custom linear solver modules. The SUNLinSol functions required by this SUNDIALS package as well as other package specific details are given in section *ARKode SUNLinearSolver interface*. The remaining sections of this chapter present the SUNLinSol modules provided with SUNDIALS.

12.1 The SUNLinearSolver API

The SUNLinSol API defines several linear solver operations that enable SUNDIALS packages to utilize any SUNLinSol implementation that provides the required functions. These functions can be divided into three categories. The first are the core linear solver functions. The second group of functions consists of set routines to supply the linear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of get routines for retrieving linear solver statistics. All of these functions are defined in the header file `sundials/sundials_linearsolver.h`.

12.1.1 SUNLinearSolver core functions

The core linear solver functions consist of two required functions to get the linear solver type (*SUNLinSolGetType()*) and solve the linear system $Ax = b$ (*SUNLinSolSolve()*). The remaining functions are for getting the solver ID (*SUNLinSolGetID()*), initializing the linear solver object once all solver-specific options have been set (*SUNLinSolInitialize()*), setting up the linear solver object to utilize an updated matrix A (*SUNLinSolSetup()*), and for destroying the linear solver object (*SUNLinSolFree()*) are optional.

SUNLinearSolver_Type **SUNLinSolGetType**(SUNLinearSolver *LS*)

Returns the type identifier for the linear solver *LS*. It is used to determine the solver type (direct, iterative, or matrix-iterative) from the abstract `SUNLinearSolver` interface. Returned values are one of the following:

- SUNLINEARSOLVER_DIRECT – 0, the SUNLinSol module requires a matrix, and computes an 'exact' solution to the linear system defined by that matrix.
- SUNLINEARSOLVER_ITERATIVE – 1, the SUNLinSol module does not require a matrix (though one may be provided), and computes an inexact solution to the linear system using a matrix-free iterative algorithm. That is it solves the linear system defined by the package-supplied ATimes routine (see *SUNLinSolSetATimes()* below), even if that linear system differs from the one encoded in the matrix object (if one is provided). As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.
- SUNLINEARSOLVER_MATRIX_ITERATIVE – 2, the SUNLinSol module requires a matrix, and computes an inexact solution to the linear system defined by that matrix using an iterative algorithm. That is it solves the linear system defined by the matrix object even if that linear system differs from that encoded by the package-supplied ATimes routine. As the solver computes the solution only inexactly (or may diverge), the linear solver should check for solution convergence/accuracy as appropriate.

Usage:

```
type = SUNLinSolGetType(LS);
```

Notes: See section *Intended use cases* for more information on intended use cases corresponding to the linear solver type.

SUNLinearSolver_ID **SUNLinSolGetID**(SUNLinearSolver *LS*)
Returns the identifier for the linear solver *LS*. It is recommended that a user-supplied SUNLinearSolver implementation return the SUNLINEARSOLVER_CUSTOM identifier.

Usage:

```
id = SUNLinSolGetID(LS);
```

int **SUNLinSolInitialize**(SUNLinearSolver *LS*)
Performs linear solver initialization (assuming that all solver-specific options have been set). This should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in section *SUNLinearSolver return codes*.

Usage:

```
retval = SUNLinSolInitialize(LS);
```

int **SUNLinSolSetup**(SUNLinearSolver *LS*, SUNMatrix *A*)
Performs any linear solver setup needed, based on an updated system SUNMatrix *A*. This may be called frequently (e.g., with a full Newton method) or infrequently (for a modified Newton method), based on the type of integrator and/or nonlinear solver requesting the solves. This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in section *SUNLinearSolver return codes*.

Usage:

```
retval = SUNLinSolSetup(LS, A);
```

int **SUNLinSolSolve**(SUNLinearSolver *LS*, SUNMatrix *A*, N_Vector *x*, N_Vector *b*, realtype *tol*)
This *required* function Solves a linear system $Ax = b$.

Arguments:

- *LS* – a SUNLinSol object.
- *A* – a SUNMatrix object.

- *x* – a `N_Vector` object containing the initial guess for the solution of the linear system, and the solution to the linear system upon return.
- *b* – a `N_Vector` object containing the linear system right-hand side.
- *tol* – the desired linear solver tolerance.

Return value: This should return zero for a successful call, a positive value for a recoverable failure and a negative value for an unrecoverable failure, ideally returning one of the generic error codes listed in section *SUNLinearSolver return codes*.

Direct solvers: can ignore the *tol* argument.

Matrix-free solvers: (those that identify as `SUNLINEARSOLVER_ITERATIVE`) can ignore the `SUNMatrix` input *A*, and should rely on the matrix-vector product function supplied through the routine *`SUNLinSolSetATimes()`*.

Iterative solvers: (those that identify as `SUNLINEARSOLVER_ITERATIVE` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`) should attempt to solve to the specified tolerance *tol* in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

Usage:

```
retval = SUNLinSolSolve(LS, A, x, b, tol);
```

int **`SUNLinSolFree`**(SUNLinearSolver *LS*)
Frees memory allocated by the linear solver. This should return zero for a successful call, and a negative value for a failure.

Usage:

```
retval = SUNLinSolFree(LS);
```

12.1.2 SUNLinearSolver set functions

The following set functions are used to supply linear solver modules with functions defined by the SUNDIALS packages and to modify solver parameters. Only the routine for setting the matrix-vector product routine is required, and that is only for matrix-free linear solver modules. Otherwise, all other set functions are optional. SUNLinSol implementations that do not provide the functionality for any optional routine should leave the corresponding function pointer `NULL` instead of supplying a dummy routine.

int **`SUNLinSolSetATimes`**(SUNLinearSolver *LS*, void* *A_data*, *ATimesFn ATimes*)
This function is *required for matrix-free linear solvers*; otherwise it is optional.

Provides a *`ATimesFn`* function pointer, as well as a `void*` pointer to a data structure used by this routine, to a linear solver object. SUNDIALS packages will call this function to set the matrix-vector product function to either a solver-provided difference-quotient via vector operations or a user-supplied solver-specific routine. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in section *SUNLinearSolver return codes*.

Usage:

```
retval = SUNLinSolSetATimes(LS, A_data, ATimes);
```

int **`SUNLinSolSetPreconditioner`**(SUNLinearSolver *LS*, void* *P_data*, *PSetupFn Pset*, *PSolveFn Psol*)
This *optional* routine provides *`PSetupFn`* and *`PSolveFn`* function pointers that implement the preconditioner solves P_1^{-1} and P_2^{-1}. This routine will be called by a SUNDIALS package, which will provide translation between the generic *Pset* and *Psol* calls and the package- or user-supplied routines. This routine should

return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in section *SUNLinearSolver return codes*.

Usage:

```
retval = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);
```

int **SUNLinSolSetScalingVectors**(SUNLinearSolver *LS*, N_Vector *s1*, N_Vector *s2*)

This *optional* routine provides left/right scaling vectors for the linear system solve. Here, *s1* and *s2* are `N_Vectors` of positive scale factors containing the diagonal of the matrices S_1 and S_2, respectively. Neither of these vectors need to be tested for positivity, and a `NULL` argument for either indicates that the corresponding scaling matrix is the identity. This routine should return zero for a successful call, and a negative value for a failure, ideally returning one of the generic error codes listed in section *SUNLinearSolver return codes*.

Usage:

```
retval = SUNLinSolSetScalingVectors(LS, s1, s2);
```

12.1.3 SUNLinearSolver get functions

The following get functions allow SUNDIALS packages to retrieve results from a linear solve. All routines are optional.

int **SUNLinSolNumIters**(SUNLinearSolver *LS*)

This *optional* routine should return the number of linear iterations performed in the last "solve" call.

Usage:

```
its = SUNLinSolNumIters(LS);
```

realtype **SUNLinSolResNorm**(SUNLinearSolver *LS*)

This *optional* routine should return the final residual norm from the last "solve" call.

Usage:

```
rnorm = SUNLinSolResNorm(LS);
```

N_Vector **SUNLinSolResid**(SUNLinearSolver *LS*)

If an iterative method computes the preconditioned initial residual and returns with a successful solve without performing any iterations (i.e., either the initial guess or the preconditioner is sufficiently accurate), then this *optional* routine may be called by the SUNDIALS package. This routine should return the `N_Vector` containing the preconditioned initial residual vector.

Usage:

```
rvec = SUNLinSolResid(LS);
```

Note: since `N_Vector` is actually a pointer, and the results are not modified, this routine should *not* require additional memory allocation. If the SUNLinSol object does not retain a vector for this purpose, then this function pointer should be set to `NULL` in the implementation.

sunindextype **SUNLinSolLastFlag**(SUNLinearSolver *LS*)

This *optional* routine should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS packages directly; it allows the user to investigate linear solver issues after a failed solve.

Usage:

```
lflag = SUNLinLastFlag(LS);
```

int **SUNLinSolSpace**(SUNLinearSolver *LS*, long int **lenrwLS*, long int **leniwLS*)

This *optional* routine should return the storage requirements for the linear solver *LS*. *lrw* is a `long int` containing the number of realtype words and *liw* is a `long int` containing the number of integer words. The return value is an integer flag denoting success/failure of the operation.

This function is advisory only, for use by users to help determine their total space requirements.

Usage:

```
retval = SUNLinSolSpace(LS, &lrw, &liw);
```

12.1.4 Functions provided by SUNDIALS packages

To interface with SUNLinSol modules, the SUNDIALS packages supply a variety of routines for evaluating the matrix-vector product, and setting up and applying the preconditioniner. These package-provided routines translate between the user-supplied ODE, DAE, or nonlinear systems and the generic interfaces to the linear systems of equations that result in their solution. The types for functions provided to a SUNLinSol module are defined in the header file `sundials/sundials_iterative.h`, and are described below.

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

These functions compute the action of a matrix on a vector, performing the operation $z = Av$. Memory for *z* will already be allocated prior to calling this function. The parameter *A_data* is a pointer to any information about A which the function needs in order to do its job. The vector v should be left unchanged. This routine should return 0 if successful and a non-zero value if unsuccessful.

typedef int **(*PSetupFn)**(void *P_data*)

These functions set up any requisite problem data in preparation for calls to the corresponding *PSolveFn*. This routine should return 0 if successful and a non-zero value if unsuccessful.

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

These functions solve the preconditioner equation $Pz = r$ for the vector z. Memory for *z* will already be allocated prior to calling this function. The parameter *P_data* is a pointer to any information about P which the function needs in order to do its job (set up by the corresponding *PSetupFn*). The parameter *lr* is input, and indicates whether P is to be taken as the left or right preconditioner: *lr* = 1 for left and *lr* = 2 for right. If preconditioning is on one side only, *lr* can be ignored. If the preconditioner is iterative, then it should strive to solve the preconditioner equation so that

$$\|Pz - r\|_{\text{wrms}} < tol$$

where the error weight vector for the WRMS norm may be accessed from the main package memory structure. The vector *r* should not be modified by the *PSolveFn*. This routine should return 0 if successful and a non-zero value if unsuccessful. On a failure, a negative return value indicates an unrecoverable condition, while a positive value indicates a recoverable one, in which the calling routine may reattempt the solution after updating preconditioner data.

12.1.5 SUNLinearSolver return codes

The functions provided to SUNLinSol modules by each SUNDIALS package, and functions within the SUNDIALS-provided SUNLinSol implementations utilize a common set of return codes, listed below. These adhere to a common pattern: 0 indicates success, a postitive value corresponds to a recoverable failure, and a negative value indicates a non-recoverable failure. Aside from this pattern, the actual values of each error code are primarily to provide additional information to the user in case of a linear solver failure.

- SUNLS_SUCCESS (0) – successful call or converged solve
- SUNLS_MEM_NULL (-801) – the memory argument to the function is NULL
- SUNLS_ILL_INPUT (-802) – an illegal input has been provided to the function
- SUNLS_MEM_FAIL (-803) – failed memory access or allocation
- SUNLS_ATIMES_NULL (-804) – the Atimes function is NULL
- SUNLS_ATIMES_FAIL_UNREC (-805) – an unrecoverable failure occurred in the ATimes routine
- SUNLS_PSET_FAIL_UNREC (-806) – an unrecoverable failure occurred in the Pset routine
- SUNLS_PSOLVE_NULL (-807) – the preconditioner solve function is NULL
- SUNLS_PSOLVE_FAIL_UNREC (-808) – an unrecoverable failure occurred in the Psolve routine
- SUNLS_PACKAGE_FAIL_UNREC (-809) – an unrecoverable failure occurred in an external linear solver package
- SUNLS_GS_FAIL (-810) – a failure occurred during Gram-Schmidt orthogonalization (SPGMR/SPFGMR)
- SUNLS_QRSOL_FAIL (-811) – a singular R matrix was encountered in a QR factorization (SPGMR/SPFGMR)
- SUNLS_VECTOROP_ERR (-812) – a vector operation error occurred
- SUNLS_RES_REDUCED (801) – an iterative solver reduced the residual, but did not converge to the desired tolerance
- SUNLS_CONV_FAIL (802) – an iterative solver did not converge (80and the residual was not reduced)
- SUNLS_ATIMES_FAIL_REC (803) – a recoverable failure occurred in the ATimes routine
- SUNLS_PSET_FAIL_REC (804) – a recoverable failure occurred in the Pset routine
- SUNLS_PSOLVE_FAIL_REC (805) – a recoverable failure occurred in the Psolve routine
- SUNLS_PACKAGE_FAIL_REC (806) – a recoverable failure occurred in an external linear solver package
- SUNLS_QRFACT_FAIL (807) – a singular matrix was encountered during a QR factorization (SPGMR/SPFGMR)
- SUNLS_LUFACT_FAIL (808) – a singular matrix was encountered during a LU factorization

12.1.6 The generic SUNLinearSolver module

SUNDIALS packages interact with specific SUNLinSol implementations through the generic SUNLinSol module on which all other SUNLinSol iplementations are built. The SUNLinearSolver type is a pointer to a structure containing an implementation-dependent *content* field, and an *ops* field. The type SUNLinearSolver is defined as

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

struct _generic_SUNLinearSolver {
  void *content;
  struct _generic_SUNLinearSolver_Ops *ops;
};
```

where the _generic_SUNLinearSolver_Ops structure is a list of pointers to the various actual linear solver operations provided by a specific implementation. The _generic_SUNLinearSolver_Ops structure is defined as

```
struct _generic_SUNLinearSolver_Ops {
  SUNLinearSolver_Type (*gettype)(SUNLinearSolver);
  SUNLinearSolver_ID   (*getid)(SUNLinearSolver);
  int                  (*setatimes)(SUNLinearSolver, void*, ATimesFn);
  int                  (*setpreconditioner)(SUNLinearSolver, void*,
                                            PSetupFn, PSolveFn);
  int                  (*setscalingvectors)(SUNLinearSolver,
                                            N_Vector, N_Vector);
  int                  (*initialize)(SUNLinearSolver);
  int                  (*setup)(SUNLinearSolver, SUNMatrix);
  int                  (*solve)(SUNLinearSolver, SUNMatrix, N_Vector,
                                N_Vector, realtype);
  int                  (*numiters)(SUNLinearSolver);
  realtype             (*resnorm)(SUNLinearSolver);
  sunindextype         (*lastflag)(SUNLinearSolver);
  int                  (*space)(SUNLinearSolver, long int*, long int*);
  N_Vector             (*resid)(SUNLinearSolver);
  int                  (*free)(SUNLinearSolver);
};
```

The generic SUNLinSol module defines and implements the linear solver operations defined in Sections *SUNLinearSolver core functions* through *SUNLinearSolver get functions*. These routines are in fact only wrappers to the linear solver operations defined by a particular SUNLinSol implementation, which are accessed through the *ops* field of the `SUNLinearSolver` structure. To illustrate this point we show below the implementation of a typical linear solver operation from the generic `SUNLinearSolver` module, namely `SUNLinSolInitialize`, which initializes a `SUNLinearSolver` object for use after it has been created and configured, and returns a flag denoting a successful or failed operation:

```
int SUNLinSolInitialize(SUNLinearSolver S)
{
  return ((int) S->ops->initialize(S));
}
```

12.1.7 Compatibility of SUNLinearSolver modules

We note that not all `SUNLinearSolver` types are compatible with all `SUNMatrix` and `N_Vector` types provided with SUNDIALS. In Table *Compatible SUNLinearSolver and SUNMatrix implementations* we show the matrix-based linear solvers available as `SUNLinearSolver` modules, and the compatible matrix implementations. Recall that Table *SUNDIALS linear solver interfaces and vector implementations that can be used for each* shows the compatibility between all `SUNLinearSolver` modules and vector implementations.

12.1.7.1 Compatible SUNLinearSolver and SUNMatrix implementations

Linear Solver	Dense	Banded	Sparse	User Supplied
Dense	X			X
LapackDense	X			X
Band		X		X
LapackBand		X		X
KLU			X	X
SuperLU_MT			X	X
User supplied	X	X	X	X

12.1.8 Implementing a custom SUNLinearSolver module

A particular implementation of the `SUNLinearSolver` module must:

- Specify the *content* field of the SUNLinSol module.
- Define and implement the required linear solver operations. See the section *ARKode SUNLinearSolver interface* to determine which SUNLinSol operations are required for this SUNDIALS package.

  Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNLinSol module (each with different `SUNLinearSolver` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free a `SUNLinearSolver` with the new *content* field and with *ops* pointing to the new linear solver operations.

We note that the function pointers for all unsupported optional routines should be set to `NULL` in the *ops* structure. This allows the SUNDIALS package that is using the SUNLinSol object to know that the associated functionality is not supported.

To aid in the creation of custom `SUNLinearSolver` modules the generic `SUNLinearSolver` module provides the utility function *`SUNLinSolNewEmpty()`*. When used in custom `SUNLinearSolver` constructors this function will ease the introduction of any new optional linear solver operations to the `SUNLinearSolver` API by ensuring only required operations need to be set.

SUNLinearSolver **`SUNLinSolNewEmpty`**()
This function allocates a new generic `SUNLinearSolver` object and initializes its content pointer and the function pointers in the operations structure to `NULL`.

Return value: If successful, this function returns a `SUNLinearSolver` object. If an error occurs when allocating the object, then this routine will return `NULL`.

void **`SUNLinSolFreeEmpty`**(SUNLinearSolver *LS*)
This routine frees the generic `SUNLinearSolver` object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is `NULL`, and, if it is not, it will free it as well.

Arguments:

- *LS* – a SUNLinearSolver object

Additionally, a `SUNLinearSolver` implementation *may* do the following:

- Define and implement additional user-callable "set" routines acting on the `SUNLinearSolver`, e.g., for setting various configuration options to tune the linear solver to a particular problem.
- Provide additional user-callable "get" routines acting on the `SUNLinearSolver` object, e.g., for returning various solve statistics.

12.1.8.1 Intended use cases

The SUNLinSol (and SUNMATRIX) APIs are designed to require a minimal set of routines to ease interfacing with custom or third-party linear solver libraries. External solvers provide similar routines with the necessary functionality and thus will require minimal effort to wrap within custom SUNMATRIX and SUNLinSol implementations. Sections *SUNMATRIX functions required by ARKode* and *ARKode SUNLinearSolver interface* include a list of the required set of routines that compatible SUNMATRIX and SUNLinSol implementations must provide. As SUNDIALS packages utilize generic SUNLinSol modules allowing for user-supplied `SUNLinearSolver` implementations, there exists a wide range of possible linear solver combinations. Some intended use cases for both the SUNDIALS-provided and user-supplied SUNLinSol modules are discussd in the following sections.

Direct linear solvers

Direct linear solver modules require a matrix and compute an 'exact' solution to the linear system *defined by the matrix*. Multiple matrix formats and associated direct linear solvers are supplied with SUNDIALS through different SUNMATRIX and SUNLinSol implementations. SUNDIALS packages strive to amortize the high cost of matrix construction by reusing matrix information for multiple nonlinear iterations. As a result, each package's linear solver interface recomputes Jacobian information as infrequently as possible.

Alternative matrix storage formats and compatible linear solvers that are not currently provided by or interfaced with SUNDIALS can leverage this infrastructure with minimal effort. To do so, a user must implement custom SUNMATRIX and SUNLinSol wrappers for the desired matrix format and/or linear solver following the APIs described in the sections *Matrix Data Structures* and *Description of the SUNLinearSolver module*. *This user-supplied SUNLinSol module must then self-identify as having* `SUNLINEARSOLVER_DIRECT` *type*.

Matrix-free iterative linear solvers

Matrix-free iterative linear solver modules do not require a matrix and compute an inexact solution to the linear system *defined by the package-supplied* `ATimes` *routine*. SUNDIALS supplies multiple scaled, preconditioned iterative linear solver (spils) SUNLinSol modules that support scaling to allow users to handle non-dimensionalization (as best as possible) within each SUNDIALS package and retain variables and define equations as desired in their applications. For linear solvers that do not support left/right scaling, the tolerance supplied to the linear solver is adjusted to compensate (see section *Iterative linear solver tolerance* for more details); however, this use case may be non-optimal and cannot handle situations where the magnitudes of different solution components or equations vary dramatically within a single problem.

To utilize alternative linear solvers that are not currently provided by or interfaced with SUNDIALS a user must implement a custom SUNLinSol wrapper for the linear solver following the API described in the section *Description of the SUNLinearSolver module*. *This user-supplied SUNLinSol module must then self-identify as having* `SUNLINEARSOLVER_ITERATIVE` *type*.

Matrix-based iterative linear solvers (reusing A)

Matrix-based iterative linear solver modules require a matrix and compute an inexact solution to the linear system *defined by the matrix*. This matrix will be updated infrequently and resued across multiple solves to amortize cost of matrix construction. As in the direct linear solver case, only wrappers for the matrix and linear solver in SUNMATRIX and SUNLinSol implementations need to be created to utilize a new linear solver. *This user-supplied SUNLinSol module must then self-identify as having* `SUNLINEARSOLVER_MATRIX_ITERATIVE` *type*.

At present, SUNDIALS has one example problem that uses this approach for wrapping a structured-grid matrix, linear solver, and preconditioner from the *hypre* library that may be used as a template for other customized implementations (see `examples/arkode/CXX_parhyp/ark_heat2D_hypre.cpp`).

Matrix-based iterative linear solvers (current A)

For users who wish to utilize a matrix-based iterative linear solver module where the matrix is *purely for preconditioning* and the linear system is *defined by the package-supplied* `ATimes` *routine*, we envision two current possibilities.

The preferred approach is for users to employ one of the SUNDIALS scaled, preconditioned iterative linear solver (spils) implementations (*`SUNLinSol_SPGMR()`*, *`SUNLinSol_SPFGMR()`*, *`SUNLinSol_SPBCGS()`*, *`SUNLinSol_SPTFQMR()`*, or *`SUNLinSol_PCG()`*) as the outer solver. The creation and storage of the preconditioner matrix, and interfacing with the corresponding linear solver, can be handled through a package's preconditioner 'setup' and 'solve' functionality (see the sections *Preconditioner setup (iterative linear solvers)* and *Precondi-*

tioner solve (iterative linear solvers), respectively) without creating SUNMATRIX and SUNLinSol implementations. This usage mode is recommended primarily because the SUNDIALS-provided spils modules support the scaling as described above.

A second approach supported by the linear solver APIs is as follows. If the SUNLinSol implementation is matrix-based, *self-identifies as having* `SUNLINEARSOLVER_ITERATIVE` *type*, and *also provides a non-NULL :c:func:'SUNLinSolSetATimes()' routine*, then each SUNDIALS package will call that routine to attach its package-specific matrix-vector product routine to the SUNLinSol object. The SUNDIALS package will then call the SUNLinSol-provided *SUNLinSolSetup()* routine (infrequently) to update matrix information, but will provide current matrix-vector products to the SUNLinSol implementation through the package-supplied `ATimesFn` routine.

12.2 ARKode SUNLinearSolver interface

In the table below, we list the SUNLinSol module linear solver functions used within the ARKLS interface. As with the SUNMATRIX module, we emphasize that the ARKode user does not need to know detailed usage of linear solver functions by the ARKode code modules in order to use ARKode. The information is presented as an implementation detail for the interested reader.

The linear solver functions listed below are marked with "X" to indicate that they are required, or with "O" to indicate that they are only called if they are non-`NULL` in the `SUNLinearSolver` implementation that is being used. Note:

1. *SUNLinSolNumIters()* is only used to accumulate overall iterative linear solver statistics. If it is not implemented by the `SUNLinearSolver` module, then ARKLS will consider all solves as requiring zero iterations.
2. Although *SUNLinSolResNorm()* is optional, if it is not implemented by the `SUNLinearSolver` then ARKLS will consider all solves a being *exact*.
3. Although ARKLS does not call *SUNLinSolLastFlag()* directly, this routine is available for users to query linear solver failure modes directly.
4. Although ARKLS does not call *SUNLinSolFree()* directly, this routine should be available for users to call when cleaning up from a simulation.

Routine	DIRECT	ITERATIVE	MATRIX_ITERATIVE
SUNLinSolGetType	X	X	X
SUNLinSolSetATimes	O	X	O
SUNLinSolSetPreconditioner	O	O	O
SUNLinSolSetScalingVectors	O	O	O
SUNLinSolInitialize	X	X	X
SUNLinSolSetup	X	X	X
SUNLinSolSolve	X	X	X
SUNLinSolNumIters[1]		O	O
SUNLinSolResNorm[2]		O	O
SUNLinSolLastFlag[3]			
SUNLinSolFree[4]			
SUNLinSolSpace	O	O	O

Since there are a wide range of potential SUNLinSol use cases, the following subsections describe some details of the ARKLS interface, in the case that interested users wish to develop custom SUNLinSol modules.

12.2.1 Lagged matrix information

If the SUNLinSol identifies as having type `SUNLINEARSOLVER_DIRECT` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`, then the SUNLinSol object solves a linear system *defined* by a SUNMATRIX object. ARKLS will update the matrix information infrequently according to the strategies outlined in the section *Updating the linear solver*. To this end, we differentiate between the *desired* linear system $\mathcal{A}x = b$ with $\mathcal{A} = (M - \gamma J)$ and the *actual* linear system

$$\tilde{\mathcal{A}}\tilde{x} = b \quad \Leftrightarrow \quad (M - \tilde{\gamma}J)\tilde{x} = b.$$

Since ARKLS updates the SUNMATRIX object infrequently, it is likely that $\gamma \neq \tilde{\gamma}$, and in turn $\mathcal{A} \neq \tilde{\mathcal{A}}$. Therefore, after calling the SUNLinSol-provided *`SUNLinSolSolve()`* routine, we test whether $\gamma/\tilde{\gamma} \neq 1$, and if this is the case we scale the solution $\tilde{x}$ to obtain the desired linear system solution x via

$$x = \frac{2}{1 + \gamma/\tilde{\gamma}}\tilde{x}. \tag{12.3}$$

The motivation for this selection of the scaling factor $c = 2/(1 + \gamma/\tilde{\gamma})$ follows the derivation in *[BBH1989]* and *[H2000]*. In short, if we consider a stationary iteration for the linear system as consisting of a solve with $\tilde{\mathcal{A}}$ followed by scaling by c, then for a linear constant-coefficient problem, the error in the solution vector will be reduced at each iteration by the error matrix $E = I - c\tilde{\mathcal{A}}^{-1}\mathcal{A}$, with a convergence rate given by the spectral radius of E. Assuming that stiff systems have a spectrum spread widely over the left half-plane, c is chosen to minimize the magnitude of the eigenvalues of E.

12.2.2 Iterative linear solver tolerance

If the SUNLinSol object self-identifies as having type `SUNLINEARSOLVER_ITERATIVE` or `SUNLINEARSOLVER_MATRIX_ITERATIVE`, then ARKLS will set the input tolerance `delta` as described in *Linear iteration error control*. However, if the iterative linear solver does not support scaling matrices (i.e., the *`SUNLinSolSetScalingVectors()`* routine is `NULL`), then ARKLS will attempt to adjust the linear solver tolerance to account for this lack of functionality. To this end, the following assumptions are made:

- All solution components have similar magnitude; hence the residual weight vector w used in the WRMS norm (see the section *Error norms*), corresponding to the left scaling matrix S_1, should satisfy the assumption

$$w_i \approx w_{mean}, \quad \text{for} \quad i = 0, \ldots, n-1.$$

- The SUNLinSol object uses a standard 2-norm to measure convergence.

Under these assumptions, ARKLS adjusts the linear solver convergence requirement as follows (using the notation from the beginning of this chapter):

$$\begin{aligned}
& \left\|\tilde{b} - \tilde{A}\tilde{x}\right\|_2 < \text{tol} \\
\Leftrightarrow \quad & \left\|S_1 P_1^{-1} b - S_1 P_1^{-1} A x\right\|_2 < \text{tol} \\
\Leftrightarrow \quad & \sum_{i=0}^{n-1} \left[w_i \left(P_1^{-1}(b - Ax)\right)_i\right]^2 < \text{tol}^2 \\
\Leftrightarrow \quad & w_{mean}^2 \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b - Ax)\right)_i\right]^2 < \text{tol}^2 \\
\Leftrightarrow \quad & \sum_{i=0}^{n-1} \left[\left(P_1^{-1}(b - Ax)\right)_i\right]^2 < \left(\frac{\text{tol}}{w_{mean}}\right)^2 \\
\Leftrightarrow \quad & \left\|P_1^{-1}(b - Ax)\right\|_2 < \frac{\text{tol}}{w_{mean}}
\end{aligned}$$

Therefore the tolerance scaling factor

$$w_{mean} = \|w\|_2/\sqrt{n}$$

is computed and the scaled tolerance `delta` $=$ tol$/w_{mean}$ is supplied to the SUNLinSol object.

12.2.3 Providing a custom SUNLinearSolver

In certain instances, users may wish to provide a custom SUNLinearSolver implementation to ARKode in order to leverage the structure of a problem. While the 'standard' API for these routines is typically sufficient for most users, others may need additional ARKode-specific information on top of what is provided. For these purposes, we note the following advanced ouptut functions available in ARKStep and MRIStep:

ARKStep advanced outputs: when solving the Newton nonlinear system of equations in predictor-corrector form,

$$\begin{aligned} G(z_{cor}) &\equiv z_{cor} - \gamma f^I\left(t^I_{n,i}, z_i\right) - \tilde{a}_i = 0 \qquad && [M = I], \\ G(z_{cor}) &\equiv M z_{cor} - \gamma f^I\left(t^I_{n,i}, z_i\right) - \tilde{a}_i = 0 \qquad && [M \text{ static}], \\ G(z_{cor}) &\equiv M(t^I_{n,i})(z_{cor} - \tilde{a}_i) - \gamma f^I\left(t^I_{n,i}, z_i\right) = 0 \qquad && [M \text{ time-dependent}]. \end{aligned}$$

- `ARKStepGetCurrentTime()` – when called within the computation of a step (i.e., within a solve) this returns $t^I_{n,i}$. Otherwise the current internal solution time is returned.
- `ARKStepGetCurrentState()` – when called within the computation of a step (i.e., within a solve) this returns the current stage vector $z_i = z_{cor} + z_{pred}$. Otherwise the current internal solution is returned.
- `ARKStepGetCurrentGamma()` – returns γ.
- `ARKStepGetCurrentMassMatrix()` – returns $M(t)$.
- `ARKStepGetNonlinearSystemData()` – returns z_i, z_{pred}, $f^I(t^I_{n,i}, y_{cur})$, $\tilde{a}_i$, and γ.

MRIStep advanced outputs: when solving the Newton nonlinear system of equations in predictor-corrector form,

$$G(z_{cor}) \equiv z_{cor} - \gamma f^S\left(t^S_{n,i}, z_i\right) - \tilde{a}_i = 0$$

- `MRIStepGetCurrentTime()` – when called within the computation of a step (i.e., within a solve) this returns $t^S_{n,i}$. Otherwise the current internal solution time is returned.
- `MRIStepGetCurrentState()` – when called within the computation of a step (i.e., within a solve) this returns the current stage vector $z_i = z_{cor} + z_{pred}$. Otherwise the current internal solution is returned.
- `MRIStepGetCurrentGamma()` – returns γ.
- `MRIStepGetNonlinearSystemData()` – returns z_i, z_{pred}, $f^I(t^I_{n,i}, y_{cur})$, $\tilde{a}_i$, and γ.

12.3 The SUNLinSol_Dense Module

The dense implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_Dense, is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS).

12.3.1 SUNLinSol_Dense Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_dense.h`. The SUNLinSol_Dense module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsoldense` module library.

The module SUNLinSol_Dense provides the following user-callable constructor routine:

SUNLinearSolver **SUNLinSol_Dense**(N_Vector *y*, SUNMatrix *A*)
: This function creates and allocates memory for a dense `SUNLinearSolver`. Its arguments are an `N_Vector` and `SUNMatrix`, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

For backwards compatibility, we also provide the wrapper function,

SUNLinearSolver **SUNDenseLinearSolver**(N_Vector *y*, SUNMatrix *A*)
: Wrapper function for *SUNLinSol_Dense()*, with identical input and output arguments

For solvers that include a Fortran interface module, the SUNLinSol_Dense module also includes the Fortran-callable function *FSUNDenseLinSolInit()* to initialize this SUNLinSol_Dense module for a given SUNDIALS solver.

subroutine FSUNDenseLinSolInit(*CODE*, *IER*)
: Initializes a dense `SUNLinearSolver` structure for use in a SUNDIALS package.

  This routine must be called *after* both the `N_Vector` and `SUNMatrix` objects have been initialized.

  Arguments:

  - *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
  - *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassDenseLinSolInit()* initializes this SUNLinSol_Dense module for solving mass matrix linear systems.

subroutine FSUNMassDenseLinSolInit(*IER*)
: Initializes a dense `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

  This routine must be called *after* both the `N_Vector` and `SUNMatrix` objects have been initialized.

  Arguments:

  - *IER* (`int`, output) – return flag (0 success, -1 for failure).

12.3.2 SUNLinSol_Dense Description

The SUNLinSol_Dense module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
  sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- `N` - size of the linear system,
- `pivots` - index array for partial pivoting in LU factorization,

- `last_flag` - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The SUNLinSol_Dense module defines dense implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_Dense`
- `SUNLinSolInitialize_Dense` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Dense` – this performs the LU factorization.
- `SUNLinSolSolve_Dense` – this uses the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_Dense`
- `SUNLinSolSpace_Dense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_Dense`

12.4 The SUNLinSol_Band Module

The band implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_Band, is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS).

12.4.1 SUNLinSol_Band Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_band.h`. The SUNLinSol_Band module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolband` module library.

The module SUNLinSol_Band provides the following user-callable constructor routine:

SUNLinearSolver **`SUNLinSol_Band`**(N_Vector *y*, SUNMatrix *A*)
: This function creates and allocates memory for a band `SUNLinearSolver`. Its arguments are an `N_Vector` and `SUNMatrix`, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  Additionally, this routine will verify that the input matrix `A` is allocated with appropriate upper bandwidth storage for the LU factorization.

If either A or y are incompatible then this routine will return NULL.

For backwards compatibility, we also provide the wrapper function,

SUNLinearSolver **SUNBandLinearSolver**(N_Vector *y*, SUNMatrix *A*)
Wrapper function for *SUNLinSol_Band()*, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLinSol_Band module also includes the Fortran-callable function *FSUNBandLinSolInit()* to initialize this SUNLinSol_Band module for a given SUNDIALS solver.

subroutine FSUNBandLinSolInit(*CODE*, *IER*)
Initializes a banded SUNLinearSolver structure for use in a SUNDIALS package.

This routine must be called *after* both the N_Vector and SUNMatrix objects have been initialized.

Arguments:

- *CODE* (int, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *IER* (int, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassBandLinSolInit()* initializes this SUNLinSol_Band module for solving mass matrix linear systems.

subroutine FSUNMassBandLinSolInit(*IER*)
Initializes a banded SUNLinearSolver structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* both the N_Vector and SUNMatrix objects have been initialized.

Arguments:

- *IER* (int, output) – return flag (0 success, -1 for failure).

12.4.2 SUNLinSol_Band Description

The SUNLinSol_Band module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
  sunindextype N;
  sunindextype *pivots;
  sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- N - size of the linear system,
- pivots - index array for partial pivoting in LU factorization,
- last_flag - last error return flag from internal function evaluations.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the pivots array.
- The "solve" call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX_BAND object.

- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth `mu` and lower bandwidth `ml`, then the upper triangular factor U can have upper bandwidth as big as `smu = MIN(N-1,mu+ml)`. The lower triangular factor L has lower bandwidth `ml`.

The SUNLinSol_Band module defines band implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_Band`
- `SUNLinSolInitialize_Band` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_Band` – this performs the LU factorization.
- `SUNLinSolSolve_Band` – this uses the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_Band`
- `SUNLinSolSpace_Band` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_Band`

12.5 The SUNLinSol_LapackDense Module

The LAPACK dense implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_LapackDense, is designed to be used with the corresponding SUNMATRIX_DENSE matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS).

12.5.1 SUNLinSol_LapackDense Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_lapackdense.h`. The installed module library to link to is `libsundials_sunlinsollapackdense.`*lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_LapackDense provides the following additional user-callable constructor routine:

SUNLinearSolver **`SUNLinSol_LapackDense`**(N_Vector *y*, SUNMatrix *A*)
: This function creates and allocates memory for a LAPACK dense `SUNLinearSolver`. Its arguments are an `N_Vector` and `SUNMatrix`, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_DENSE matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  If either `A` or `y` are incompatible then this routine will return `NULL`.

For backwards compatibility, we also provide the wrapper function,

SUNLinearSolver **`SUNLapackDense`**(N_Vector *y*, SUNMatrix *A*)
: Wrapper function for *`SUNLinSol_LapackDense()`*, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLinSol_LapackDense module also includes the Fortran-callable function *FSUNLapackDenseInit()* to initialize this SUNLinSol_LapackDense module for a given SUNDIALS solver.

subroutine FSUNLapackDenseInit(*CODE*, *IER*)
Initializes a dense LAPACK SUNLinearSolver structure for use in a SUNDIALS package.

This routine must be called *after* both the N_Vector and SUNMatrix objects have been initialized.

Arguments:

- *CODE* (int, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *IER* (int, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassLapackDenseInit()* initializes this SUNLinSol_LapackDense module for solving mass matrix linear systems.

subroutine FSUNMassLapackDenseInit(*IER*)
Initializes a dense LAPACK SUNLinearSolver structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* both the N_Vector and SUNMatrix objects have been initialized.

Arguments:

- *IER* (int, output) – return flag (0 success, -1 for failure).

12.5.2 SUNLinSol_LapackDense Description

The SUNLinSol_LapackDense module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
  sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- N - size of the linear system,
- pivots - index array for partial pivoting in LU factorization,
- last_flag - last error return flag from internal function evaluations.

The SUNLinSol_LapackDense module is a SUNLinearSolver wrapper for the LAPACK dense matrix factorization and solve routines, *GETRF and *GETRS, where * is either D or S, depending on whether SUNDIALS was configured to have realtype set to double or single, respectively (see section *Data Types* for details). In order to use the SUNLinSol_LapackDense module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see section *Working with external Libraries* for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using extended precision for realtype. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLinSol_LapackDense module also cannot be compiled when using int64_t for the sunindextype.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting ($\mathcal{O}(N^3)$ cost), $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_DENSE object A, with pivoting information encoding P stored in the `pivots` array.
- The "solve" call performs pivoting and forward and backward substitution using the stored `pivots` array and the LU factors held in the SUNMATRIX_DENSE object ($\mathcal{O}(N^2)$ cost).

The SUNLinSol_LapackDense module defines dense implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_LapackDense`
- `SUNLinSolInitialize_LapackDense` – this does nothing, since all consistency checks are performed at solver creation.
- `SUNLinSolSetup_LapackDense` – this calls either `DGETRF` or `SGETRF` to perform the LU factorization.
- `SUNLinSolSolve_LapackDense` – this calls either `DGETRS` or `SGETRS` to use the LU factors and `pivots` array to perform the solve.
- `SUNLinSolLastFlag_LapackDense`
- `SUNLinSolSpace_LapackDense` – this only returns information for the storage *within* the solver object, i.e. storage for `N`, `last_flag`, and `pivots`.
- `SUNLinSolFree_LapackDense`

12.6 The SUNLinSol_LapackBand Module

The LAPACK band implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_LapackBand, is designed to be used with the corresponding SUNMATRIX_BAND matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). The

12.6.1 SUNLinSol_LapackBand Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_lapackband.h`. The installed module library to link to is `libsundials_sunlinsollapackband.`*lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_LapackBand provides the following user-callable routine:

SUNLinearSolver **`SUNLinSol_LapackBand`**(N_Vector *y*, SUNMatrix *A*)
: This function creates and allocates memory for a LAPACK band `SUNLinearSolver`. Its arguments are an `N_Vector` and `SUNMatrix`, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

  This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_BAND matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

  Additionally, this routine will verify that the input matrix `A` is allocated with appropriate upper bandwidth storage for the LU factorization.

If either A or y are incompatible then this routine will return NULL.

For backwards compatibility, we also provide the wrapper function,

SUNLinearSolver **SUNLapackBand**(N_Vector *y*, SUNMatrix *A*)
Wrapper function for *SUNLinSol_LapackBand()*, with identical input and output arguments.

For solvers that include a Fortran interface module, the SUNLinSol_LapackBand module also includes the Fortran-callable function *FSUNLapackBandInit()* to initialize this SUNLinSol_LapackBand module for a given SUNDIALS solver.

subroutine FSUNLapackBandInit(*CODE*, *IER*)
Initializes a banded LAPACK SUNLinearSolver structure for use in a SUNDIALS package.

This routine must be called *after* both the N_Vector and SUNMatrix objects have been initialized.

Arguments:

- *CODE* (int, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *IER* (int, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassLapackBandInit()* initializes this SUNLinSol_LapackBand module for solving mass matrix linear systems.

subroutine FSUNMassLapackBandInit(*IER*)
Initializes a banded LAPACK SUNLinearSolver structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* both the N_Vector and SUNMatrix objects have been initialized.

Arguments:

- *IER* (int, output) – return flag (0 success, -1 for failure).

12.6.2 SUNLinSol_LapackBand Description

SUNLinSol_LapackBand module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
  sunindextype N;
  sunindextype *pivots;
  sunindextype last_flag;
};
```

These entries of the *content* field contain the following information:

- N - size of the linear system,
- pivots - index array for partial pivoting in LU factorization,
- last_flag - last error return flag from internal function evaluations.

The SUNLinSol_LapackBand module is a SUNLinearSolver wrapper for the LAPACK band matrix factorization and solve routines, *GBTRF and *GBTRS, where * is either D or S, depending on whether SUNDIALS was configured to have realtype set to double or single, respectively (see section *Data Types* for details). In order to use the SUNLinSol_LapackBand module it is assumed that LAPACK has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with LAPACK (see section *Working with external Libraries* for details). We note that since there do not exist 128-bit floating-point factorization and solve routines in LAPACK, this interface cannot be compiled when using extended precision for

realtype. Similarly, since there do not exist 64-bit integer LAPACK routines, the SUNLinSol_LapackBand module also cannot be compiled when using int64_t for the sunindextype.

This solver is constructed to perform the following operations:

- The "setup" call performs a LU factorization with partial (row) pivoting, $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix with 1's on the diagonal, and U is an upper triangular matrix. This factorization is stored in-place on the input SUNMATRIX_BAND object A, with pivoting information encoding P stored in the pivots array.
- The "solve" call performs pivoting and forward and backward substitution using the stored pivots array and the LU factors held in the SUNMATRIX_BAND object.
- A must be allocated to accommodate the increase in upper bandwidth that occurs during factorization. More precisely, if A is a band matrix with upper bandwidth mu and lower bandwidth ml, then the upper triangular factor U can have upper bandwidth as big as smu = MIN(N-1,mu+ml). The lower triangular factor L has lower bandwidth ml.

The SUNLinSol_LapackBand module defines band implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- SUNLinSolGetType_LapackBand
- SUNLinSolInitialize_LapackBand – this does nothing, since all consistency checks are performed at solver creation.
- SUNLinSolSetup_LapackBand – this calls either DGBTRF or SGBTRF to perform the LU factorization.
- SUNLinSolSolve_LapackBand – this calls either DGBTRS or SGBTRS to use the LU factors and pivots array to perform the solve.
- SUNLinSolLastFlag_LapackBand
- SUNLinSolSpace_LapackBand – this only returns information for the storage *within* the solver object, i.e. storage for N, last_flag, and pivots.
- SUNLinSolFree_LapackBand

12.7 The SUNLinSol_KLU Module

The KLU implementation of the SUNLinearSolver module provided with SUNDIALS, SUNLinSol_KLU, is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory N_Vector implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS).

12.7.1 SUNLinSol_KLU Usage

The header file to be included when using this module is sunlinsol/sunlinsol_klu.h. The installed module library to link to is libsundials_sunlinsolklu *.lib* where *.lib* is typically .so for shared libraries and .a for static libraries.

The module SUNLinSol_KLU provides the following additional user-callable routines:

SUNLinearSolver **SUNLinSol_KLU**(N_Vector *y*, SUNMatrix *A*)
: This constructor function creates and allocates memory for a SUNLinSol_KLU object. Its arguments are an N_Vector and SUNMatrix, that it uses to determine the linear system size and to assess compatibility with the linear solver implementation.

This routine will perform consistency checks to ensure that it is called with consistent N_Vector and SUNMatrix implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

If either A or y are incompatible then this routine will return NULL.

int **SUNLinSol_KLUReInit**(SUNLinearSolver *S*, SUNMatrix *A*, sunindextype *nnz*, int *reinit_type*)
This function reinitializes memory and flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeroes has changed or if the structure of the linear system has changed which would require a new symbolic (and numeric factorization).

The reinit_type argument governs the level of reinitialization. The allowed values are:

1. The Jacobian matrix will be destroyed and a new one will be allocated based on the nnz value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.
2. Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of nnz given in the sparse matrix provided to the original constructor routine (or the previous SUNKLUReInit call).

This routine assumes no other changes to solver use are necessary.

The return values from this function are SUNLS_MEM_NULL (either S or A are NULL), SUNLS_ILL_INPUT (A does not have type SUNMATRIX_SPARSE or reinit_type is invalid), SUNLS_MEM_FAIL (reallocation of the sparse matrix failed) or SUNLS_SUCCESS.

int **SUNLinSol_KLUSetOrdering**(SUNLinearSolver *S*, int *ordering_choice*)
This function sets the ordering used by KLU for reducing fill in the linear solve. Options for ordering_choice are:

0. AMD,
1. COLAMD, and
2. the natural ordering.

The default is 1 for COLAMD.

The return values from this function are SUNLS_MEM_NULL (S is NULL), SUNLS_ILL_INPUT (invalid ordering_choice), or SUNLS_SUCCESS.

sun_klu_symbolic* **SUNLinSol_KLUGetSymbolic**(SUNLinearSolver *S*)
This function returns a pointer to the KLU symbolic factorization stored in the SUNLinSol_KLU content structure.

When SUNDIALS is compiled with 32-bit indices (SUNDIALS_INDEX_SIZE=32), sun_klu_symbolic is mapped to the KLU type klu_symbolic; when SUNDIALS compiled with 64-bit indices (SUNDIALS_INDEX_SIZE=64) this is mapped to the KLU type klu_l_symbolic.

sun_klu_numeric* **SUNLinSol_KLUGetNumeric**(SUNLinearSolver *S*)
This function returns a pointer to the KLU numeric factorization stored in the SUNLinSol_KLU content structure.

When SUNDIALS is compiled with 32-bit indices (SUNDIALS_INDEX_SIZE=32), sun_klu_numeric is mapped to the KLU type klu_numeric; when SUNDIALS is compiled with 64-bit indices (SUNDIALS_INDEX_SIZE=64) this is mapped to the KLU type klu_l_numeric.

sun_klu_common* **SUNLinSol_KLUGetCommon**(SUNLinearSolver *S*)
This function returns a pointer to the KLU common structure stored in the SUNLinSol_KLU content structure.

When SUNDIALS is compiled with 32-bit indices (`SUNDIALS_INDEX_SIZE=32`), `sun_klu_common` is mapped to the KLU type `klu_common`; when SUNDIALS is compiled with 64-bit indices (`SUNDIALS_INDEX_SIZE=64`) this is mapped to the KLU type `klu_l_common`.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNKLU**(N_Vector *y*, SUNMatrix *A*)
Wrapper function for *SUNLinSol_KLU()*

int **SUNKLUReInit**(SUNLinearSolver *S*, SUNMatrix *A*, sunindextype *nnz*, int *reinit_type*)
Wrapper function for *SUNLinSol_KLUReInit()*

int **SUNKLUSetOrdering**(SUNLinearSolver *S*, int *ordering_choice*)
Wrapper function for *SUNLinSol_KLUSetOrdering()*

For solvers that include a Fortran interface module, the SUNLinSol_KLU module also includes the Fortran-callable function *FSUNKLUInit()* to initialize this SUNLinSol_KLU module for a given SUNDIALS solver.

subroutine FSUNKLUInit(*CODE*, *IER*)
Initializes a KLU sparse `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* both the `N_Vector` and `SUNMatrix` objects have been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassKLUInit()* initializes this SUNLinSol_KLU module for solving mass matrix linear systems.

subroutine FSUNMassKLUInit(*IER*)
Initializes a KLU sparse `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* both the `N_Vector` and `SUNMatrix` objects have been initialized.

Arguments:

- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_KLUReInit()* and *SUNLinSol_KLUSetOrdering()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNKLUReInit(*CODE*, *NNZ*, *REINIT_TYPE*, *IER*)
Fortran interface to *SUNLinSol_KLUReInit()* for system linear solvers.

This routine must be called *after* *FSUNKLUInit()* has been called.

Arguments: *NNZ* should have type `long int`, all others should have type `int`; all arguments have meanings identical to those listed above.

subroutine FSUNMassKLUReInit(*NNZ*, *REINIT_TYPE*, *IER*)
Fortran interface to *SUNLinSol_KLUReInit()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassKLUInit()* has been called.

Arguments: *NNZ* should have type `long int`, all others should have type `int`; all arguments have meanings identical to those listed above.

subroutine FSUNKLUSetOrdering(*CODE*, *ORDERING*, *IER*)
Fortran interface to *SUNLinSol_KLUSetOrdering()* for system linear solvers.

This routine must be called *after* *FSUNKLUInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassKLUSetOrdering(*ORDERING*, *IER*)
Fortran interface to *SUNLinSol_KLUSetOrdering()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassKLUInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

12.7.2 SUNLinSol_KLU Description

The SUNLinSol_KLU module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_KLU {
  int              last_flag;
  int              first_factorize;
  sun_klu_symbolic *symbolic;
  sun_klu_numeric  *numeric;
  sun_klu_common   common;
  sunindextype     (*klu_solver)(sun_klu_symbolic*, sun_klu_numeric*,
                                 sunindextype, sunindextype,
                                 double*, sun_klu_common*);
};
```

These entries of the *content* field contain the following information:

- `last_flag` - last error return flag from internal function evaluations,
- `first_factorize` - flag indicating whether the factorization has ever been performed,
- `Symbolic` - KLU storage structure for symbolic factorization components, with underlying type `klu_symbolic` or `klu_l_symbolic`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,
- `Numeric` - KLU storage structure for numeric factorization components, with underlying type `klu_numeric` or `klu_l_numeric`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,
- `Common` - storage structure for common KLU solver components, with underlying type `klu_common` or `klu_l_common`, depending on whether SUNDIALS was installed with 32-bit versus 64-bit indices, respectively,
- `klu_solver` – pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix, and on whether SUNDIALS was installed with 32-bit or 64-bit indices).

The SUNLinSol_KLU module is a `SUNLinearSolver` wrapper for the KLU sparse matrix factorization and solver library written by Tim Davis (*[KLU]*, *[DP2010]*). In order to use the SUNLinSol_KLU interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see section *Working with external Libraries* for details). Additionally, this wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `realtype` set to either `extended` or `single` (see section *Data Types* for details). Since the KLU library supports both 32-bit and 64-bit integers, this interface will be compiled for either of the available `sunindextype` options.

The KLU library has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Of these ordering choices, the default value in the SUNLinSol_KLU module is the COLAMD ordering.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLinSol_KLU module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, it calls the appropriate KLU "refactor" routine, followed by estimates of the numerical conditioning using the relevant "rcond", and if necessary "condest", routine(s). If these estimates of the condition number are larger than $\varepsilon^{-2/3}$ (where ε is the double-precision unit roundoff), then a new factorization is performed.
- The module includes the routine `SUNKLUReInit`, that can be called by the user to force a full refactorization at the next "setup" call.
- The "solve" call performs pivoting and forward and backward substitution using the stored KLU data structures. We note that in this solve KLU operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLinSol_KLU module defines implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_KLU`
- `SUNLinSolInitialize_KLU` – this sets the `first_factorize` flag to 1, forcing both symbolic and numerical factorizations on the subsequent "setup" call.
- `SUNLinSolSetup_KLU` – this performs either a LU factorization or refactorization of the input matrix.
- `SUNLinSolSolve_KLU` – this calls the appropriate KLU solve routine to utilize the LU factors to solve the linear system.
- `SUNLinSolLastFlag_KLU`
- `SUNLinSolSpace_KLU` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the KLU documentation.
- `SUNLinSolFree_KLU`

12.8 The SUNLinSol_SuperLUDIST Module

The SuperLU_DIST implementation of the SUNLinearSolver module provided with SUNDIALS, SUNLinsol_SuperLUDIST, is designed to be used with the SUNMatrix_SLUNRloc SUNMatrix, and one of the serial, threaded or parallel N_Vector implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, NVECTOR_PTHREADS, NVECTOR_PARALLEL, NVECTOR_PARHYP).

12.8.1 SUNLinSol_SuperLUDIST Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_superludist.h`. The installed module library to link to is `libsundials_sunlinsolsuperludist` *.lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_SuperLUDIST provides the following user-callable routines:

.. warning: Starting with SuperLU_DIST version 6.3.0, some structures were

renamed to have a prefix for the floating point type. The double precision API functions have the prefix 'd'. To maintain backwards compatibility with the unprefixed types, SUNDIALS provides macros to these SuperLU_DIST types with an 'x' prefix that expand to the correct prefix. E.g., the SUNDIALS macro `xLUstruct_t` expands to `dLUstruct_t` or `LUstruct_t` based on the SuperLU_DIST version.

SUNLinearSolver **SUNLinSol_SuperLUDIST**(N_Vector *y*, SuperMatrix **A*, gridinfo_t **grid*, xLUstruct_t **lu*, xScalePermstruct_t **scaleperm*, xSOLVEstruct_t **solve*, SuperLUStat_t **stat*, superlu_dist_options_t **options*)

This constructor function creates and allocates memory for a SUNLinSol_SuperLUDIST object. Its arguments are an `N_Vector`, a `SUNMatrix`, and SuperLU_DIST `gridinfo_t*`, `LUstuct_t*`, `xScalePermstruct_t*`, `xSOLVEstruct_t*`, `SuperLUStat_t*`, and `superlu_dist_options_t*` pointers. This routine analyzes the input matrix and vector to determine the linear system size and to assess the compatibility with the SuperLU_DIST library.

This routine will perform consistency checks to ensure that it is called with consistent N_Vector and SUNMatrix implementations. These are currently limited to the SUNMatrix_SLUNRloc matrix type and the NVECTOR_SERIAL, NVECTOR_OPENMP, NVECTOR_PTHREADS, NVECTOR_PARALLEL, and NVECTOR_PARHYP vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

The `grid`, `lu`, `scaleperm`, `solve`, and `options` arguments are not checked and are passed directly to SuperLU_DIST routines.

Some struct members of the `options` argument are modified internally by the SUNLinSol_SuperLUDIST solver. Specifically, the member `Fact` is modified in the setup and solve routines.

realtype **SUNLinSol_SuperLUDIST_GetBerr**(SUNLinearSolver *LS*)

This function returns the componentwise relative backward error of the computed solution. It takes one argument, the `SUNLinearSolver` object. The return type is `realtype`.

gridinfo_t* **SUNLinSol_SuperLUDIST_GetGridinfo**(SUNLinearSolver *LS*)

This function returns a pointer to the SuperLU_DIST structure that contains the 2D process grid. It takes one argument, the `SUNLinearSolver` object.

xLUstruct_t* **SUNLinSol_SuperLUDIST_GetLUstruct**(SUNLinearSolver *LS*)

This function returns a pointer to the SuperLU_DIST structure that contains the distributed `L` and `U` structures. It takes one argument, the `SUNLinearSolver` object.

superlu_dist_options_t* **SUNLinSol_SuperLUDIST_GetSuperLUOptions**(SUNLinearSolver *LS*)

This function returns a pointer to the SuperLU_DIST structure that contains the options which control how the linear system is factorized and solved. It takes one argument, the `SUNLinearSolver` object.

xScalePermstruct_t* **SUNLinSol_SuperLUDIST_GetScalePermstruct**(SUNLinearSolver *LS*)

This function returns a pointer to the SuperLU_DIST structure that contains the vectors that describe the transformations done to the matrix `A`. It takes one argument, the `SUNLinearSolver` object.

xSOLVEstruct_t* **SUNLinSol_SuperLUDIST_GetSOLVEstruct**(SUNLinearSolver *LS*)

This function returns a pointer to the SuperLU_DIST structure that contains information for communication during the solution phase. It takes one argument the `SUNLinearSolver` object.

SuperLUStat_t* **SUNLinSol_SuperLUDIST_GetSuperLUStat**(SUNLinearSolver *LS*)

This function returns a pointer to the SuperLU_DIST structure that stores information about runtime and flop count. It takes one argument, the `SUNLinearSolver` object.

12.8.2 SUNLinSol_SuperLUDIST Description

The SUNLinSol_SuperLUDIST module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUDIST {
  booleantype             first_factorize;
  int                     last_flag;
  realtype                berr;
  gridinfo_t              *grid;
  xLUstruct_t             *lu;
  superlu_dist_options_t  *options;
  xScalePermstruct_t      *scaleperm;
  xSOLVEstruct_t          *solve;
  SuperLUStat_t           *stat;
  sunindextype            N;
};
```

These entries of the *content* field contain the following information:

- `first_factorize` – flag indicating whether the factorization has ever been performed,
- `last_flag` – last error return flag from internal function evaluations,
- `berr` – the componentwise relative backward error of the computed solution,
- `grid` – pointer to the SuperLU_DIST structure that strores the 2D process grid
- `lu` – pointer to the SuperLU_DIST structure that stores the distributed `L` and `U` factors,
- `scaleperm` – pointer to the SuperLU_DIST structure that stores vectors describing the transformations done to the matrix `A`,
- `options` – pointer to the SuperLU_DIST stucture which contains options that control how the linear system is factorized and solved,
- `solve` – pointer to the SuperLU_DIST solve structure,
- `stat` – pointer to the SuperLU_DIST structure that stores information about runtime and flop count,
- `N` – the number of equations in the system.

The SUNLinSol_SuperLUDIST module is a SUNLinearSolver adapter for the SuperLU_DIST sparse matrix factorization and solver library written by X. Sherry Li (*[SuperLUDIST]*, *[GDL2007]*, *[LD2003]*, *[SLUUG1999]*). The package uses a SPMD parallel programming model and multithreading to enhance efficiency in distributed-memory parallel environments with multicore nodes and possibly GPU accelerators. It uses MPI for communication, OpenMP for threading, and CUDA for GPU support. In order to use the SUNLinSol_SuperLUDIST interface to SuperLU_DIST, it is assumed that SuperLU_DIST has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SuperLU_DIST (see Appendix *Working with external Libraries* for details). Additionally, the wrapper only supports double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to use single or extended precision. Moreover, since the SuperLU_DIST library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SuperLU_DIST library is installed using the same integer size as SUNDIALS.

The SuperLU_DIST library provides many options to control how a linear system will be factorized and solved. These options may be set by a user on an instance of the `superlu_dist_options_t` struct, and then it may be provided as an argument to the SUNLinSol_SuperLUDIST constructor. The SUNLinSol_SuperLUDIST module will respect all options set except for `Fact` – this option is necessarily modified by the SUNLinSol_SuperLUDIST module in the setup and solve routines.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLinSol_SuperLUDIST module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it sets the SuperLU_DIST option `Fact` to `DOFACT` so that a subsequent call to the "solve" routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to solve the system.
- On subsequent calls to the "setup" routine, it sets the SuperLU_DIST option `Fact` to `SamePattern` so that a subsequent call to "solve" will perform factorization assuming the same sparsity pattern as prior, i.e. it will reuse the column permutation vector.
- If "setup" is called prior to the "solve" routine, then the "solve" routine will perform a symbolic factorization, followed by an initial numerical factorization before continuing to the sparse triangular solves, and, potentially, iterative refinement. If "setup" is not called prior, "solve" will skip to the triangular solve step. We note that in this solve SuperLU_DIST operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLinSol_SuperLUDIST module defines implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_SuperLUDIST`
- `SUNLinSolInitialize_SuperLUDIST` – this sets the `first_factorize` flag to 1 and resets the internal SuperLU_DIST statistics variables.
- `SUNLinSolSetup_SuperLUDIST` – this sets the appropriate SuperLU_DIST options so that a subsequent solve will perform a symbolic and numerical factorization before proceeding with the triangular solves
- `SUNLinSolSolve_SuperLUDIST` – this calls the SuperLU_DIST solve routine to perform factorization (if the setup routine was called prior) and then use the LU factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUDIST`
- `SUNLinSolSpace_SuperLUDIST` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the SuperLU_DIST documentation.
- `SUNLinSolFree_SuperLUDIST`

12.9 The SUNLinSol_SuperLUMT Module

The SuperLU_MT implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_SuperLUMT, is designed to be used with the corresponding SUNMATRIX_SPARSE matrix type, and one of the serial or shared-memory `N_Vector` implementations (NVECTOR_SERIAL, NVECTOR_OPENMP, or NVECTOR_PTHREADS). While these are compatible, it is not recommended to use a threaded vector module with SUNLinSol_SuperLUMT unless it is the NVECTOR_OPENMP module and the SuperLU_MT library has also been compiled with OpenMP.

12.9.1 SUNLinSol_SuperLUMT Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_superlumt.h`. The installed module library to link to is `libsundials_sunlinsolsuperlumt` *.lib* where *.lib* is typically `.so` for shared libraries and `.a` for static libraries.

The module SUNLinSol_SuperLUMT provides the following user-callable routines:

SUNLinearSolver **`SUNLinSol_SuperLUMT`**(N_Vector *y*, SUNMatrix *A*, int *num_threads*)
: This constructor function creates and allocates memory for a SUNLinSol_SuperLUMT object. Its arguments are an `N_Vector`, a `SUNMatrix`, and a desired number of threads (OpenMP or Pthreads, depending on how

SuperLU_MT was installed) to use during the factorization steps. This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the SuperLU_MT library.

This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_SPARSE matrix type (using either CSR or CSC storage formats) and the NVECTOR_SERIAL, NVECTOR_OPENMP, and NVECTOR_PTHREADS vector types. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

If either `A` or `y` are incompatible then this routine will return `NULL`. The `num_threads` argument is not checked and is passed directly to SuperLU_MT routines.

int **SUNLinSol_SuperLUMTSetOrdering**(SUNLinearSolver *S*, int *ordering_choice*)

This function sets the ordering used by SuperLU_MT for reducing fill in the linear solve. Options for `ordering_choice` are:

0. natural ordering
1. minimal degree ordering on $A^T A$
2. minimal degree ordering on $A^T + A$
3. COLAMD ordering for unsymmetric matrices

The default is 3 for COLAMD.

The return values from this function are `SUNLS_MEM_NULL` (`S` is `NULL`), `SUNLS_ILL_INPUT` (invalid `ordering_choice`), or `SUNLS_SUCCESS`.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNSuperLUMT**(N_Vector *y*, SUNMatrix *A*, int *num_threads*)

Wrapper for *SUNLinSol_SuperLUMT()*.

and

int **SUNSuperLUMTSetOrdering**(SUNLinearSolver *S*, int *ordering_choice*)

Wrapper for *SUNLinSol_SuperLUMTSetOrdering()*.

For solvers that include a Fortran interface module, the SUNLinSol_SuperLUMT module also includes the Fortran-callable function *FSUNSuperLUMTInit()* to initialize this SUNLinSol_SuperLUMT module for a given SUNDIALS solver.

subroutine FSUNSuperLUMTInit(*CODE*, *NUM_THREADS*, *IER*)

Initializes a SuperLU_MT sparse `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* both the `N_Vector` and `SUNMatrix` objects have been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *NUM_THREADS* (`int`, input) – desired number of OpenMP/Pthreads threads to use in the factorization.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassSuperLUMTInit()* initializes this SUNLinSol_SuperLUMT module for solving mass matrix linear systems.

subroutine FSUNMassSuperLUMTInit(*NUM_THREADS*, *IER*)
Initializes a SuperLU_MT sparse `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* both the `N_Vector` and the mass `SUNMatrix` objects have been initialized.

Arguments:

- *NUM_THREADS* (`int`, input) – desired number of OpenMP/Pthreads threads to use in the factorization.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_SuperLUMTSetOrdering()* routine also supports Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSuperLUMTSetOrdering(*CODE*, *ORDERING*, *IER*)
Fortran interface to *SUNLinSol_SuperLUMTSetOrdering()* for system linear solvers.

This routine must be called *after* *FSUNSuperLUMTInit()* has been called

Arguments: all should have type `int` and have meanings identical to those listed above

subroutine FSUNMassSuperLUMTSetOrdering(*ORDERING*, *IER*)
Fortran interface to *SUNLinSol_SuperLUMTSetOrdering()* for mass matrix linear solves in ARKode.

This routine must be called *after* *FSUNMassSuperLUMTInit()* has been called

Arguments: all should have type `int` and have meanings identical to those listed above

12.9.2 SUNLinSol_SuperLUMT Description

The SUNLinSol_SuperLUMT module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
  int          last_flag;
  int          first_factorize;
  SuperMatrix  *A, *AC, *L, *U, *B;
  Gstat_t      *Gstat;
  sunindextype *perm_r, *perm_c;
  sunindextype N;
  int          num_threads;
  realtype     diag_pivot_thresh;
  int          ordering;
  superlumt_options_t *options;
};
```

These entries of the *content* field contain the following information:

- `last_flag` - last error return flag from internal function evaluations,
- `first_factorize` - flag indicating whether the factorization has ever been performed,
- `A, AC, L, U, B` - `SuperMatrix` pointers used in solve,
- `Gstat` - `GStat_t` object used in solve,
- `perm_r, perm_c` - permutation arrays used in solve,
- `N` - size of the linear system,

- `num_threads` - number of OpenMP/Pthreads threads to use,
- `diag_pivot_thresh` - threshold on diagonal pivoting,
- `ordering` - flag for which reordering algorithm to use,
- `options` - pointer to SuperLU_MT options structure.

The SUNLinSol_SuperLUMT module is a `SUNLinearSolver` wrapper for the SuperLU_MT sparse matrix factorization and solver library written by X. Sherry Li (*[SuperLUMT]*, *[L2005]*, *[DGL1999]*). The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step. In order to use the SUNLinSol_SuperLUMT interface to SuperLU_MT, it is assumed that SuperLU_MT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SuperLU_MT (see section *Working with external Libraries* for details). Additionally, this wrapper only supports single- and double-precision calculations, and therefore cannot be compiled if SUNDIALS is configured to have `realtype` set to `extended` (see section *Data Types* for details). Moreover, since the SuperLU_MT library may be installed to support either 32-bit or 64-bit integers, it is assumed that the SuperLU_MT library is installed using the same integer precision as the SUNDIALS `sunindextype` option.

The SuperLU_MT library has a symbolic factorization routine that computes the permutation of the linear system matrix to reduce fill-in on subsequent LU factorizations (using COLAMD, minimal degree ordering on $A^T * A$, minimal degree ordering on $A^T + A$, or natural ordering). Of these ordering choices, the default value in the SUNLinSol_SuperLUMT module is the COLAMD ordering.

Since the linear systems that arise within the context of SUNDIALS calculations will typically have identical sparsity patterns, the SUNLinSol_SuperLUMT module is constructed to perform the following operations:

- The first time that the "setup" routine is called, it performs the symbolic factorization, followed by an initial numerical factorization.
- On subsequent calls to the "setup" routine, it skips the symbolic factorization, and only refactors the input matrix.
- The "solve" call performs pivoting and forward and backward substitution using the stored SuperLU_MT data structures. We note that in this solve SuperLU_MT operates on the native data arrays for the right-hand side and solution vectors, without requiring costly data copies.

The SUNLinSol_SuperLUMT module defines implementations of all "direct" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_SuperLUMT`
- `SUNLinSolInitialize_SuperLUMT` – this sets the `first_factorize` flag to 1 and resets the internal SuperLU_MT statistics variables.
- `SUNLinSolSetup_SuperLUMT` – this performs either a LU factorization or refactorization of the input matrix.
- `SUNLinSolSolve_SuperLUMT` – this calls the appropriate SuperLU_MT solve routine to utilize the LU factors to solve the linear system.
- `SUNLinSolLastFlag_SuperLUMT`
- `SUNLinSolSpace_SuperLUMT` – this only returns information for the storage within the solver *interface*, i.e. storage for the integers `last_flag` and `first_factorize`. For additional space requirements, see the SuperLU_MT documentation.
- `SUNLinSolFree_SuperLUMT`

12.10 The SUNLinSol_cuSolverSp_batchQR Module

The `SUNLinearSolver_cuSolverSp_batchQR` implementation of the SUNLinearSolver API is designed to be used with the SUNMATRIX_CUSPARSE matrix, and the NVECTOR_CUDA vector. The header file to include when using this module is `sunlinsol/sunlinsol_cusolversp_batchqr.h`. The installed library to link to is `libsundials_sunlinsolcusolversp.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

The SUNLinearSolver_cuSolverSp_batchQR module is experimental and subject to change.

12.10.1 SUNLinSol_cuSolverSp_batchQR description

The SUNLinearSolver_cuSolverSp_batchQR implementation provides an interface to the batched sparse QR factorization method provided by the NVIDIA cuSOLVER library (*[cuSOLVER]*). The module is designed for solving block diagonal linear systems of the form

$$\begin{bmatrix} \mathbf{A_1} & 0 & \cdots & 0 \\ 0 & \mathbf{A_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{A_n} \end{bmatrix} x_j = b_j$$

where all block matrices $\mathbf{A_j}$ share the same sparsisty pattern. The matrix must be the *The SUNMATRIX_CUSPARSE Module*.

12.10.2 SUNLinSol_cuSolverSp_batchQR functions

The `SUNLinearSolver_cuSolverSp_batchQR` module defines implementations of all "direct" linear solver operations listed in *The SUNLinearSolver API*:

- `SUNLinSolGetType_cuSolverSp_batchQR`
- `SUNLinSolInitialize_cuSolverSp_batchQR` – this sets the `first_factorize` flag to 1
- `SUNLinSolSetup_cuSolverSp_batchQR` – this always copies the relevant SUNMATRIX_SPARSE data to the GPU; if this is the first setup it will perform symbolic analysis on the system
- `SUNLinSolSolve_cuSolverSp_batchQR` – this calls the `cusolverSpXcsrqrsvBatched` routine to perform factorization
- `SUNLinSolLastFlag_cuSolverSp_batchQR`
- `SUNLinSolFree_cuSolverSp_batchQR`

In addition, the module provides the following user-callable routines:

SUNLinearSolver **`SUNLinSol_cuSolverSp_batchQR`**(N_Vector *y*, SUNMatrix *A*, cusolverHandle_t *cusol*)

The function `SUNLinSol_cuSolverSp_batchQR` creates and allocates memory for a SUNLinearSolver object.

This returns a SUNLinearSolver object. If either `A` or `y` are incompatible then this routine will return `NULL`.

This routine analyzes the input matrix and vector to determine the linear system size and to assess compatibility with the solver.

This routine will perform consistency checks to ensure that it is called with consistent `N_Vector` and `SUNMatrix` implementations. These are currently limited to the SUNMATRIX_CUSPARSE matrix type

and the NVECTOR_CUDA vector type. Since the SUNMATRIX_CUSPARSE matrix type is only compatible with the NVECTOR_CUDA the restriction is also in place for the linear solver. As additional compatible matrix and vector implementations are added to SUNDIALS, these will be included within this compatibility check.

void **SUNLinSol_cuSolverSp_batchQR_GetDescription**(SUNLinearSolver *LS*, char **desc*)
The function `SUNLinSol_cuSolverSp_batchQR_GetDescription` accesses the string description of the object (empty by default).

void **SUNLinSol_cuSolverSp_batchQR_SetDescription**(SUNLinearSolver *LS*, const char *desc*)
The function `SUNLinSol_cuSolverSp_batchQR_SetDescription` sets the string description of the object (empty by default).

void **SUNLinSol_cuSolverSp_batchQR_GetDeviceSpace**(SUNLinearSolver *S*, size_t* *cuSolverInternal*, size_t* *cuSolverWorkspace*)
The function `SUNLinSol_cuSolverSp_batchQR_GetDeviceSpace` returns the cuSOLVER batch QR method internal buffer size, in bytes, in the argument `cuSolverInternal` and the cuSOLVER batch QR workspace buffer size, in bytes, in the agrument `cuSolverWorkspace`. The size of the internal buffer is proportional to the number of matrix blocks while the size of the workspace is almost independent of the number of blocks.

12.10.3 SUNLinSol_cuSolverSp_batchQR content

The SUNLinSol_cuSolverSp_batchQR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_cuSolverSp_batchQR {
   int                 last_flag;       /* last return flag                          */
   booleantype         first_factorize; /* is this the first factorization?          */
   size_t              internal_size;   /* size of cusolver buffer for Q and R       */
   size_t              workspace_size;  /* size of cusolver memory for factorization */
   cusolverSpHandle_t  cusolver_handle; /* cuSolverSp context                        */
   csrqrInfo_t         info;            /* opaque cusolver data structure            */
   void*               workspace;       /* memory block used by cusolver             */
   const char*         desc;            /* description of this linear solver         */
};
```

12.11 The SUNLinSol_SPGMR Module

The SPGMR (Scaled, Preconditioned, Generalized Minimum Residual *[SS1986]*) implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_SPGMR, is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (*N_VClone()*, *N_VDotProd()*, *N_VScale()*, *N_VLinearSum()*, *N_VProd()*, *N_VConst()*, *N_VDiv()*, and *N_VDestroy()*).

12.11.1 SUNLinSol_SPGMR Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_spgmr.h`.
The SUNinSol_SPGMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspgmr` module library.

The module SUNLinSol_SPGMR provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SPGMR**(N_Vector *y*, int *pretype*, int *maxl*)
This constructor function creates and allocates memory for a SPGMR SUNLinearSolver. Its arguments are an N_Vector, the desired type of preconditioning, and the number of Krylov basis vectors to use.

This routine will perform consistency checks to ensure that it is called with a consistent N_Vector implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

A maxl argument that is ≤ 0 will result in the default value (5).

Allowable inputs for pretype are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLinSol_SPGMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

int **SUNLinSol_SPGMRSetPrecType**(SUNLinearSolver *S*, int *pretype*)
This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2) and PREC_BOTH (3).

This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSol_SPGMRSetGSType**(SUNLinearSolver *S*, int *gstype*)
This function sets the type of Gram-Schmidt orthogonalization to use. Supported values are MODIFIED_GS (1) and CLASSICAL_GS (2). Any other integer input will result in a failure, returning error code SUNLS_ILL_INPUT.

This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal gstype), SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSol_SPGMRSetMaxRestarts**(SUNLinearSolver *S*, int *maxrs*)
This function sets the number of GMRES restarts to allow. A negative input will result in the default of 0.

This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSolSetInfoFile_SPGMR**(SUNLinearSolver *LS*, FILE* *info_file*)
The function *SUNLinSolSetInfoFile_SPGMR()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- *info_file* – pointer to output file (stdout by default); a NULL input will disable output

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **SUNLinSolSetPrintLevel_SPGMR**(SUNLinearSolver *LS*, int *print_level*)
The function *SUNLinSolSetPrintLevel_SPGMR()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each linear iteration the residual norm is printed

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was `NULL`
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or if the print level value was invalid

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNSPGMR**(N_Vector *y*, int *pretype*, int *maxl*)
Wrapper function for *SUNLinSol_SPGMR()*

int **SUNSPGMRSetPrecType**(SUNLinearSolver *S*, int *pretype*)
Wrapper function for *SUNLinSol_SPGMRSetPrecType()*

int **SUNSPGMRSetGSType**(SUNLinearSolver *S*, int *gstype*)
Wrapper function for *SUNLinSol_SPGMRSetGSType()*

int **SUNSPGMRSetMaxRestarts**(SUNLinearSolver *S*, int *maxrs*)
Wrapper function for *SUNLinSol_SPGMRSetMaxRestarts()*

For solvers that include a Fortran interface module, the SUNLinSol_SPGMR module also includes the Fortran-callable function *FSUNSPGMRInit()* to initialize this SUNLinSol_SPGMR module for a given SUNDIALS solver.

subroutine FSUNSPGMRInit(*CODE*, *PRETYPE*, *MAXL*, *IER*)
Initializes a SPGMR `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *PRETYPE* (`int`, input) – flag denoting type of preconditioning to use: none=0, left=1, right=2, both=3.
- *MAXL* (`int`, input) – number of GMRES basis vectors to use.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassSPGMRInit()* initializes this SUNLinSol_SPGMR module for solving mass matrix linear systems.

subroutine FSUNMassSPGMRInit(*PRETYPE*, *MAXL*, *IER*)
Initializes a SPGMR `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *PRETYPE* (`int`, input) – flag denoting type of preconditioning to use: none=0, left=1, right=2, both=3.
- *MAXL* (`int`, input) – number of GMRES basis vectors to use.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_SPGMRSetGSType()*, *SUNLinSol_SPGMRSetPrecType()* and *SUNLinSol_SPGMRSetMaxRestarts()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPGMRSetGSType(*CODE*, *GSTYPE*, *IER*)
Fortran interface to *SUNLinSol_SPGMRSetGSType()* for system linear solvers.

This routine must be called *after* *FSUNSPGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPGMRSetGSType(*GSTYPE*, *IER*)
Fortran interface to *SUNLinSol_SPGMRSetGSType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNSPGMRSetPrecType(*CODE*, *PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPGMRSetPrecType()* for system linear solvers.

This routine must be called *after* *FSUNSPGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPGMRSetPrecType(*PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPGMRSetPrecType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNSPGMRSetMaxRS(*CODE*, *MAXRS*, *IER*)
Fortran interface to `SUNLinSol_SPGMRSetMaxRS()` for system linear solvers.

This routine must be called *after* *FSUNSPGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPGMRSetMaxRS(*MAXRS*, *IER*)
Fortran interface to `SUNLinSol_SPGMRSetMaxRS()` for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

12.11.2 SUNLinSol_SPGMR Description

The SUNLinSol_SPGMR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPGMR {
  int maxl;
  int pretype;
  int gstype;
```

```
  int max_restarts;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector *V;
  realtype **Hes;
  realtype *givens;
  N_Vector xcor;
  realtype *yg;
  N_Vector vtemp;
  int      print_level;
  FILE*    info_file;
};
```

These entries of the *content* field contain the following information:

- `maxl` - number of GMRES basis vectors to use (default is 5),
- `pretype` - flag for type of preconditioning to employ (default is none),
- `gstype` - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
- `max_restarts` - number of GMRES restarts to allow (default is 0),
- `numiters` - number of iterations from the most-recent solve,
- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is `NULL`),
- `V` - the array of Krylov basis vectors $v_1, \ldots, v_{\text{maxl}+1}$, stored in `V[0], ... V[maxl]`. Each v_i is a vector of type `N_Vector`,
- `Hes` - the $(\text{maxl} + 1) \times$ maxl Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`,
- `givens` - a length 2 maxl array which represents the Givens rotation matrices that arise in the GMRES algo-

rithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ..., `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$,

- `xcor` - a vector which holds the scaled, preconditioned correction to the initial guess,
- `yg` - a length $(\text{maxl}+1)$ array of `realtype` values used to hold "short" vectors (e.g. y and g),
- `vtemp` - temporary vector storage.
- `print_level` - controls the amount of information to be printed to the info file
- `info_file` - the file where all informative (non-error) messages will be directed

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPGMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg`)
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call, the GMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The SUNLinSol_SPGMR module defines implementations of all "iterative" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_SPGMR`
- `SUNLinSolInitialize_SPGMR`
- `SUNLinSolSetATimes_SPGMR`
- `SUNLinSolSetPreconditioner_SPGMR`
- `SUNLinSolSetScalingVectors_SPGMR`
- `SUNLinSolSetup_SPGMR`
- `SUNLinSolSolve_SPGMR`
- `SUNLinSolNumIters_SPGMR`
- `SUNLinSolResNorm_SPGMR`
- `SUNLinSolResid_SPGMR`

- SUNLinSolLastFlag_SPGMR
- SUNLinSolSpace_SPGMR
- SUNLinSolFree_SPGMR

12.12 The SUNLinSol_SPFGMR Module

The SPFGMR (Scaled, Preconditioned, Flexible, Generalized Minimum Residual *[S1993]*) implementation of the SUNLinearSolver module provided with SUNDIALS, SUNLinSol_SPFGMR, is an iterative linear solver that is designed to be compatible with any N_Vector implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (*N_VClone()*, *N_VDotProd()*, *N_VScale()*, *N_VLinearSum()*, *N_VProd()*, *N_VConst()*, *N_VDiv()*, and *N_VDestroy()*). Unlike the other Krylov iterative linear solvers supplied with SUNDIALS, FGMRES is specifically designed to work with a changing preconditioner (e.g. from an iterative method).

12.12.1 SUNLinSol_SPFGMR Usage

The header file to be included when using this module is sunlinsol/sunlinsol_spfgmr.h.
The SUNLinSol_SPFGMR module is accessible from all SUNDIALS solvers *without* linking to the libsundials_sunlinsolspfgmr module library.

The module SUNLinSol_SPFGMR provides the following user-callable routines:

SUNLinearSolver **SUNLinSol_SPFGMR**(N_Vector *y*, int *pretype*, int *maxl*)
: This constructor function creates and allocates memory for a SPFGMR SUNLinearSolver. Its arguments are an N_Vector, a flag indicating to use preconditioning, and the number of Krylov basis vectors to use.

  This routine will perform consistency checks to ensure that it is called with a consistent N_Vector implementation (i.e. that it supplies the requisite vector operations). If y is incompatible, then this routine will return NULL.

  A maxl argument that is ≤ 0 will result in the default value (5).

  Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of PREC_RIGHT; any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS). While it is possible to use a right-preconditioned SUNLinSol_SPFGMR object for these packages, this use mode is not supported and may result in inferior performance.

int **SUNLinSol_SPFGMRSetPrecType**(SUNLinearSolver *S*, int *pretype*)
: This function updates the flag indicating use of preconditioning. Since the FGMRES algorithm is designed to only support right preconditioning, then any of the pretype inputs PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will result in use of PREC_RIGHT; any other integer input will result in the default (no preconditioning).

  This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSol_SPFGMRSetGSType**(SUNLinearSolver *S*, int *gstype*)
: This function sets the type of Gram-Schmidt orthogonalization to use. Supported values are MODIFIED_GS (1) and CLASSICAL_GS (2). Any other integer input will result in a failure, returning error code SUNLS_ILL_INPUT.

  This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal gstype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

int **SUNLinSol_SPFGMRSetMaxRestarts**(SUNLinearSolver *S*, int *maxrs*)
This function sets the number of FGMRES restarts to allow. A negative input will result in the default of 0.

This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSolSetInfoFile_SPFGMR**(SUNLinearSolver *LS*, FILE* *info_file*)
The function *SUNLinSolSetInfoFile_SPFGMR()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (stdout by default);** a NULL input will disable output

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option ‘‘SUNDIALS_BUILD_WITH_MONITORING‘‘, to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **SUNLinSolSetPrintLevel_SPFGMR**(SUNLinearSolver *LS*, int *print_level*)
The function *SUNLinSolSetPrintLevel_SPFGMR()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each linear iteration the residual norm is printed

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or if the print level value was invalid

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option ‘‘SUNDIALS_BUILD_WITH_MONITORING‘‘, to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNSPFGMR**(N_Vector *y*, int *pretype*, int *maxl*)
Wrapper function for *SUNLinSol_SPFGMR()*

int **SUNSPFGMRSetPrecType**(SUNLinearSolver *S*, int *pretype*)
Wrapper function for *SUNLinSol_SPFGMRSetPrecType()*

int **SUNSPFGMRSetGSType**(SUNLinearSolver *S*, int *gstype*)
Wrapper function for *SUNLinSol_SPFGMRSetGSType()*

int **SUNSPFGMRSetMaxRestarts**(SUNLinearSolver *S*, int *maxrs*)
Wrapper function for *SUNLinSol_SPFGMRSetMaxRestarts()*

For solvers that include a Fortran interface module, the SUNLinSol_SPFGMR module also includes the Fortran-callable function *FSUNSPFGMRInit()* to initialize this SUNLinSol_SPFGMR module for a given SUNDIALS solver.

subroutine FSUNSPFGMRInit(*CODE*, *PRETYPE*, *MAXL*, *IER*)
Initializes a SPFGMR `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *PRETYPE* (`int`, input) – flag denoting whether to use preconditioning: no=0, yes=1.
- *MAXL* (`int`, input) – number of FGMRES basis vectors to use.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassSPFGMRInit()* initializes this SUNLinSol_SPFGMR module for solving mass matrix linear systems.

subroutine FSUNMassSPFGMRInit(*PRETYPE*, *MAXL*, *IER*)
Initializes a SPFGMR `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *PRETYPE* (`int`, input) – flag denoting whether to use preconditioning: no=0, yes=1.
- *MAXL* (`int`, input) – number of FGMRES basis vectors to use.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_SPFGMRSetGSType()*, *SUNLinSol_SPFGMRSetPrecType()* and *SUNLinSol_SPFGMRSetMaxRestarts()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPFGMRSetGSType(*CODE*, *GSTYPE*, *IER*)
Fortran interface to *SUNLinSol_SPFGMRSetGSType()* for system linear solvers.

This routine must be called *after* *FSUNSPFGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPFGMRSetGSType(*GSTYPE*, *IER*)
Fortran interface to *SUNLinSol_SPFGMRSetGSType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPFGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNSPFGMRSetPrecType(*CODE*, *PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPFGMRSetPrecType()* for system linear solvers.

This routine must be called *after* *FSUNSPFGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPFGMRSetPrecType(*PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPFGMRSetPrecType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPFGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNSPFGMRSetMaxRS(*CODE*, *MAXRS*, *IER*)
Fortran interface to `SUNLinSol_SPFGMRSetMaxRS()` for system linear solvers.

This routine must be called *after* *FSUNSPFGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPFGMRSetMaxRS(*MAXRS*, *IER*)
Fortran interface to `SUNLinSol_SPFGMRSetMaxRS()` for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPFGMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

12.12.2 SUNLinSol_SPFGMR Description

The SUNLinSol_SPFGMR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPFGMR {
  int maxl;
  int pretype;
  int gstype;
  int max_restarts;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector *V;
  N_Vector *Z;
  realtype **Hes;
  realtype *givens;
  N_Vector xcor;
  realtype *yg;
  N_Vector vtemp;
  int      print_level;
  FILE*    info_file;
};
```

These entries of the *content* field contain the following information:

- `maxl` - number of FGMRES basis vectors to use (default is 5),
- `pretype` - flag for use of preconditioning (default is none),
- `gstype` - flag for type of Gram-Schmidt orthogonalization (default is modified Gram-Schmidt),
- `max_restarts` - number of FGMRES restarts to allow (default is 0),
- `numiters` - number of iterations from the most-recent solve,

- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is `NULL`),
- `V` - the array of Krylov basis vectors $v_1, \ldots, v_{\text{maxl}+1}$, stored in `V[0], ..., V[maxl]`. Each v_i is a vector of type `N_Vector`,
- `Z` - the array of preconditioned Krylov basis vectors $z_1, \ldots, z_{\text{maxl}+1}$, stored in `Z[0], ..., Z[maxl]`. Each z_i is a vector of type `N_Vector`,
- `Hes` - the $(\text{maxl}+1) \times \text{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`,
- `givens` - a length $2\,\text{maxl}$ array which represents the Givens rotation matrices that arise in the FGMRES algorithm. These matrices are $F_0, F_1, \ldots, F_j$, where

$$F_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & -s_i & & & \\ & & & s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix},$$

  are represented in the `givens` vector as `givens[0]` $= c_0$, `givens[1]` $= s_0$, `givens[2]` $= c_1$, `givens[3]` $= s_1$, ..., `givens[2j]` $= c_j$, `givens[2j+1]` $= s_j$,
- `xcor` - a vector which holds the scaled, preconditioned correction to the initial guess,
- `yg` - a length $(\text{maxl}+1)$ array of `realtype` values used to hold "short" vectors (e.g. y and g),
- `vtemp` - temporary vector storage.
- `print_level` - controls the amount of information to be printed to the info file
- `info_file` - the file where all informative (non-error) messages will be directed

This solver is constructed to perform the following operations:

- During construction, the `xcor` and `vtemp` arrays are cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPFGMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the remaining solver data is allocated (`V`, `Hes`, `givens`, and `yg`)
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).

- In the "solve" call, the FGMRES iteration is performed. This will include scaling, preconditioning, and restarts if those options have been supplied.

The SUNLinSol_SPFGMR module defines implementations of all "iterative" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_SPFGMR`
- `SUNLinSolInitialize_SPFGMR`
- `SUNLinSolSetATimes_SPFGMR`
- `SUNLinSolSetPreconditioner_SPFGMR`
- `SUNLinSolSetScalingVectors_SPFGMR`
- `SUNLinSolSetup_SPFGMR`
- `SUNLinSolSolve_SPFGMR`
- `SUNLinSolNumIters_SPFGMR`
- `SUNLinSolResNorm_SPFGMR`
- `SUNLinSolResid_SPFGMR`
- `SUNLinSolLastFlag_SPFGMR`
- `SUNLinSolSpace_SPFGMR`
- `SUNLinSolFree_SPFGMR`

12.13 The SUNLinSol_SPBCGS Module

The SPBCGS (Scaled, Preconditioned, Bi-Conjugate Gradient, Stabilized *[V1992]*) implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_SPBCGS, is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (*`N_VClone()`*, *`N_VDotProd()`*, *`N_VScale()`*, *`N_VLinearSum()`*, *`N_VProd()`*, *`N_VDiv()`*, and *`N_VDestroy()`*). Unlike the SPGMR and SPFGMR algorithms, SPBCGS requires a fixed amount of memory that does not increase with the number of allowed iterations.

12.13.1 SUNLinSol_SPBCGS Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_spbcgs.h`.
The SUNLinSol_SPBCGS module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolspbcgs` module library.

The module SUNLinSol_SPBCGS provides the following user-callable routines:

SUNLinearSolver **`SUNLinSol_SPBCGS`**(N_Vector *y*, int *pretype*, int *maxl*)
: This constructor function creates and allocates memory for a SPBCGS `SUNLinearSolver`. Its arguments are an `N_Vector`, the desired type of preconditioning, and the number of linear iterations to allow.

  This routine will perform consistency checks to ensure that it is called with a consistent `N_Vector` implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible, then this routine will return `NULL`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for `pretype` are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2) and `PREC_BOTH` (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS

solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLinSol_SPBCGS object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

int **SUNLinSol_SPBCGSSetPrecType**(SUNLinearSolver *S*, int *pretype*)
This function updates the type of preconditioning to use. Supported values are PREC_NONE (0), PREC_LEFT (1), PREC_RIGHT (2), and PREC_BOTH (3).

This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

int **SUNLinSol_SPBCGSSetMaxl**(SUNLinearSolver *S*, int *maxl*)
This function updates the number of linear solver iterations to allow.

A maxl argument that is ≤ 0 will result in the default value (5).

This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSolSetInfoFile_SPBCGS**(SUNLinearSolver *LS*, FILE* *info_file*)
The function *SUNLinSolSetInfoFile_SPBCGS()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (stdout by default);** a NULL input will disable output

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **SUNLinSolSetPrintLevel_SPBCGS**(SUNLinearSolver *LS*, int *print_level*)
The function *SUNLinSolSetPrintLevel_SPBCGS()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each linear iteration the residual norm is printed

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or if the print level value was invalid

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNSPBCGS**(N_Vector *y*, int *pretype*, int *maxl*)
Wrapper function for *SUNLinSol_SPBCGS()*

int **SUNSPBCGSSetPrecType**(SUNLinearSolver *S*, int *pretype*)
Wrapper function for *SUNLinSol_SPBCGSSetPrecType()*

int **SUNSPBCGSSetMaxl**(SUNLinearSolver *S*, int *maxl*)
Wrapper function for *SUNLinSol_SPBCGSSetMaxl()*

For solvers that include a Fortran interface module, the SUNLinSol_SPBCGS module also includes the Fortran-callable function *FSUNSPBCGSInit()* to initialize this SUNLinSol_SPBCGS module for a given SUNDIALS solver.

subroutine FSUNSPBCGSInit(*CODE*, *PRETYPE*, *MAXL*, *IER*)
Initializes a SPBCGS `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *PRETYPE* (`int`, input) – flag denoting type of preconditioning to use: none=0, left=1, right=2, both=3.
- *MAXL* (`int`, input) – number of SPBCGS iterations to allow.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassSPBCGSInit()* initializes this SUNLinSol_SPBCGS module for solving mass matrix linear systems.

subroutine FSUNMassSPBCGSInit(*PRETYPE*, *MAXL*, *IER*)
Initializes a SPBCGS `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *PRETYPE* (`int`, input) – flag denoting type of preconditioning to use: none=0, left=1, right=2, both=3.
- *MAXL* (`int`, input) – number of SPBCGS iterations to allow.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_SPBCGSSetPrecType()* and *SUNLinSol_SPBCGSSetMaxl()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPBCGSSetPrecType(*CODE*, *PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPBCGSSetPrecType()* for system linear solvers.

This routine must be called *after* *FSUNSPBCGSInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPBCGSSetPrecType(*PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPBCGSSetPrecType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* `FSUNMassSPBCGSInit()` has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNSPBCGSSetMaxl(*CODE*, *MAXL*, *IER*)
Fortran interface to `SUNLinSol_SPBCGSSetMaxl()` for system linear solvers.

This routine must be called *after* `FSUNSPBCGSInit()` has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPBCGSSetMaxl(*MAXL*, *IER*)
Fortran interface to `SUNLinSol_SPBCGSSetMaxl()` for mass matrix linear solvers in ARKode.

This routine must be called *after* `FSUNMassSPBCGSInit()` has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

12.13.2 SUNLinSol_SPBCGS Description

The SUNLinSol_SPBCGS module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPBCGS {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector r;
  N_Vector r_star;
  N_Vector p;
  N_Vector q;
  N_Vector u;
  N_Vector Ap;
  N_Vector vtemp;
  int      print_level;
  FILE*    info_file;
};
```

These entries of the *content* field contain the following information:

- `maxl` - number of SPBCGS iterations to allow (default is 5),
- `pretype` - flag for type of preconditioning to employ (default is none),
- `numiters` - number of iterations from the most-recent solve,
- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,

- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is `NULL`),
- `r` - a `N_Vector` which holds the current scaled, preconditioned linear system residual,
- `r_star` - a `N_Vector` which holds the initial scaled, preconditioned linear system residual,
- `p, q, u, Ap, vtemp` - `N_Vector` used for workspace by the SPBCGS algorithm.
- `print_level` - controls the amount of information to be printed to the info file
- `info_file` - the file where all informative (non-error) messages will be directed

This solver is constructed to perform the following operations:

- During construction all `N_Vector` solver data is allocated, with vectors cloned from a template `N_Vector` that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPBCGS to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the SPBCGS iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLinSol_SPBCGS module defines implementations of all "iterative" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_SPBCGS`
- `SUNLinSolInitialize_SPBCGS`
- `SUNLinSolSetATimes_SPBCGS`
- `SUNLinSolSetPreconditioner_SPBCGS`
- `SUNLinSolSetScalingVectors_SPBCGS`
- `SUNLinSolSetup_SPBCGS`
- `SUNLinSolSolve_SPBCGS`
- `SUNLinSolNumIters_SPBCGS`
- `SUNLinSolResNorm_SPBCGS`
- `SUNLinSolResid_SPBCGS`
- `SUNLinSolLastFlag_SPBCGS`
- `SUNLinSolSpace_SPBCGS`
- `SUNLinSolFree_SPBCGS`

12.14 The SUNLinSol_SPTFQMR Module

The SPTFQMR (Scaled, Preconditioned, Transpose-Free Quasi-Minimum Residual *[F1993]*) implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_SPTFQMR, is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (*`N_VClone()`*, *`N_VDotProd()`*, *`N_VScale()`*, *`N_VLinearSum()`*, *`N_VProd()`*, *`N_VConst()`*, *`N_VDiv()`*, and *`N_VDestroy()`*). Unlike the SPGMR and SPFGMR algorithms, SPTFQMR requires a fixed amount of memory that does not increase with the number of allowed iterations.

12.14.1 SUNLinSol_SPTFQMR Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_sptfqmr.h`.
The SUNLinSol_SPTFQMR module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolsptfqmr` module library.

The module SUNLinSol_SPTFQMR provides the following user-callable routines:

SUNLinearSolver **`SUNLinSol_SPTFQMR`**(N_Vector *y*, int *pretype*, int *maxl*)
: This constructor function creates and allocates memory for a SPTFQMR `SUNLinearSolver`. Its arguments are an `N_Vector`, the desired type of preconditioning, and the number of linear iterations to allow.

  This routine will perform consistency checks to ensure that it is called with a consistent `N_Vector` implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible, then this routine will return `NULL`.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  Allowable inputs for `pretype` are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2) and `PREC_BOTH` (3); any other integer input will result in the default (no preconditioning). We note that some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL). While it is possible to configure a SUNLinSol_SPTFQMR object to use any of the preconditioning options with these solvers, this use mode is not supported and may result in inferior performance.

int **`SUNLinSol_SPTFQMRSetPrecType`**(SUNLinearSolver *S*, int *pretype*)
: This function updates the type of preconditioning to use. Supported values are `PREC_NONE` (0), `PREC_LEFT` (1), `PREC_RIGHT` (2), and `PREC_BOTH` (3).

  This routine will return with one of the error codes `SUNLS_ILL_INPUT` (illegal `pretype`), `SUNLS_MEM_NULL` (`S` is `NULL`), or `SUNLS_SUCCESS`.

int **`SUNLinSol_SPTFQMRSetMaxl`**(SUNLinearSolver *S*, int *maxl*)
: This function updates the number of linear solver iterations to allow.

  A `maxl` argument that is ≤ 0 will result in the default value (5).

  This routine will return with one of the error codes `SUNLS_MEM_NULL` (`S` is `NULL`) or `SUNLS_SUCCESS`.

int **`SUNLinSolSetInfoFile_SPTFQMR`**(SUNLinearSolver *LS*, FILE* *info_file*)
: The function *`SUNLinSolSetInfoFile_SPTFQMR()`* sets the output file where all informative (non-error) messages should be directed.

  Arguments:

  - *LS* – a SUNLinSol object
  - ***info_file* – pointer to output file (`stdout` by default);** a `NULL` input will disable output

  Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was `NULL`
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to `stdout`.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **SUNLinSolSetPrintLevel_SPTFQMR**(SUNLinearSolver *LS*, int *print_level*)
The function *SUNLinSolSetPrintLevel_SPTFQMR()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each linear iteration the residual norm is printed

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was `NULL`
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or if the print level value was invalid

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNSPTFQMR**(N_Vector *y*, int *pretype*, int *maxl*)
Wrapper function for *SUNLinSol_SPTFQMR()*

int **SUNSPTFQMRSetPrecType**(SUNLinearSolver *S*, int *pretype*)
Wrapper function for *SUNLinSol_SPTFQMRSetPrecType()*

int **SUNSPTFQMRSetMaxl**(SUNLinearSolver *S*, int *maxl*)
Wrapper function for *SUNLinSol_SPTFQMRSetMaxl()*

For solvers that include a Fortran interface module, the SUNLinSol_SPTFQMR module also includes the Fortran-callable function *FSUNSPTFQMRInit()* to initialize this SUNLinSol_SPTFQMR module for a given SUNDIALS solver.

subroutine FSUNSPTFQMRInit(*CODE*, *PRETYPE*, *MAXL*, *IER*)
Initializes a SPTFQMR `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.

- *PRETYPE* (`int`, input) – flag denoting type of preconditioning to use: none=0, left=1, right=2, both=3.
- *MAXL* (`int`, input) – number of SPTFQMR iterations to allow.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassSPTFQMRInit()* initializes this SUNLinSol_SPTFQMR module for solving mass matrix linear systems.

subroutine FSUNMassSPTFQMRInit(*PRETYPE*, *MAXL*, *IER*)
Initializes a SPTFQMR `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *PRETYPE* (`int`, input) – flag denoting type of preconditioning to use: none=0, left=1, right=2, both=3.
- *MAXL* (`int`, input) – number of SPTFQMR iterations to allow.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_SPTFQMRSetPrecType()* and *SUNLinSol_SPTFQMRSetMaxl()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPTFQMRSetPrecType(*CODE*, *PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPTFQMRSetPrecType()* for system linear solvers.

This routine must be called *after* *FSUNSPTFQMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPTFQMRSetPrecType(*PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_SPTFQMRSetPrecType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPTFQMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNSPTFQMRSetMaxl(*CODE*, *MAXL*, *IER*)
Fortran interface to *SUNLinSol_SPTFQMRSetMaxl()* for system linear solvers.

This routine must be called *after* *FSUNSPTFQMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassSPTFQMRSetMaxl(*MAXL*, *IER*)
Fortran interface to *SUNLinSol_SPTFQMRSetMaxl()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassSPTFQMRInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

12.14.2 SUNLinSol_SPTFQMR Description

The SUNLinSol_SPTFQMR module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_SPTFQMR {
  int maxl;
  int pretype;
```

```
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s1;
  N_Vector s2;
  N_Vector r_star;
  N_Vector q;
  N_Vector d;
  N_Vector v;
  N_Vector p;
  N_Vector *r;
  N_Vector u;
  N_Vector vtemp1;
  N_Vector vtemp2;
  N_Vector vtemp3;
  int      print_level;
  FILE*    info_file;
};
```

These entries of the *content* field contain the following information:

- `maxl` - number of TFQMR iterations to allow (default is 5),
- `pretype` - flag for type of preconditioning to employ (default is none),
- `numiters` - number of iterations from the most-recent solve,
- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s1, s2` - vector pointers for supplied scaling matrices (default is `NULL`),
- `r_star` - a `N_Vector` which holds the initial scaled, preconditioned linear system residual,
- `q, d, v, p, u` - `N_Vector` used for workspace by the SPTFQMR algorithm,
- `r` - array of two `N_Vector` used for workspace within the SPTFQMR algorithm,
- `vtemp1, vtemp2, vtemp3` - temporary vector storage.
- `print_level` - controls the amount of information to be printed to the info file
- `info_file` - the file where all informative (non-error) messages will be directed

This solver is constructed to perform the following operations:

- During construction all `N_Vector` solver data is allocated, with vectors cloned from a template `N_Vector` that is input, and default solver parameters are set.

- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_SPTFQMR to supply the `ATimes`, `PSetup`, and `Psolve` function pointers and `s1` and `s2` scaling vectors.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-`NULL` `PSetup` function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic `PSetup` function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the TFQMR iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLinSol_SPTFQMR module defines implementations of all "iterative" linear solver operations listed in the section *The SUNLinearSolver API*:

- `SUNLinSolGetType_SPTFQMR`
- `SUNLinSolInitialize_SPTFQMR`
- `SUNLinSolSetATimes_SPTFQMR`
- `SUNLinSolSetPreconditioner_SPTFQMR`
- `SUNLinSolSetScalingVectors_SPTFQMR`
- `SUNLinSolSetup_SPTFQMR`
- `SUNLinSolSolve_SPTFQMR`
- `SUNLinSolNumIters_SPTFQMR`
- `SUNLinSolResNorm_SPTFQMR`
- `SUNLinSolResid_SPTFQMR`
- `SUNLinSolLastFlag_SPTFQMR`
- `SUNLinSolSpace_SPTFQMR`
- `SUNLinSolFree_SPTFQMR`

12.15 The SUNLinSol_PCG Module

The PCG (Preconditioned Conjugate Gradient *[HS1952]* implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLinSol_PCG, is an iterative linear solver that is designed to be compatible with any `N_Vector` implementation (serial, threaded, parallel, and user-supplied) that supports a minimal subset of operations (*`N_VClone()`*, *`N_VDotProd()`*, *`N_VScale()`*, *`N_VLinearSum()`*, *`N_VProd()`*, and *`N_VDestroy()`*). Unlike the SPGMR and SPFGMR algorithms, PCG requires a fixed amount of memory that does not increase with the number of allowed iterations.

Unlike all of the other iterative linear solvers supplied with SUNDIALS, PCG should only be used on *symmetric* linear systems (e.g. mass matrix linear systems encountered in ARKode). As a result, the explanation of the role of scaling and preconditioning matrices given in general must be modified in this scenario. The PCG algorithm solves a linear system $Ax = b$ where A is a symmetric ($A^T = A$), real-valued matrix. Preconditioning is allowed, and is applied in a symmetric fashion on both the right and left. Scaling is also allowed and is applied symmetrically. We denote the preconditioner and scaling matrices as follows:

- P is the preconditioner (assumed symmetric),
- S is a diagonal matrix of scale factors.

The matrices A and P are not required explicitly; only routines that provide A and P^{-1} as operators are required. The diagonal of the matrix S is held in a single `N_Vector`, supplied by the user.

In this notation, PCG applies the underlying CG algorithm to the equivalent transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{12.4}$$

where

$$\begin{aligned} \tilde{A} &= SP^{-1}AP^{-1}S, \\ \tilde{b} &= SP^{-1}b, \\ \tilde{x} &= S^{-1}Px. \end{aligned} \tag{12.5}$$

The scaling matrix must be chosen so that the vectors $SP^{-1}b$ and $S^{-1}Px$ have dimensionless components.

The stopping test for the PCG iterations is on the L2 norm of the scaled preconditioned residual:

$$\begin{aligned} & \|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta \\ \Leftrightarrow \\ & \|SP^{-1}b - SP^{-1}Ax\|_2 < \delta \\ \Leftrightarrow \\ & \|P^{-1}b - P^{-1}Ax\|_S < \delta \end{aligned}$$

where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance δ.

12.15.1 SUNLinSol_PCG Usage

The header file to be included when using this module is `sunlinsol/sunlinsol_pcg.h`. The SUNLinSol_PCG module is accessible from all SUNDIALS solvers *without* linking to the `libsundials_sunlinsolpcg` module library.

The module SUNLinSol_PCG provides the following user-callable routines:

SUNLinearSolver **`SUNLinSol_PCG`**(N_Vector *y*, int *pretype*, int *maxl*)

This constructor function creates and allocates memory for a PCG `SUNLinearSolver`. Its arguments are an `N_Vector`, a flag indicating to use preconditioning, and the number of linear iterations to allow.

This routine will perform consistency checks to ensure that it is called with a consistent `N_Vector` implementation (i.e. that it supplies the requisite vector operations). If `y` is incompatible then this routine will return `NULL`.

A `maxl` argument that is ≤ 0 will result in the default value (5).

Since the PCG algorithm is designed to only support symmetric preconditioning, then any of the `pretype` inputs `PREC_LEFT` (1), `PREC_RIGHT` (2), or `PREC_BOTH` (3) will result in use of the symmetric preconditioner; any other integer input will result in the default (no preconditioning). Although some SUNDIALS solvers are designed to only work with left preconditioning (IDA and IDAS) and others with only right preconditioning (KINSOL), PCG should *only* be used with these packages when the linear systems are known to be *symmetric*. Since the scaling of matrix rows and columns must be identical in a symmetric matrix, symmetric preconditioning should work appropriately even for packages designed with one-sided preconditioning in mind.

int **SUNLinSol_PCGSetPrecType**(SUNLinearSolver *S*, int *pretype*)
This function updates the flag indicating use of preconditioning. As above, any one of the input values, PREC_LEFT (1), PREC_RIGHT (2), or PREC_BOTH (3) will enable preconditioning; PREC_NONE (0) disables preconditioning.

This routine will return with one of the error codes SUNLS_ILL_INPUT (illegal pretype), SUNLS_MEM_NULL (S is NULL), or SUNLS_SUCCESS.

int **SUNLinSol_PCGSetMaxl**(SUNLinearSolver *S*, int *maxl*)
This function updates the number of linear solver iterations to allow.

A maxl argument that is ≤ 0 will result in the default value (5).

This routine will return with one of the error codes SUNLS_MEM_NULL (S is NULL) or SUNLS_SUCCESS.

int **SUNLinSolSetInfoFile_PCG**(SUNLinearSolver *LS*, FILE* *info_file*)
The function *SUNLinSolSetInfoFile_PCG()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *LS* – a SUNLinSol object
- ***info_file* – pointer to output file (stdout by default);** a NULL input will disable output

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the file pointer is set to stdout.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **SUNLinSolSetPrintLevel_PCG**(SUNLinearSolver *LS*, int *print_level*)
The function *SUNLinSolSetPrintLevel_PCG()* specifies the level of verbosity of the output.

Arguments:

- *LS* – a SUNLinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each linear iteration the residual norm is printed

Return value:

- *SUNLS_SUCCESS* if successful
- *SUNLS_MEM_NULL* if the SUNLinearSolver memory was NULL
- *SUNLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or if the print level value was invalid

Notes: This function is intended for users that wish to monitor the linear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

For backwards compatibility, we also provide the wrapper functions, each with identical input and output arguments to the routines that they wrap:

SUNLinearSolver **SUNPCG**(N_Vector *y*, int *pretype*, int *maxl*)
Wrapper function for *SUNLinSol_PCG()*

int **SUNPCGSetPrecType**(SUNLinearSolver *S*, int *pretype*)
Wrapper function for *SUNLinSol_PCGSetPrecType()*

int **SUNPCGSetMaxl**(SUNLinearSolver *S*, int *maxl*)
Wrapper function for *SUNLinSol_PCGSetMaxl()*

For solvers that include a Fortran interface module, the SUNLinSol_PCG module also includes the Fortran-callable function *FSUNPCGInit()* to initialize this SUNLinSol_PCG module for a given SUNDIALS solver.

subroutine FSUNPCGInit(*CODE*, *PRETYPE*, *MAXL*, *IER*)
Initializes a PCG `SUNLinearSolver` structure for use in a SUNDIALS package.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, KINSOL=3, ARKode=4.
- *PRETYPE* (`int`, input) – flag denoting whether to use symmetric preconditioning: no=0, yes=1.
- *MAXL* (`int`, input) – number of PCG iterations to allow.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

Additionally, when using ARKode with a non-identity mass matrix, the Fortran-callable function *FSUNMassPCGInit()* initializes this SUNLinSol_PCG module for solving mass matrix linear systems.

subroutine FSUNMassPCGInit(*PRETYPE*, *MAXL*, *IER*)
Initializes a PCG `SUNLinearSolver` structure for use in solving mass matrix systems in ARKode.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *PRETYPE* (`int`, input) – flag denoting whether to use symmetric preconditioning: no=0, yes=1.
- *MAXL* (`int`, input) – number of PCG iterations to allow.
- *IER* (`int`, output) – return flag (0 success, -1 for failure).

The *SUNLinSol_PCGSetPrecType()* and *SUNLinSol_PCGSetMaxl()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNPCGSetPrecType(*CODE*, *PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_PCGSetPrecType()* for system linear solvers.

This routine must be called *after* *FSUNPCGInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassPCGSetPrecType(*PRETYPE*, *IER*)
Fortran interface to *SUNLinSol_PCGSetPrecType()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassPCGInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNPCGSetMaxl(*CODE*, *MAXL*, *IER*)
Fortran interface to *SUNLinSol_PCGSetMaxl()* for system linear solvers.

This routine must be called *after* *FSUNPCGInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

subroutine FSUNMassPCGSetMaxl(*MAXL*, *IER*)
Fortran interface to *SUNLinSol_PCGSetMaxl()* for mass matrix linear solvers in ARKode.

This routine must be called *after* *FSUNMassPCGInit()* has been called.

Arguments: all should have type `int`, and have meanings identical to those listed above.

12.15.2 SUNLinSol_PCG Description

The SUNLinSol_PCG module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_PCG {
  int maxl;
  int pretype;
  int numiters;
  realtype resnorm;
  int last_flag;
  ATimesFn ATimes;
  void* ATData;
  PSetupFn Psetup;
  PSolveFn Psolve;
  void* PData;
  N_Vector s;
  N_Vector r;
  N_Vector p;
  N_Vector z;
  N_Vector Ap;
  int      print_level;
  FILE*    info_file;
};
```

These entries of the *content* field contain the following information:

- `maxl` - number of PCG iterations to allow (default is 5),
- `pretype` - flag for use of preconditioning (default is none),
- `numiters` - number of iterations from the most-recent solve,
- `resnorm` - final linear residual norm from the most-recent solve,
- `last_flag` - last error return flag from an internal function,
- `ATimes` - function pointer to perform Av product,
- `ATData` - pointer to structure for `ATimes`,
- `Psetup` - function pointer to preconditioner setup routine,
- `Psolve` - function pointer to preconditioner solve routine,
- `PData` - pointer to structure for `Psetup` and `Psolve`,
- `s` - vector pointer for supplied scaling matrix (default is `NULL`),
- `r` - a `N_Vector` which holds the preconditioned linear system residual,
- `p, z, Ap` - `N_Vector` used for workspace by the PCG algorithm.
- `print_level` - controls the amount of information to be printed to the info file

- info_file - the file where all informative (non-error) messages will be directed

This solver is constructed to perform the following operations:

- During construction all N_Vector solver data is allocated, with vectors cloned from a template N_Vector that is input, and default solver parameters are set.
- User-facing "set" routines may be called to modify default solver parameters.
- Additional "set" routines are called by the SUNDIALS solver that interfaces with SUNLinSol_PCG to supply the ATimes, PSetup, and Psolve function pointers and s scaling vector.
- In the "initialize" call, the solver parameters are checked for validity.
- In the "setup" call, any non-NULL PSetup function is called. Typically, this is provided by the SUNDIALS solver itself, that translates between the generic PSetup function and the solver-specific routine (solver-supplied or user-supplied).
- In the "solve" call the PCG iteration is performed. This will include scaling and preconditioning if those options have been supplied.

The SUNLinSol_PCG module defines implementations of all "iterative" linear solver operations listed in the section *The SUNLinearSolver API*:

- SUNLinSolGetType_PCG
- SUNLinSolInitialize_PCG
- SUNLinSolSetATimes_PCG
- SUNLinSolSetPreconditioner_PCG
- SUNLinSolSetScalingVectors_PCG – since PCG only supports symmetric scaling, the second N_Vector argument to this function is ignored
- SUNLinSolSetup_PCG
- SUNLinSolSolve_PCG
- SUNLinSolNumIters_PCG
- SUNLinSolResNorm_PCG
- SUNLinSolResid_PCG
- SUNLinSolLastFlag_PCG
- SUNLinSolSpace_PCG
- SUNLinSolFree_PCG

12.16 SUNLinearSolver Examples

There are SUNLinearSolver examples that may be installed for each implementation; these make use of the functions in test_sunlinsol.c. These example functions show simple usage of the SUNLinearSolver family of modules. The inputs to the examples depend on the linear solver type, and are output to stdout if the example is run without the appropriate number of command-line arguments.

The following is a list of the example functions in test_sunlinsol.c:

- Test_SUNLinSolGetType: Verifies the returned solver type against the value that should be returned.
- Test_SUNLinSolInitialize: Verifies that SUNLinSolInitialize can be called and returns successfully.

- `Test_SUNLinSolSetup`: Verifies that `SUNLinSolSetup` can be called and returns successfully.
- `Test_SUNLinSolSolve`: Given a `SUNMatrix` object A, `N_Vector` objects x and b (where $Ax = b$) and a desired solution tolerance `tol`, this routine clones x into a new vector y, calls `SUNLinSolSolve` to fill y as the solution to $Ay = b$ (to the input tolerance), verifies that each entry in x and y match to within `10*tol`, and overwrites x with y prior to returning (in case the calling routine would like to investigate further).
- `Test_SUNLinSolSetATimes` (iterative solvers only): Verifies that `SUNLinSolSetATimes` can be called and returns successfully.
- `Test_SUNLinSolSetPreconditioner` (iterative solvers only): Verifies that `SUNLinSolSetPreconditioner` can be called and returns successfully.
- `Test_SUNLinSolSetScalingVectors` (iterative solvers only): Verifies that `SUNLinSolSetScalingVectors` can be called and returns successfully.
- `Test_SUNLinSolLastFlag`: Verifies that `SUNLinSolLastFlag` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolNumIters` (iterative solvers only): Verifies that `SUNLinSolNumIters` can be called, and outputs the result to `stdout`.
- `Test_SUNLinSolResNorm` (iterative solvers only): Verifies that `SUNLinSolResNorm` can be called, and that the result is non-negative.
- `Test_SUNLinSolResid` (iterative solvers only): Verifies that `SUNLinSolResid` can be called.
- `Test_SUNLinSolSpace` verifies that `SUNLinSolSpace` can be called, and outputs the results to `stdout`.

We'll note that these tests should be performed in a particular order. For either direct or iterative linear solvers, `Test_SUNLinSolInitialize` must be called before `Test_SUNLinSolSetup`, which must be called before `Test_SUNLinSolSolve`. Additionally, for iterative linear solvers `Test_SUNLinSolSetATimes`, `Test_SUNLinSolSetPreconditioner` and `Test_SUNLinSolSetScalingVectors` should be called before `Test_SUNLinSolInitialize`; similarly `Test_SUNLinSolNumIters`, `Test_SUNLinSolResNorm` and `Test_SUNLinSolResid` should be called after `Test_SUNLinSolSolve`. These are called in the appropriate order in all of the example problems.

Chapter 13

Description of the SUNNonlinearSolver Module

SUNDIALS time integration packages are written in terms of generic nonlinear solver operations defined by the SUNNonlinSol API and implemented by a particular SUNNonlinSol module of type `SUNNonlinearSolver`. Users can supply their own SUNNonlinSol module, or use one of the modules provided with SUNDIALS. Depending on the package, nonlinear solver modules can either target system presented in a rootfinding ($F(y) = 0$) or fixed-point ($G(y) = y$) formulation. For more information on the formulation of the nonlinear system(s) see the *ARKode SUNNonlinearSolver interface* section.

The time integrators in SUNDIALS specify a default nonlinear solver module and as such this chapter is intended for users that wish to use a non-default nonlinear solver module or would like to provide their own nonlinear solver implementation. Users interested in using a non-default solver module may skip the description of the SUNNonlinSol API in section *The SUNNonlinearSolver API* and proceeded to the subsequent sections in this chapter that describe the SUNNonlinSol modules provided with SUNDIALS.

For users interested in providing their own SUNNonlinSol module, the following section presents the SUNNonlinSol API and its implementation beginning with the definition of SUNNonlinSol functions in the sections *SUNNonlinearSolver core functions*, *SUNNonlinearSolver set functions* and *SUNNonlinearSolver get functions*. This is followed by the definition of functions supplied to a nonlinear solver implementation in the section *Functions provided by SUNDIALS integrators*. The nonlinear solver return codes are given in the section *SUNNonlinearSolver return codes*. The `SUNNonlinearSolver` type and the generic SUNNonlinSol module are defined in the section *The generic SUNNonlinearSolver module*. Finally, the section *Implementing a Custom SUNNonlinearSolver Module* lists the requirements for supplying a custom SUNNonlinSol module. Users wishing to supply their own SUNNonlinSol module are encouraged to use the SUNNonlinSol implementations provided with SUNDIALS as a template for supplying custom nonlinear solver modules.

13.1 The SUNNonlinearSolver API

The SUNNonlinSol API defines several nonlinear solver operations that enable SUNDIALS integrators to utilize any SUNNonlinSol implementation that provides the required functions. These functions can be divided into three categories. The first are the core nonlinear solver functions. The second group of functions consists of set routines to supply the nonlinear solver with functions provided by the SUNDIALS time integrators and to modify solver parameters. The final group consists of get routines for retrieving nonlinear solver statistics. All of these functions are defined in the header file `sundials/sundials_nonlinearsolver.h`.

13.1.1 SUNNonlinearSolver core functions

The core nonlinear solver functions consist of two required functions to get the nonlinear solver type (`SUNNonlinsSolGetType`) and solve the nonlinear system (`SUNNonlinSolSolve`). The remaining three functions for nonlinear solver initialization (`SUNNonlinSolInitialization`), setup (`SUNNonlinSolSetup`), and destruction (`SUNNonlinSolFree`) are optional.

SUNNonlinearSolver_Type **`SUNNonlinSolGetType`**(SUNNonlinearSolver *NLS*)
The *required* function *`SUNNonlinSolGetType()`* returns the nonlinear solver type.

Arguments:

- *NLS* – a SUNNonlinSol object

Return value: the SUNNonlinSol type identifier (of type `int`) will be one of the following:

- `SUNNONLINEARSOLVER_ROOTFIND` – `0`, the SUNNonlinSol module solves $F(y) = 0$.
- `SUNNONLINEARSOLVER_FIXEDPOINT` – `1`, the SUNNonlinSol module solves $G(y) = y$.

int **`SUNNonlinSolInitialize`**(SUNNonlinearSolver *NLS*)
The *optional* function *`SUNNonlinSolInitialize()`* performs nonlinear solver initialization and may perform any necessary memory allocations.

Arguments:

- *NLS* – a SUNNonlinSol object

Return value: the return value is zero for a successful call and a negative value for a failure.

Notes: It is assumed all solver-specific options have been set prior to calling *`SUNNonlinSolInitialize()`*. SUNNonlinSol implementations that do not require initialization may set this operation to `NULL`.

int **`SUNNonlinSolSetup`**(SUNNonlinearSolver *NLS*, N_Vector *y*, void* *mem*)
The *optional* function *`SUNNonlinSolSetup()`* performs any solver setup needed for a nonlinear solve.

Arguments:

- *NLS* – a SUNNonlinSol object
- *y* – the initial iteration passed to the nonlinear solver.
- *mem* – the SUNDIALS integrator memory structure.

Return value: the return value is zero for a successful call and a negative value for a failure.

Notes: SUNDIALS integrators call `SUNonlinSolSetup()` before each step attempt. SUNNonlinSol implementations that do not require setup may set this operation to `NULL`.

int **`SUNNonlinSolSolve`**(SUNNonlinearSolver *NLS*, N_Vector *y0*, N_Vector *ycor*, N_Vector *w*, realtype *tol*, booleantype *callLSetup*, void **mem*)
The *required* function *`SUNNonlinSolSolve()`* solves the nonlinear system $F(y) = 0$ or $G(y) = y$.

Arguments:

- *NLS* – a SUNNonlinSol object
- *y0* – the predicted value for the new solution state. This *must* remain unchanged throughout the solution process. See the *ARKode SUNNonlinearSolver interface* section for more detail on the nonlinear system formulation.
- *ycor* – on input the initial guess for the correction to the predicted state (zero) and on output the final correction to the predicted state. See the *ARKode SUNNonlinearSolver interface* section for more detail on the nonlinear system formulation.

- *w* – the solution error weight vector used for computing weighted error norms.
- *tol* – the requested solution tolerance in the weighted root-mean-squared norm.
- *callLSetup* – a flag indicating that the integrator recommends for the linear solver setup function to be called.
- *mem* – the SUNDIALS integrator memory structure.

Return value: the return value is zero for a successul solve, a positive value for a recoverable error (i.e., the solve failed and the integrator should reduce the step size and reattempt the step), and a negative value for an unrecoverable error (i.e., the solve failed the and the integrator should halt and return an error to the user).

int **SUNNonlinSolFree**(SUNNonlinearSolver *NLS*)

The *optional* function `SUNNonlinSolFree()` frees any memory allocated by the nonlinear solver.

Arguments:

- *NLS* – a SUNNonlinSol object

Return value: the return value should be zero for a successful call, and a negative value for a failure. SUNNonlinSol implementations that do not allocate data may set this operation to `NULL`.

13.1.2 SUNNonlinearSolver set functions

The following set functions are used to supply nonlinear solver modules with functions defined by the SUNDIALS integrators and to modify solver parameters. Only the routine for setting the nonlinear system defining function (`SUNNonlinSolSetSysFn`) is required. All other set functions are optional.

int **SUNNonlinSolSetSysFn**(SUNNonlinearSolver *NLS*, *SUNNonlinSolSysFn SysFn*)

The *required* function `SUNNonlinSolSetSysFn()` is used to provide the nonlinear solver with the function defining the nonlinear system. This is the function $F(y)$ in $F(y) = 0$ for `SUNNONLINEARSOLVER_ROOTFIND` modules or $G(y)$ in $G(y) = y$ for `SUNNONLINEARSOLVER_FIXEDPOINT` modules.

Arguments:

- *NLS* – a SUNNonlinSol object
- *SysFn* – the function defining the nonlinear system. See the section *Functions provided by SUNDIALS integrators* for the definition of `SUNNonlinSolSysFn()`.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

int **SUNNonlinSolSetLSetupFn**(SUNNonlinearSolver *NLS*, *SUNNonlinSolLSetupFn SetupFn*)

The *optional* function `SUNNonlinSolLSetupFn()` is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver setup function.

Arguments:

- *NLS* – a SUNNonlinSol object
- *SetupFn* – a wrapper function to the SUNDIALS integrator's linear solver setup function. See the section *Functions provided by SUNDIALS integrators* for the definition of `SUNNonlinSolLSetupFn`.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

Notes: The `SUNNonlinSolLSetupFn` function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ (when using SUNLinSol direct linear solvers) or calls

the user-defined preconditioner setup function (when using SUNLinSol iterative linear solvers). SUNNonlinSol implementations that do not require solving this system, do not utilize SUNLinSol linear solvers, or use SUNLinSol linear solvers that do not require setup may set this operation to `NULL`.

int **SUNNonlinSolSetLSolveFn**(SUNNonlinearSolver *NLS*, *SUNNonlinSolLSolveFn SolveFn*)
The *optional* function `SUNNonlinSolSetLSolveFn()` is called by SUNDIALS integrators to provide the nonlinear solver with access to its linear solver solve function.

Arguments:

- *NLS* – a SUNNonlinSol object
- *SolveFn* – a wrapper function to the SUNDIALS integrator's linear solver solve function. See the section *Functions provided by SUNDIALS integrators* for the definition of `SUNNonlinSolLSolveFn`.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

Notes: The `SUNNonlinSolLSolveFn` function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$. SUNNonlinSol implementations that do not require solving this system or do not use SUNLinSol linear solvers may set this operation to `NULL`.

int **SUNNonlinSolSetConvTestFn**(SUNNonlinearSolver *NLS*, *SUNNonlinSolConvTestFn CTestFn*, void* *ctest_data*)
The *optional* function `SUNNonlinSolSetConvTestFn()` is used to provide the nonlinear solver with a function for determining if the nonlinear solver iteration has converged. This is typically called by SUNDIALS integrators to define their nonlinear convergence criteria, but may be replaced by the user.

Arguments:

- *NLS* – a SUNNonlinSol object
- *CTestFn* – a SUNDIALS integrator's nonlinear solver convergence test function. See the section *Functions provided by SUNDIALS integrators* for the definition of `SUNNonlinSolConvTestFn()`.
- *ctest_data* – is a data pointer passed to *CTestFn* every time it is called.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

Notes: SUNNonlinSol implementations utilizing their own convergence test criteria may set this function to `NULL`.

int **SUNNonlinSolSetMaxIters**(SUNNonlinearSolver *NLS*, int *maxiters*)
The *optional* function `SUNNonlinSolSetMaxIters()` sets the maximum number of nonlinear solver iterations. This is typically called by SUNDIALS integrators to define their default iteration limit, but may be adjusted by the user.

Arguments:

- *NLS* – a SUNNonlinSol object
- *maxiters* – the maximum number of nonlinear iterations.

Return value: the return value should be zero for a successful call, and a negative value for a failure (e.g., $maxiters < 1$).

13.1.3 SUNNonlinearSolver get functions

The following get functions allow SUNDIALS integrators to retrieve nonlinear solver statistics. The routines to get the number of iterations in the most recent solve (`SUNNonlinSolGetNumIters`) and number of convergence failures are optional. The routine to get the current nonlinear solver iteration (`SUNNonlinSolGetCurIter`) is

required when using the convergence test provided by the SUNDIALS integrator or when using a SUNLinSol spils linear solver otherwise, `SUNNonlinSolGetCurIter` is optional.

int **`SUNNonlinSolGetNumIters`**(SUNNonlinearSolver *NLS*, long int **niters*)
The *optional* function *`SUNNonlinSolGetNumIters()`* returns the number of nonlinear solver iterations in the most recent solve. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

Arguments:

- *NLS* – a SUNNonlinSol object
- *niters* – the total number of nonlinear solver iterations.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

int **`SUNNonlinSolGetCurIter`**(SUNNonlinearSolver *NLS*, int **iter*)

The function *`SUNNonlinSolGetCurIter()`* returns the iteration index of the current nonlinear solve. This function is *required* when using SUNDIALS integrator-provided convergence tests or when using a SUNLinSol spils linear solver; otherwise it is *optional*.

Arguments:

- *NLS* – a SUNNonlinSol object
- *iter* – the nonlinear solver iteration in the current solve starting from zero.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

int **`SUNNonlinSolGetNumConvFails`**(SUNNonlinearSolver *NLS*, long int **nconvfails*)
The *optional* function *`SUNNonlinSolGetNumConvFails()`* returns the number of nonlinear solver convergence failures in the most recent solve. This is typically called by the SUNDIALS integrator to store the nonlinear solver statistics, but may also be called by the user.

Arguments:

- *NLS* – a SUNNonlinSol object
- *nconvfails* – the total number of nonlinear solver convergence failures.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

13.1.4 Functions provided by SUNDIALS integrators

To interface with SUNNonlinSol modules, the SUNDIALS integrators supply a variety of routines for evaluating the nonlinear system, calling the SUNLinSol setup and solve functions, and testing the nonlinear iteration for convergence. These integrator-provided routines translate between the user-supplied ODE or DAE systems and the generic interfaces to the nonlinear or linear systems of equations that result in their solution. The types for functions provided to a SUNNonlinSol module are defined in the header file `sundials/sundials_nonlinearsolver.h`, and are described below.

typedef int **`(*SUNNonlinSolSysFn)`**(N_Vector *ycor*, N_Vector *F*, void* *mem*)
These functions evaluate the nonlinear system $F(y)$ for `SUNNONLINEARSOLVER_ROOTFIND` type modules or $G(y)$ for `SUNNONLINEARSOLVER_FIXEDPOINT` type modules. Memory for *F* must by be allocated prior to calling this function. The vector *ycor* will be left unchanged.

Arguments:

- *ycor* – is the current correction to the predicted state at which the nonlinear system should be evaluated. See the *ARKode SUNNonlinearSolver interface* section for more detail on the nonlinear system function.

- *F* – is the output vector containing $F(y)$ or $G(y)$, depending on the solver type.
- *mem* – is the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

As discussed in section *ARKode SUNNonlinearSolver interface*, SUNDIALS integrators formulate nonlinear systems as a function of the correction to the predicted solution. On each call to the nonlinear system function the integrator will compute and store the current solution based on the input correction. Additionally, the residual will store the value of the ODE right-hand side function or DAE residual used in computing the nonlinear system. These stored values are then directly used in the integrator-supplied linear solver setup and solve functions as applicable.

typedef int **(*SUNNonlinSolLSetupFn)**(booleantype *jbad*, booleantype* *jcur*, void* *mem*)

These functions are wrappers to the SUNDIALS integrator's function for setting up linear solves with SUNLinSol modules.

Arguments:

- *jbad* – is an input indicating whether the nonlinear solver believes that A has gone stale (`SUNTRUE`) or not (`SUNFALSE`).
- *jcur* – is an output indicating whether the routine has updated the Jacobian A (`SUNTRUE`) or not (`SUNFALSE`).
- *mem* – is the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes: The *SUNNonlinSolLSetupFn* function sets up the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$ (when using SUNLinSol direct linear solvers) or calls the user-defined preconditioner setup function (when using SUNLinSol iterative linear solvers). SUNNonlinSol implementations that do not require solving this system, do not utilize SUNLinSol linear solvers, or use SUNLinSol linear solvers that do not require setup may ignore these functions.

As discussed in the description of `SUNNonlinSolSysFn()`, the linear solver setup function assumes that the nonlinear system function has been called prior to the linear solver setup function as the setup will utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

typedef int **(*SUNNonlinSolLSolveFn)**(N_Vector *b*, void* *mem*)

These functions are wrappers to the SUNDIALS integrator's function for solving linear systems with SUNLinSol modules.

Arguments:

- *b* – contains the right-hand side vector for the linear solve on input and the solution to the linear system on output.
- *mem* – is the SUNDIALS integrator memory structure.

Return value: The return value is zero for a successul solve, a positive value for a recoverable error, and a negative value for an unrecoverable error.

Notes: The *SUNNonlinSolLSolveFn* function solves the linear system $Ax = b$ where $A = \frac{\partial F}{\partial y}$ is the linearization of the nonlinear residual function $F(y) = 0$. SUNNonlinSol implementations that do not require solving this system or do not use SUNLinSol linear solvers may ignore these functions.

As discussed in the description of `SUNNonlinSolSysFn()`, the linear solver solve function assumes that the nonlinear system function has been called prior to the linear solver solve function as the setup may utilize saved values from the nonlinear system evaluation (e.g., the updated solution).

int **(*SUNNonlinSolConvTestFn)**(SUNNonlinearSolver *NLS*, N_Vector *ycor*, N_Vector *del*, realtype *tol*, N_Vector *ewt*, void* *ctest_data*)

These functions are SUNDIALS integrator-specific convergence tests for nonlinear solvers and are typically supplied by each SUNDIALS integrator, but users may supply custom problem-specific versions as desired.

Arguments:

- *NLS* – is the SUNNonlinSol object.
- *ycor* – is the current correction (nonlinear iterate).
- *del* – is the difference between the current and prior nonlinear iterates.
- *tol* – is the nonlinear solver tolerance.
- *ewt* – is the weight vector used in computing weighted norms.
- *ctest_data* – is the data pointer provided to `SUNNonlinSolSetConvTestFn()`.

Return value: The return value of this routine will be a negative value if an unrecoverable error occurred or one of the following:

- `SUN_NLS_SUCCESS` – the iteration is converged.
- `SUN_NLS_CONTINUE` – the iteration has not converged, keep iterating.
- `SUN_NLS_CONV_RECVR` – the iteration appears to be diverging, try to recover.

Notes: The tolerance passed to this routine by SUNDIALS integrators is the tolerance in a weighted root-mean-squared norm with error weight vector `ewt`. SUNNonlinSol modules utilizing their own convergence criteria may ignore these functions.

13.1.5 SUNNonlinearSolver return codes

The functions provided to SUNNonlinSol modules by each SUNDIALS integrator, and functions within the SUNDIALS-provided SUNNonlinSol implementations utilize a common set of return codes, shown in the table below. Here, negative values correspond to non-recoverable failures, positive values to recoverable failures, and zero to a successful call.

Description of the `SUNNonlinearSolver` return codes:

Name	Value	Description
SUN_NLS_SUCCESS	0	successful call or converged solve
SUN_NLS_CONTINUE	901	the nonlinear solver is not converged, keep iterating
SUN_NLS_CONV_RECVR	902	the nonlinear solver appears to be diverging, try to recover
SUN_NLS_MEM_NULL	-901	a memory argument is `NULL`
SUN_NLS_MEM_FAIL	-902	a memory access or allocation failed
SUN_NLS_ILL_INPUT	-903	an illegal input option was provided
SUN_NLS_VECTOROP_ERR	-904	a NVECTOR operation failed
SUN_NLS_EXT_FAIL	-905	an external library call returned an error

13.1.6 The generic SUNNonlinearSolver module

SUNDIALS integrators interact with specific SUNNonlinSol implementations through the generic SUNNonlinSol module on which all other SUNNonlinSol implementations are built. The `SUNNonlinearSolver` type is a pointer to a structure containing an implementation-dependent *content* field and an *ops* field. The type `SUNNonlinearSolver` is defined as follows:

```
typedef struct _generic_SUNNonlinearSolver *SUNNonlinearSolver;

struct _generic_SUNNonlinearSolver {
  void *content;
  struct _generic_SUNNonlinearSolver_Ops *ops;
};
```

where the `_generic_SUNNonlinearSolver_Ops` structure is a list of pointers to the various actual nonlinear solver operations provided by a specific implementation. The `_generic_SUNNonlinearSolver_Ops` structure is defined as

```
struct _generic_SUNNonlinearSolver_Ops {
  SUNNonlinearSolver_Type (*gettype)(SUNNonlinearSolver);
  int                     (*initialize)(SUNNonlinearSolver);
  int                     (*setup)(SUNNonlinearSolver, N_Vector, void*);
  int                     (*solve)(SUNNonlinearSolver, N_Vector, N_Vector,
                                   N_Vector, realtype, booleantype, void*);
  int                     (*free)(SUNNonlinearSolver);
  int                     (*setsysfn)(SUNNonlinearSolver, SUNNonlinSolSysFn);
  int                     (*setlsetupfn)(SUNNonlinearSolver, SUNNonlinSolLSetupFn);
  int                     (*setlsolvefn)(SUNNonlinearSolver, SUNNonlinSolLSolveFn);
  int                     (*setctestfn)(SUNNonlinearSolver, SUNNonlinSolConvTestFn,
                                        void*);
  int                     (*setmaxiters)(SUNNonlinearSolver, int);
  int                     (*getnumiters)(SUNNonlinearSolver, long int*);
  int                     (*getcuriter)(SUNNonlinearSolver, int*);
  int                     (*getnumconvfails)(SUNNonlinearSolver, long int*);
};
```

The generic SUNNonlinSol module defines and implements the nonlinear solver operations defined in Sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions*. These routines are in fact only wrappers to the nonlinear solver operations provided by a particular SUNNonlinSol implementation, which are accessed through the ops field of the `SUNNonlinearSolver` structure. To illustrate this point we show below the implementation of a typical nonlinear solver operation from the generic SUNNonlinSol module, namely `SUNNonlinSolSolve`, which solves the nonlinear system and returns a flag denoting a successful or failed solve:

```
int SUNNonlinSolSolve(SUNNonlinearSolver NLS,
                      N_Vector y0, N_Vector y,
                      N_Vector w, realtype tol,
                      booleantype callLSetup, void* mem)
{
  return((int) NLS->ops->solve(NLS, y0, y, w, tol, callLSetup, mem));
}
```

13.1.7 Implementing a Custom SUNNonlinearSolver Module

A SUNNonlinSol implementation *must* do the following:

- Specify the content of the SUNNonlinSol module.
- Define and implement the required nonlinear solver operations defined in Sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions*. Note that the names of the module routines should be unique to that implementation in order to permit using more than one SUNNonlinSol module (each with different `SUNNonlinearSolver` internal data representations) in the same code.
- Define and implement a user-callable constructor to create a `SUNNonlinearSolver` object.

To aid in the creation of custom SUNNonlinearSolver modules the generic SUNNonlinearSolver module provides the utility functions *SUNNonlinSolNewEmpty()* and SUNNonlinsolFreeEmpty(). When used in custom SUNNonlinearSolver constructors this function will ease the introduction of any new optional nonlinear solver operations to the SUNNonlinearSolver API by ensuring only required operations need to be set.

SUNNonlinearSolver **SUNNonlinSolNewEmpty**()
: This function allocates a new generic SUNNonlinearSolver object and initializes its content pointer and the function pointers in the operations structure to NULL.

  Return value: If successful, this function returns a SUNNonlinearSolver object. If an error occurs when allocating the object, then this routine will return NULL.

void **SUNNonlinSolFreeEmpty**(SUNNonlinearSolver *NLS*)
: This routine frees the generic SUNNonlinearSolver object, under the assumption that any implementation-specific data that was allocated within the underlying content structure has already been freed. It will additionally test whether the ops pointer is NULL, and, if it is not, it will free it as well.

  Arguments:

  - *NLS* – a SUNNonlinearSolver object

Additionally, a SUNNonlinearSolver implementation *may* do the following:

- Define and implement additional user-callable "set" routines acting on the SUNNonlinearSolver object, e.g., for setting various configuration options to tune the performance of the nonlinear solve algorithm.
- Provide additional user-callable "get" routines acting on the SUNNonlinearSolver object, e.g., for returning various solve statistics.

13.2 ARKode SUNNonlinearSolver interface

As discussed in *Mathematical Considerations* integration steps often require the (approximate) solution of a nonlinear system. This system can be formulated as the rootfinding problem

$$\begin{aligned} G(z_i) &\equiv z_i - \gamma f^I\left(t^I_{n,i}, z_i\right) - a_i = 0 && [M = I], \\ G(z_i) &\equiv Mz_i - \gamma f^I\left(t^I_{n,i}, z_i\right) - a_i = 0 && [M \text{ static}], \\ G(z_i) &\equiv M(t^I_{n,i})(z_i - a_i) - \gamma f^I\left(t^I_{n,i}, z_i\right) = 0 && [M \text{ time-dependent}], \end{aligned}$$

where z_i is the i-th stage at time t_i and a_i is known data that depends on the integration method.

Alternately, the nonlinear system above may be formulated as the fixed-point problem

$$z_i = z_i - M(t^I_{n,i})^{-1} G(z_i),$$

where $G(z_i)$ is the variant of the rootfinding problem listed above, and $M(t^I_{n,i})$ may equal either M or I, as applicable.

Rather than solving the above nonlinear systems for the stage value z_i directly, ARKode modules solve for the correction z_{cor} to the predicted stage value z_{pred} so that $z_i = z_{pred} + z_{cor}$. Thus these nonlinear systems rewritten in terms of z_{cor} are

$$\begin{aligned} G(z_{cor}) &\equiv z_{cor} - \gamma f^I\left(t^I_{n,i}, z_i\right) - \tilde{a}_i = 0 && [M = I], \\ G(z_{cor}) &\equiv Mz_{cor} - \gamma f^I\left(t^I_{n,i}, z_i\right) - \tilde{a}_i = 0 && [M \text{ static}], \\ G(z_{cor}) &\equiv M(t^I_{n,i})(z_{cor} - \tilde{a}_i) - \gamma f^I\left(t^I_{n,i}, z_i\right) = 0 && [M \text{ time-dependent}], \end{aligned} \tag{13.1}$$

for the rootfinding problem and

$$z_{cor} = z_{cor} - M(t^I_{n,i})^{-1} G(z_i), \tag{13.2}$$

for the fixed-point problem.

Similarly, in MRIStep (that always assumes $M = I$), we have the nonlinear residual in predictor-corrector form,

$$G(z_{cor}) \equiv z_{cor} - \gamma f^S\left(t^S_{n,i}, z_i\right) - \tilde{a}_i = 0, \tag{13.3}$$

and the corresponding fixed-point problem,

$$z_{cor} = z_{cor} - G(z_i). \tag{13.4}$$

The nonlinear system functions provided by ARKode modules to the nonlinear solver module internally update the current value of the stage based on the input correction vector i.e., $z_i = z_{pred} + z_{cor}$. The updated vector z_i is used when calling the ODE right-hand side function and when setting up linear solves (e.g., updating the Jacobian or preconditioner).

ARKode modules also provide several advanced functions that will not be needed by most users, but might be useful for users who choose to provide their own SUNNonlinearSolver implementation for use by ARKode. These routines provide access to the internal integrator data required to evaluate (13.1) or (13.2) for ARKStep and (13.3) or (13.4) for MRIStep.

13.2.1 ARKStep advanced output functions

int **ARKStepGetCurrentState**(void* *arkode_mem*, N_Vector* *state*)

Returns the current state vector. When called within the computation of a step (i.e., during a nonlinear solve) this is the current stage state vector $z_i = z_{pred} + z_{cor}$. Otherwise this is the current internal solution state vector $y(t)$. In either case the corresponding stage or solution time can be obtained from `ARKStepGetCurrentTime()`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *state* – N_Vector pointer that will get set to the current stage or z_i or solution state vector $y(t)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetCurrentGamma**(void* *arkode_mem*, realtype* *gamma*)

Returns the current value of the scalar γ

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *gamma* – the current value of the scalar γ appearing in the Newton equation $A = M - \gamma J$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetCurrentMassMatrix**(void* *arkode_mem*, SUNMatrix* *M*)

Returns the current mass matrix. For a time dependent mass matrix the corresponding time can be obtained from `ARKStepGetCurrentTime()`.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *M* – SUNMatrix pointer that will get set to the current mass matrix $M(t)$. If a matrix-free method is used the output is `NULL`.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **ARKStepGetNonlinearSystemData**(void* *arkode_mem*, realtype **tcur*, N_Vector **zpred*, N_Vector **z*, N_Vector **Fi*, realtype **gamma*, N_Vector **sdata*, void ***user_data*)

Returns all internal data required to construct the current nonlinear implicit system (13.1) or (13.2):

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *tcur* – value of the independent variable corresponding to implicit stage, $t^I_{n,i}$.
- *zpred* – the predicted stage vector z_{pred} at $t^I_{n,i}$. This vector must not be changed.
- *z* – the stage vector z_i above. This vector may be not current and may need to be filled (see the note below).
- *Fi* – the implicit function evaluated at the current time and state, $f^I(t^I_{n,i}, z_i)$. This vector may be not current and may need to be filled (see the note below).
- *gamma* – current γ for implicit stage calculation.
- *sdata* – accumulated data from previous solution and stages, $\tilde{a}_i$. This vector must not be changed.
- *user_data* – pointer to the user-defined data structure (as specified through `ARKStepSetUserData()`, or `NULL` otherwise)

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

Note: This routine is intended for users who whish to attach a custom `SUNNonlinSolSysFn` to an existing SUNNonlinearSolver object (through a call to `SUNNonlinSolSetSysFn()`) or who need access to nonlinear system data to compute the nonlinear system fucntion as part of a custom SUNNonlinearSolver object.

When supplying a custom `SUNNonlinSolSysFn` to an existing SUNNonlinearSolver object, the user should call `ARKStepGetNonlinearSystemData()` **inside** the nonlinear system function to access the requisite data for evaluting the nonlinear systen function of their choosing. Additionlly, if the SUNNonlinearSolver object (existing or custom) leverages the `SUNNonlinSolLSetupFn` and/or `SUNNonlinSolLSolveFn` functions supplied by ARKStep (through calls to `SUNNonlinSolSetLSetupFn()` and `SUNNonlinSolSetLSolveFn()` respectively) the vectors *z* and *Fi* **must be filled** in by the user's `SUNNonlinSolSysFn` with the current state and corresponding evaluation of the right-hand side function respectively i.e.,

$$z = z_{pred} + z_{cor},$$
$$Fi = f^I\left(t^I_{n,i}, z_i\right),$$

where z_{cor} was the first argument supplied to the *SUNNonlinSolSysFn*.

If this function is called as part of a custom linear solver (i.e., the default *SUNNonlinSolSysFn* is used) then the vectors *z* and *Fi* are only current when *ARKStepGetNonlinearSystemData()* is called after an evaluation of the nonlinear system function.

---

int **ARKStepComputeState**(void* *arkode_mem*, N_Vector *zcor*, N_Vector *z*)

Computes the current stage state vector using the stored prediction and the supplied correction from the nonlinear solver i.e., $z_i(t) = z_{pred} + z_{cor}$.

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *zcor* – the correction from the nonlinear solver
- *z* – on output, the current stage state vector z_i

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

13.2.2 MRIStep advanced output functions

int **MRIStepGetCurrentState**(void* *arkode_mem*, N_Vector* *state*)

Returns the current state vector. When called within the computation of a step (i.e., during a nonlinear solve) this is the current stage state vector $z_i = z_{pred} + z_{cor}$. Otherwise this is the current internal solution state vector $y(t)$. In either case the corresponding stage or solution time can be obtained from *ARKStepGetCurrentTime()*.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *state* – N_Vector pointer that will get set to the current stage z_i or solution state vector $y(t)$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

int **MRIStepGetCurrentGamma**(void* *arkode_mem*, realtype* *gamma*)

Returns the current value of the scalar γ

Arguments:

- *arkode_mem* – pointer to the ARKStep memory block.
- *gamma* – the current value of the scalar γ appearing in the Newton equation $A = I - \gamma J$.

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the ARKStep memory was `NULL`

int **MRIStepGetNonlinearSystemData**(void* *arkode_mem*, realtype **tcur*, N_Vector **zpred*, N_Vector **z*, N_Vector **F*, realtype **gamma*, N_Vector **sdata*, void ***user_data*)

Returns all internal data required to construct the current nonlinear implicit system (13.3) or (13.4):

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *tcur* – value of independent variable corresponding to slow stage ($t^S_{n,i}$ above).
- *zpred* – predicted nonlinear solution (z_{pred} above). This vector must not be changed.
- *z* – stage vector (z_i above). This vector may be not current and may need to be filled (see the note below).
- *F* – memory available for evaluating the slow RHS ($f^S(t^S_{n,i}, z_i)$ above). This vector may be not current and may need to be filled (see the note below).
- *gamma* – current γ for slow stage calculation.
- *sdata* – accumulated data from previous solution and stages ($\tilde{a}_i$ above). This vector must not be changed.
- *user_data* – pointer to the user-defined data structure (as specified through `MRIStepSetUserData()`, or `NULL` otherwise).

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

Note: This routine is intended for users who whish to attach a custom `SUNNonlinSolSysFn` to an existing SUNNonlinearSolver object (through a call to `SUNNonlinSolSetSysFn()`) or who need access to nonlinear system data to compute the nonlinear system fucntion as part of a custom SUNNonlinearSolver object.

When supplying a custom `SUNNonlinSolSysFn` to an existing SUNNonlinearSolver object, the user should call `MRIStepGetNonlinearSystemData()` **inside** the nonlinear system function to access the requisite data for evaluting the nonlinear systen function of their choosing. Additionlly, if the SUNNonlinearSolver object (existing or custom) leverages the `SUNNonlinSolLSetupFn` and/or `SUNNonlinSolLSolveFn` functions supplied by MRIStep (through calls to `SUNNonlinSolSetLSetupFn()` and `SUNNonlinSolSetLSolveFn()` respectively) the vectors *z* and *F* **must be filled** in by the user's `SUNNonlinSolSysFn` with the current state and corresponding evaluation of the right-hand side function respectively i.e.,

$$z = z_{pred} + z_{cor},$$
$$F = f^S\left(t^S_{n,i}, z_i\right),$$

where z_{cor} was the first argument supplied to the `SUNNonlinSolSysFn`.

If this function is called as part of a custom linear solver (i.e., the default `SUNNonlinSolSysFn` is used) then the vectors *z* and *F* are only current when `MRIStepGetNonlinearSystemData()` is called after an evaluation of the nonlinear system function.

---

int **`MRIStepComputeState`**(void* *arkode_mem*, N_Vector *zcor*, N_Vector *z*)

Computes the current stage state vector using the stored prediction and the supplied correction from the nonlinear solver i.e., $z_i = z_{pred} + z_{cor}$.

Arguments:

- *arkode_mem* – pointer to the MRIStep memory block.
- *zcor* – the correction from the nonlinear solver
- *z* – on output, the current stage state vector z_i

Return value:

- *ARK_SUCCESS* if successful
- *ARK_MEM_NULL* if the MRIStep memory was `NULL`

13.3 The SUNNonlinearSolver_Newton implementation

This section describes the SUNNonlinSol implementation of Newton's method. To access the SUNNonlinSol_Newton module, include the header file `sunnonlinsol/sunnonlinsol_newton.h`. We note that the SUNNonlinSol_Newton module is accessible from SUNDIALS integrators *without* separately linking to the `libsundials_sunnonlinsolnewton` module library.

13.3.1 SUNNonlinearSolver_Newton description

To find the solution to

$$F(y) = 0 \tag{13.5}$$

given an initial guess $y^{(0)}$, Newton's method computes a series of approximate solutions

$$y^{(m+1)} = y^{(m)} + \delta^{(m+1)}$$

where m is the Newton iteration index, and the Newton update $\delta^{(m+1)}$ is the solution of the linear system

$$A(y^{(m)})\delta^{(m+1)} = -F(y^{(m)})\,, \tag{13.6}$$

in which A is the Jacobian matrix

$$A \equiv \partial F/\partial y\,. \tag{13.7}$$

Depending on the linear solver used, the SUNNonlinSol_Newton module will employ either a Modified Newton method, or an Inexact Newton method *[B1987]*, *[BS1990]*, *[DES1982]*, *[DS1996]*, *[K1995]*. When used with a direct linear solver, the Jacobian matrix A is held constant during the Newton iteration, resulting in a Modified Newton method. With a matrix-free iterative linear solver, the iteration is an Inexact Newton method.

In both cases, calls to the integrator-supplied `SUNNonlinSolLSetupFn()` function are made infrequently to amortize the increased cost of matrix operations (updating A and its factorization within direct linear solvers, or updating the preconditioner within iterative linear solvers). Specifically, SUNNonlinSol_Newton will call the `SUNNonlinSolLSetupFn()` function in two instances:

1. when requested by the integrator (the input `callLSetSetup` is `SUNTRUE`) before attempting the Newton iteration, or
2. when reattempting the nonlinear solve after a recoverable failure occurs in the Newton iteration with stale Jacobian information (`jcur` is `SUNFALSE`). In this case, SUNNonlinSol_Newton will set `jbad` to `SUNTRUE` before calling the `SUNNonlinSolLSetupFn()` function.

Whether the Jacobian matrix A is fully or partially updated depends on logic unique to each integrator-supplied `SUNNonlinSolSetupFn()` routine. We refer to the discussion of nonlinear solver strategies provided in Chapter *Mathematical Considerations* for details on this decision.

The default maximum number of iterations and the stopping criteria for the Newton iteration are supplied by the SUNDIALS integrator when SUNNonlinSol_Newton is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling the `SUNNonlinSolSetMaxIters()` and/or `SUNNonlinSolSetConvTestFn()` functions after attaching the SUNNonlinSol_Newton object to the integrator.

13.3.2 SUNNonlinearSolver_Newton functions

The SUNNonlinSol_Newton module provides the following constructor for creating the `SUNNonlinearSolver` object.

SUNNonlinearSolver **`SUNNonlinSol_Newton`**(N_Vector *y*)
The function `SUNNonlinSol_Newton()` creates a `SUNNonlinearSolver` object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ using Newton's method.

Arguments:

- *y* – a template for cloning vectors needed within the solver.

Return value: a SUNNonlinSol object if the constructor exits successfully, otherwise it will be `NULL`.

The SUNNonlinSol_Newton module implements all of the functions defined in sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions* except for the `SUNNonlinSolSetup()` function. The SUNNonlinSol_Newton functions have the same names as those defined by the generic SUNNonlinSol API with `_Newton` appended to the function name. Unless using the SUNNonlinSol_Newton module as a standalone nonlinear solver the generic functions defined in sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions* should be called in favor of the SUNNonlinSol_Newton-specific implementations.

The SUNNonlinSol_Newton module also defines the following additional user-callable function.

int **`SUNNonlinSolGetSysFn_Newton`**(SUNNonlinearSolver *NLS*, *SUNNonlinSolSysFn* **SysFn*)
The function `SUNNonlinSolGetSysFn_Newton()` returns the residual function that defines the nonlinear system.

Arguments:

- *NLS* – a SUNNonlinSol object
- *SysFn* – the function defining the nonlinear system.

Return value: the return value should be zero for a successful call, and a negative value for a failure.

Notes: This function is intended for users that wish to evaluate the nonlinear residual in a custom convergence test function for the SUNNonlinSol_Newton module. We note that SUNNonlinSol_Newton will not leverage the results from any user calls to *SysFn*.

int **`SUNNonlinSolSetInfoFile_Newton`**(SUNNonlinearSolver *NLS*, FILE* *info_file*)
The function `SUNNonlinSolSetInfoFile_Newton()` sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *NLS* – a SUNNonlinSol object
- ***info_file* – pointer to output file (`stdout` by default);** a `NULL` input will disable output

Return value:

- *SUN_NLS_SUCCESS* if successful

- *SUN_NLS_MEM_NULL* if the SUNNonlinearSolver memory was `NULL`
- *SUN_NLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the file pointer is set to `stdout`.

SUNDIALS must be built with the CMake option "SUNDIALS_BUILD_WITH_MONITORING", to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **SUNNonlinSolSetPrintLevel_Newton**(SUNNonlinearSolver *NLS*, int *print_level*)
The function *SUNNonlinSolSetPrintLevel_Newton()* specifies the level of verbosity of the output.

Arguments:

- *NLS* – a SUNNonlinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each nonlinear iteration the residual norm is printed

Return value:

- *SUN_NLS_SUCCESS* if successful
- *SUN_NLS_MEM_NULL* if the SUNNonlinearSolver memory was `NULL`
- *SUN_NLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option "SUNDIALS_BUILD_WITH_MONITORING", to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

13.3.3 SUNNonlinearSolver_Newton content

The *content* field of the SUNNonlinSol_Newton module is the following structure.

```
struct _SUNNonlinearSolverContent_Newton {

  SUNNonlinSolSysFn      Sys;
  SUNNonlinSolLSetupFn   LSetup;
  SUNNonlinSolLSolveFn   LSolve;
  SUNNonlinSolConvTestFn CTest;

  N_Vector    delta;
  booleantype jcur;
  int         curiter;
  int         maxiters;
  long int    niters;
  long int    nconvfails;
  void*       ctest_data;

  int         print_level;
  FILE*       info_file;
};
```

These entries of the *content* field contain the following information:

- `Sys` – the function for evaluating the nonlinear system,
- `LSetup` – the package-supplied function for setting up the linear solver,
- `LSolve` – the package-supplied function for performing a linear solve,
- `CTest` – the function for checking convergence of the Newton iteration,
- `delta` – the Newton iteration update vector,
- `jcur` – the Jacobian status (`SUNTRUE` = current, `SUNFALSE` = stale),
- `curiter` – the current number of iterations in the solve attempt,
- `maxiters` – the maximum number of Newton iterations allowed in a solve,
- `niters` – the total number of nonlinear iterations across all solves,
- `nconvfails` – the total number of nonlinear convergence failures across all solves,
- `ctest_data` – the data pointer passed to the convergence test function.
- `print_level` - controls the amount of information to be printed to the info file
- `info_file` - the file where all informative (non-error) messages will be directed

13.3.4 SUNNonlinearSolver_Newton Fortran interface

For SUNDIALS integrators that include a Fortran interface, the SUNNonlinSol_Newton module also includes a Fortran-callable function for creating a `SUNNonlinearSolver` object.

subroutine FSUNNewtonInit(*CODE*, *IER*)

The function *FSUNNewtonInit()* can be called for Fortran programs to create a `SUNNonlinearSolver` object for use with SUNDIALS integrators to solve nonlinear systems of the form $F(y) = 0$ with Newton's method.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, ARKode=4.
- *IER* (`int`, output) – return flag (0 success, -1 for failure). See printed message for details in case of failure.

13.4 The SUNNonlinearSolver_FixedPoint implementation

This section describes the SUNNonlinSol implementation of a fixed point (functional) iteration with optional Anderson acceleration. To access the SUNNonlinSol_FixedPoint module, include the header file `sunnonlinsol/sunnonlinsol_fixedpoint.h`. We note that the SUNNonlinSol_FixedPoint module is accessible from SUNDIALS integrators *without* separately linking to the `libsundials_sunnonlinsolfixedpoint` module library.

13.4.1 SUNNonlinearSolver_FixedPoint description

To find the solution to

$$G(y) = y \tag{13.8}$$

given an initial guess $y^{(0)}$, the fixed point iteration computes a series of approximate solutions

$$y^{(n+1)} = G(y^{(n)}) \tag{13.9}$$

where n is the iteration index. The convergence of this iteration may be accelerated using Anderson's method *[A1965]*, *[WN2011]*, *[FS2009]*, *[LWWY2012]*. With Anderson acceleration using subspace size m, the series of approximate solutions can be formulated as the linear combination

$$y^{(n+1)} = \beta \sum_{i=0}^{m_n} \alpha_i^{(n)} G(y^{(n-m_n+i)}) + (1-\beta) \sum_{i=0}^{m_n} \alpha_i^{(n)} y_{n-m_n+i} \tag{13.10}$$

where $m_n = \min\{m, n\}$ and the factors

$$\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)})$$

solve the minimization problem $\min_\alpha \|F_n \alpha^T\|_2$ under the constraint that $\sum_{i=0}^{m_n} \alpha_i = 1$ where

$$F_n = (f_{n-m_n}, \ldots, f_n)$$

with $f_i = G(y^{(i)}) - y^{(i)}$. Due to this constraint, in the limit of $m = 0$ the accelerated fixed point iteration formula (13.10) simplifies to the standard fixed point iteration (13.9).

Following the recommendations made in *[WN2011]*, the SUNNonlinSol_FixedPoint implementation computes the series of approximate solutions as

$$y^{(n+1)} = G(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i} - (1-\beta)(f(y^{(n)}) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta f_{n-m_n+i}) \tag{13.11}$$

with $\Delta g_i = G(y^{(i+1)}) - G(y^{(i)})$ and where the factors

$$\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})$$

solve the unconstrained minimization problem $\min_\gamma \|f_n - \Delta F_n \gamma^T\|_2$ where

$$\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1}),$$

with $\Delta f_i = f_{i+1} - f_i$. The least-squares problem is solved by applying a QR factorization to $\Delta F_n = Q_n R_n$ and solving $R_n \gamma = Q_n^T f_n$.

The acceleration subspace size m is required when constructing the SUNNonlinSol_FixedPoint object. The default maximum number of iterations and the stopping criteria for the fixed point iteration are supplied by the SUNDIALS integrator when SUNNonlinSol_FixedPoint is attached to it. Both the maximum number of iterations and the convergence test function may be modified by the user by calling `SUNNonlinSolSetMaxIters()` and `SUNNonlinSolSetConvTestFn()` functions after attaching the SUNNonlinSol_FixedPoint object to the integrator.

13.4.2 SUNNonlinearSolver_FixedPoint functions

The SUNNonlinSol_FixedPoint module provides the following constructor for creating the `SUNNonlinearSolver` object.

SUNNonlinearSolver **SUNNonlinSol_FixedPoint** (N_Vector *y*, int *m*)
The function *SUNNonlinSol_FixedPoint()* creates a `SUNNonlinearSolver` object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$.

Arguments:

- *y* – a template for cloning vectors needed within the solver.
- *m* – the number of acceleration vectors to use.

Return value: a SUNNonlinSol object if the constructor exits successfully, otherwise it will be `NULL`.

Since the accelerated fixed point iteration (13.9) does not require the setup or solution of any linear systems, the SUNNonlinSol_FixedPoint module implements all of the functions defined in sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions* except for the *SUNNonlinSolSetup()*, *SUNNonlinSolSetLSetupFn()*, and *SUNNonlinSolSetLSolveFn()* functions, that are set to `NULL`. The SUNNonlinSol_FixedPoint functions have the same names as those defined by the generic SUNNonlinSol API with `_FixedPoint` appended to the function name. Unless using the SUNNonlinSol_FixedPoint module as a standalone nonlinear solver the generic functions defined in sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions* should be called in favor of the SUNNonlinSol_FixedPoint-specific implementations.

The SUNNonlinSol_FixedPoint module also defines the following additional user-callable functions.

int **SUNNonlinSolGetSysFn_FixedPoint** (SUNNonlinearSolver *NLS*, *SUNNonlinSolSysFn* ***SysFn*)
The function *SUNNonlinSolGetSysFn_FixedPoint()* returns the fixed-point function that defines the nonlinear system.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *SysFn* – the function defining the nonlinear system.

Return value: The return value is zero for a successful call, and a negative value for a failure.

Notes: This function is intended for users that wish to evaluate the fixed-point function in a custom convergence test function for the SUNNonlinSol_FixedPoint module. We note that SUNNonlinSol_FixedPoint will not leverage the results from any user calls to *SysFn*.

int **SUNNonlinSolSetDamping_FixedPoint** (SUNNonlinearSolver *NLS*, realtype *beta*)
The function *SUNNonlinSolSetDamping_FixedPoint()* sets the damping parameter β to use with Anderson acceleration. By default damping is disabled i.e., $\beta = 1.0$.

Arguments:

- *NLS* – a SUNNonlinSol object.
- *beta* – the damping parameter $0 < \beta \leq 1$.

Return value: The return value is zero for a successful call, `SUN_NLS_MEM_NULL` if `NLS` is `NULL`, or `SUN_NLS_ILL_INPUT` if `beta` is negative.

Notes: A `beta` value should be great than zero and less than one if damping is to be used. A value of one or more will disable damping.

int **SUNNonlinSolSetInfoFile_FixedPoint** (SUNNonlinearSolver *NLS*, FILE* *info_file*)
The function *SUNNonlinSolSetInfoFile_FixedPoint()* sets the output file where all informative (non-error) messages should be directed.

Arguments:

- *NLS* – a SUNNonlinSol object
- ***info_file* – pointer to output file (`stdout` by default);** a `NULL` input will disable output

Return value:

- *SUN_NLS_SUCCESS* if successful
- *SUN_NLS_MEM_NULL* if the SUNNonlinearSolver memory was `NULL`
- *SUN_NLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the file pointer is set to `stdout`.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

int **`SUNNonlinSolSetPrintLevel_FixedPoint`**(SUNNonlinearSolver *NLS*, int *print_level*)
The function *`SUNNonlinSolSetPrintLevel_FixedPoint()`* specifies the level of verbosity of the output.

Arguments:

- *NLS* – a SUNNonlinSol object
- *print_level* – flag indicating level of verbosity; must be one of:
  - 0, no information is printed (default)
  - 1, for each nonlinear iteration the residual norm is printed

Return value:

- *SUN_NLS_SUCCESS* if successful
- *SUN_NLS_MEM_NULL* if the SUNNonlinearSolver memory was `NULL`
- *SUN_NLS_ILL_INPUT* if SUNDIALS was not built with monitoring enabled, or the print level value was invalid

Notes: This function is intended for users that wish to monitor the nonlinear solver progress. By default, the print level is 0.

SUNDIALS must be built with the CMake option ''SUNDIALS_BUILD_WITH_MONITORING'', to utilize this function. See section *Configuration options (Unix/Linux)* for more information.

13.4.3 SUNNonlinearSolver_FixedPoint content

The *content* field of the SUNNonlinSol_FixedPoint module is the following structure.

```
struct _SUNNonlinearSolverContent_FixedPoint {

  SUNNonlinSolSysFn      Sys;
  SUNNonlinSolConvTestFn CTest;

  int          m;
  int         *imap;
  realtype    *R;
  booleantype  damping
  realtype     beta
  realtype    *gamma;
```

```
  realtype    *cvals;
  N_Vector    *df;
  N_Vector    *dg;
  N_Vector    *q;
  N_Vector    *Xvecs;
  N_Vector     yprev;
  N_Vector     gy;
  N_Vector     fold;
  N_Vector     gold;
  N_Vector     delta;
  int          curiter;
  int          maxiters;
  long int     niters;
  long int     nconvfails;
  void        *ctest_data;
  int          print_level;
  FILE*        info_file;
};
```

The following entries of the *content* field are always allocated:

- `Sys` – function for evaluating the nonlinear system,
- `CTest` – function for checking convergence of the fixed point iteration,
- `yprev` – `N_Vector` used to store previous fixed-point iterate,
- `gy` – `N_Vector` used to store $G(y)$ in fixed-point algorithm,
- `delta` – `N_Vector` used to store difference between successive fixed-point iterates,
- `curiter` – the current number of iterations in the solve attempt,
- `maxiters` – the maximum number of fixed-point iterations allowed in a solve,
- `niters` – the total number of nonlinear iterations across all solves,
- `nconvfails` – the total number of nonlinear convergence failures across all solves,
- `ctest_data` – the data pointer passed to the convergence test function, and
- `m` – number of acceleration vectors.
- `print_level` - controls the amount of information to be printed to the info file
- `info_file` - the file where all informative (non-error) messages will be directed

If Anderson acceleration is requested (i.e., $m > 0$ in the call to *SUNNonlinSol_FixedPoint()*), then the following items are also allocated within the *content* field:

- `imap` – index array used in acceleration algorithm (length `m`),
- `damping` – a flag indicating if damping is enabled,
- `beta` – the damping parameter,
- `R` – small matrix used in acceleration algorithm (length `m*m`),
- `gamma` – small vector used in acceleration algorithm (length `m`),
- `cvals` – small vector used in acceleration algorithm (length `m+1`),
- `df` – array of `N_Vectors` used in acceleration algorithm (length `m`),
- `dg` – array of `N_Vectors` used in acceleration algorithm (length `m`),

- `q` – array of `N_Vectors` used in acceleration algorithm (length `m`),
- `Xvecs` – `N_Vector` pointer array used in acceleration algorithm (length `m+1`),
- `fold` – `N_Vector` used in acceleration algorithm, and
- `gold` – `N_Vector` used in acceleration algorithm.

13.4.4 SUNNonlinearSolver_FixedPoint Fortran interface

For SUNDIALS integrators that include a Fortran interface, the SUNNonlinSol_FixedPoint module also includes a Fortran-callable function for creating a `SUNNonlinearSolver` object.

subroutine FSUNFixedPointInit(*CODE*, *M*, *IER*)

The function *FSUNFixedPointInit()* can be called for Fortran programs to create a `SUNNonlinearSolver` object for use with SUNDIALS integrators to solve nonlinear systems of the form $G(y) = y$.

This routine must be called *after* the `N_Vector` object has been initialized.

Arguments:

- *CODE* (`int`, input) – flag denoting the SUNDIALS solver this matrix will be used for: CVODE=1, IDA=2, ARKode=4.
- *M* (`int`, input) – the number of acceleration vectors.
- *IER* (`int`, output) – return flag (0 success, -1 for failure). See printed message for details in case of failure.

13.5 The SUNNonlinearSolver_PetscSNES implementation

This section describes the SUNNonlinSol interface to the PETSc SNES nonlinear solver(s).To enable the SUNonlinSol_PetscSNES module, SUNDIALS must be configured to use PETSc. Instructions on how to do thus are given in Chapter *Building with PETSc*. To access the SUNNonlinSol_PetscSNES module, include the header file `sunnonlinsol/sunnonlinsol_petscsnes.h`. The library to link to is `libsundials_sunnonlinsolpetsc.lib` where `.lib` is typically `.so` for shared libaries and `.a` for static libraries. Users of the `SUNNonlinearSolver_PetscSNES` should also see the section *The NVECTOR_PETSC Module* which discusses the NVECTOR interface to the PETSc Vec API.

13.5.1 SUNNonlinearSolver_PetscSNES description

The `SUNNonlinearSolver_PetscSNES` implementation allows users to utilize a PETSc SNES nonlinear solver to solve the nonlinear systems that arise in the SUNDIALS integrators. Since SNES uses the KSP linear solver interface underneath it, the `SUNNonlinearSolver_PetscSNES` implementation does not interface with SUNDIALS linear solvers. Instead, users should set nonlinear solver options, linear solver options, and preconditioner options through the PETSc SNES, KSP, and PC APIs.

Important usage notes for the ‘‘SUNNonlinearSolver_PetscSNES‘‘ implementation are provided below:

- The `SUNNonlinearSolver_PetscSNES` implementation handles calling `SNESSetFunction` at construction. The actual residual function $F(y)$ is set by the SUNDIALS integrator when the `SUNNonlinearSolver_PetscSNES` object is attached to it. Therefore, a user should not call `SNESSetFunction` on a `SNES` object that is being used with `SUNNonlinearSolver_PetscSNES`. For these reasons, it is recommended, although not always necessary, that the user calls `SUNNonlinSol_PetscSNES` with the new `SNES` object immediately after calling `SNESCreate`.

- The number of nonlinear iterations is tracked by SUNDIALS separately from the count kept by SNES. As such, the function `SUNNonlinSolGetNumIters` reports the cumulative number of iterations across the lifetime of the `SUNNonlinearSolver` object.
- Some "converged" and "diverged" convergence reasons returned by SNES are treated as recoverable convergence failures by SUNDIALS. Therefore, the count of convergence failures returned by `SUNNonlinSolGetNumConvFails` will reflect the number of recoverable convergence failures as determined by SUNDIALS, and may differ from the count returned by `SNESGetNonlinearStepFailures`.
- The `SUNNonlinearSolver_PetscSNES` module is not currently compatible with the CVODES or IDAS staggered or simultaneous sensitivity strategies.

13.5.2 SUNNonlinearSolver_PetscSNES functions

The `SUNNonlinearSolver_PetscSNES` module provides the following constructor for creating a `SUNNonlinearSolver` object.

SUNNonlinearSolver **`SUNNonlinSol_PetscSNES`** (N_Vector *y*, SNES *snes*)
The function `SUNNonlinSol_PetscSNES` creates a `SUNNonlinearSolver` object that wraps a PETSc `SNES` object for use with SUNDIALS. This will call `SNESSetFunction` on the provided `SNES` object.

Arguments:

- *snes* – a PETSc `SNES` object
- *y* – a `N_Vector` object of type NVECTOR_PETSC that is used as a template for the residual vector

Return value: a SUNNonlinSol object if the constructor exits successfully, otherwise it will be `NULL`.

This function calls ''SNESSetFunction'' and will overwrite whatever function was previously set. Users should not call ''SNESSetFunction'' on the ''SNES'' object provided to the constructor.

The `SUNNonlinSol_PetscSNES` module implements all of the functions defined in sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions* except for `SUNNonlinSolSetup`, `SUNNonlinSolSetLSetupFn`, `SUNNonlinSolSetLSolveFn`, `SUNNonlinSolSetConvTestFn`, and `SUNNonlinSolSetMaxIters`.

The `SUNNonlinSol_PetscSNES` functions have the same names as those defined by the generic `SUNNonlinearSolver` API with `_PetscSNES` appended to the function name. Unless using the `SUNNonlinSol_PetscSNES` module as a standalone nonlinear solver the generic functions defined in sections *SUNNonlinearSolver core functions* through *SUNNonlinearSolver get functions* should be called in favor of the `SUNNonlinSol_PetscSNES` specific implementations.

The `SUNNonlinSol_PetscSNES` module also defines the following additional user-callable functions.

int **`SUNNonlinSolGetSNES_PetscSNES`** (SUNNonlinearSolver *NLS*, SNES* *snes*)
The function `SUNNonlinSolGetSNES_PetscSNES` gets the `SNES` object that was wrapped.

Arguments:

- *NLS* – a `SUNNonlinearSolver` object
- *snes* – a pointer to a PETSc `SNES` object that will be set upon return

Return value: The return value (of type `int`) should be zero for a successful call, and a negative value for a failure.

int **`SUNNonlinSolGetPetscError_PetscSNES`** (SUNNonlinearSolver *NLS*, PestcErrorCode* *error*)
The function `SUNNonlinSolGetPetscError_PetscSNES` gets the last error code returned by the last internal call to a PETSc API function.

Arguments:

- *NLS* – a `SUNNonlinearSolver` object
- *error* – a pointer to a PETSc error integer that will be set upon return

Return value: The return value (of type `int`) should be zero for a successful call, and a negative value for a failure.

int **`SUNNonlinSolGetSysFn_PetscSNES`**(SUNNonlinearSolver *NLS*, *SUNNonlinSolSysFn* SysFn*)
The function `SUNNonlinSolGetSysFn_PetscSNES` returns the residual function that defines the nonlinear system.

Arguments:

- *NLS* – a `SUNNonlinearSolver` object
- *SysFn* – the function defining the nonlinear system

Return value: The return value (of type `int`) should be zero for a successful call, and a negative value for a failure.

13.5.3 SUNNonlinearSolver_PetscSNES content

The *content* field of the SUNNonlinSol_Newton module is the following structure.

```
struct _SUNNonlinearSolverContent_PetscSNES {
  int sysfn_last_err;
  PetscErrorCode petsc_last_err;
  long int nconvfails;
  long int nni;
  void *imem;
  SNES snes;
  Vec r;
  N_Vector y, f;
  SUNNonlinSolSysFn Sys;
};
```

These entries of the *content* field contain the following information:

- `sysfn_last_err` – last error returned by the system defining function,
- `petsc_last_err` – last error returned by PETSc
- `nconvfails` – number of nonlinear converge failures (recoverable or not),
- `nni` – number of nonlinear iterations,
- `imem` – SUNDIALS integrator memory,
- `snes` – PETSc `SNES` object,
- `r` – the nonlinear residual,
- `y` – wrapper for PETSc vectors used in the system function,
- `f` – wrapper for PETSc vectors used in the system function,
- `Sys` – nonlinear system definining function.

Chapter 14

Tools for Memory Management

To support applications which leverage memory pools, or utilize a memory abstraction layer, sundials provides a set of utilities we will collectively refer to as the SUNMemoryHelper API. The goal of this API is to allow users to leverage operations defined by native sundials data structures while allowing the user to have finer-grained control of the memory management.

14.1 The SUNMemoryHelper API

This API consists of three new sundials types: `SUNMemoryType`, `SUNMemory`, and `SUNMemoryHelper`, which we now define.

The `SUNMemory` structure wraps a pointer to actual data. This structure is defined as

```
typedef struct _SUNMemory
{
  void*         ptr;
  SUNMemoryType type;
  booleantype   own;
} *SUNMemory;
```

The `SUNMemoryType` type is an enumeration that defines the four supported memory types:

```
typedef enum
{
  SUNMEMTYPE_HOST,      /* pageable memory accessible on the host     */
  SUNMEMTYPE_PINNED,    /* page-locked memory accesible on the host   */
  SUNMEMTYPE_DEVICE,    /* memory accessible from the device          */
  SUNMEMTYPE_UVM        /* memory accessible from the host or device  */
} SUNMemoryType;
```

Finally, the `SUNMemoryHelper` structure is defined as

```
struct _SUNMemoryHelper
{
  void*               content;
  SUNMemoryHelper_Ops ops;
} *SUNMemoryHelper;
```

where `SUNMemoryHelper_Ops` is defined as

```
typedef struct _SUNMemoryHelper_Ops
{
  /* operations that implementations are required to provide */
  int              (*alloc)(SUNMemoryHelper, SUNMemory* memptr
                            size_t mem_size, SUNMemoryType mem_type);
  int              (*dealloc)(SUNMemoryHelper, SUNMemory mem);
  int              (*copy)(SUNMemoryHelper, SUNMemory dst, SUNMemory src,
                           size_t mem_size);

  /* operations that provide default implementations */
  int              (*copyasync)(SUNMemoryHelper, SUNMemory dst, SUNMemory src,
                                size_t mem_size, void* ctx);
  SUNMemoryHelper  (*clone)(SUNMemoryHelper);
  int              (*destroy)(SUNMemoryHelper);
} *SUNMemoryHelper_Ops;
```

14.1.1 Implementation defined operations

The SUNMemory API also defines the following operations which do require a SUNMemoryHelper instance and **require** the implementation to define them:

SUNMemory **SUNMemoryHelper_Alloc**(SUNMemoryHelper *helper*, SUNMemory* *memptr*,
size_t *mem_size*, SUNMemoryType *mem_type*)

Allocates a `SUNMemory` object whose `ptr` field is allocated for `mem_size` bytes and is of type `mem_type`. The new object will have ownership of `ptr` and will be deallocated when `SUNMemoryHelper_Dealloc` is called.

Arguments:

- *helper* – the `SUNMemoryHelper` object
- *memptr* – pointer to the allocated `SUNMemory`
- *mem_size* – the size in bytes of the `ptr`
- *mem_type* – the `SUNMemoryType` of the `ptr`

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Dealloc**(SUNMemoryHelper *helper*, SUNMemory *mem*)

Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

Arguments:

- *helper* – the `SUNMemoryHelper` object
- *mem* – the `SUNMemory` object

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Copy**(SUNMemoryHelper *helper*, SUNMemory *dst*, SUNMemory *src*,
size_t *mem_size*)

Synchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object should use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments:

- *helper* – the SUNMemoryHelper object
- *dst* – the destination memory to copy to
- *src* – the source memory to copy from
- *mem_size* – the number of bytes to copy

Returns:

An int flag indicating success (zero) or failure (non-zero).

14.1.2 Utility Functions

The SUNMemoryHelper API defines the following functions which do not require a SUNMemoryHelper instance:

SUNMemory **SUNMemoryHelper_Alias**(SUNMemory *mem1*)
Returns a SUNMemory object whose ptr field points to the same address as mem1. The new object *will not* have ownership of ptr, therefore, it will not free ptr when SUNMemoryHelper_Dealloc is called.

Arguments:

- *mem1* – a SUNMemory object

Returns:

A SUNMemory object.

SUNMemory **SUNMemoryHelper_Wrap**(void* *ptr*, SUNMemoryType *mem_type*)
Returns a SUNMemory object whose ptr field points to the ptr argument passed to the function. The new object *will not* have ownership of ptr, therefore, it will not free ptr when SUNMemoryHelper_Dealloc is called.

Arguments:

- *ptr* – the data pointer to wrap in a SUNMemory object
- *mem_type* – the SUNMemoryType of the ptr

Returns:

A SUNMemory object.

SUNMemoryHelper **SUNMemoryHelper_NewEmpty**()
Returns an empty SUNMemoryHelper. This is useful for building custom SUNMemoryHelper implementations.

Returns:

A SUNMemoryHelper object.

int **SUNMemoryHelper_CopyOps**(SUNMemoryHelper *src*, SUNMemoryHelper *dst*)
Copies the ops field of src to the ops field of dst. This is useful for building custom SUNMemoryHelper implementations.

Arguments:

- *src* – the object to copy from
- *dst* – the object to copy to

Returns:

An int flag indicating success (zero) or failure (non-zero).

14.1.3 Implementation overridable operations with defaults

In addition, the SUNMemoryHelper API defines the following *optionally overridable* operations which do require a SUNMemoryHelper instance:

int **SUNMemoryHelper_CopyAsync**(SUNMemoryHelper *helper*, SUNMemory *dst*, SUNMemory *src*, size_t *mem_size*, void* *ctx*)

Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object should use the memory types of `dst` and `src` to determine the appropriate transfer type necessary. The `ctx` argument is used when a different execution stream needs to be provided to perform the copy in, e.g. with `CUDA` this would be a `cudaStream_t`.

Arguments:

- *helper* – the `SUNMemoryHelper` object
- *dst* – the destination memory to copy to
- *src* – the source memory to copy from
- *mem_size* – the number of bytes to copy
- *ctx* – typically a handle for an object representing an alternate execution stream, but it can be any implementation specific data

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

Note: If this operation is not defined by the implementation, then `SUNMemoryHelper_Copy` will be used.

SUNMemoryHelper **SUNMemoryHelper_Clone**(SUNMemoryHelper *helper*)

Clones the `SUNMemoryHelper` object itself.

Arguments:

- *helper* – the `SUNMemoryHelper` object to clone

Returns:

A `SUNMemoryHelper` object.

Note: If this operation is not defined by the implementation, then the default clone will only copy the `SUNMemoryHelper_Ops` structure stored in `helper->ops`, and not the `helper->content` field.

---

int **SUNMemoryHelper_Destroy**(SUNMemoryHelper *helper*)

Destroys (frees) the `SUNMemoryHelper` object itself.

Arguments:

- *helper* – the `SUNMemoryHelper` object to destroy

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

Note: If this operation is not defined by the implementation, then the default destroy will only free the `helper->ops` field and the `helper` itself. The `helper->content` field will not be freed.

14.1.4 Implementing a custom SUNMemoryHelper

A particular implementation of the SUNMemoryHelper API must:

- Define and implement the required operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one SUNMemoryHelper module in the same code.
- Optionally, specify the *content* field of SUNMemoryHelper.
- Optionally, define and implement additional user-callable routines acting on the newly defined SUNMemoryHelper.

An example of a custom SUNMemoryHelper is given in examples/utilities/custom_memory_helper.h.

14.2 The SUNMemoryHelper_Cuda Implementation

The `SUNMemoryHelper_Cuda` module is an implementation of the `SUNMemoryHelper` API that interfaces to the NVIDIA *[CUDA]* library. The implementation defines the constructor

SUNMemoryHelper **`SUNMemoryHelper_Cuda`**()
: Allocates and returns a `SUNMemoryHelper` object for handling CUDA memory. A `SUNMemoryHelper` object if successful, or `NULL` if not.

14.2.1 SUNMemoryHelper API Functions

The implementation provides the following operations defined by the `SUNMemoryHelper` API:

SUNMemory **`SUNMemoryHelper_Alloc_Cuda`**(SUNMemoryHelper *helper*, SUNMemory *memptr*, size_t *mem_size*, SUNMemoryType *mem_type*)
: Allocates a `SUNMemory` object whose `ptr` field is allocated for `mem_size` bytes and is of type `mem_type`. The new object will have ownership of `ptr` and will be deallocated when `SUNMemoryHelper_Dealloc` is called.

  The `SUNMemoryType` supported are

  - `SUNMEMTYPE_HOST` – memory is allocated with a call to `malloc`
  - `SUNMEMTYPE_PINNED` – memory is allocated with a call to `cudaMallocHost`
  - `SUNMEMTYPE_DEVICE` – memory is allocated with a call to `cudaMalloc`
  - `SUNMEMTYPE_UVM` – memory is allocated with a call to `cudaMallocManaged`

  Arguments:

  - *helper* – the `SUNMemoryHelper` object
  - *memptr* – pointer to the allocated `SUNMemory`
  - *mem_size* – the size in bytes of the `ptr`
  - *mem_type* – the `SUNMemoryType` of the `ptr`

  Returns:

  An `int` flag indicating success (zero) or failure (non-zero).

int **`SUNMemoryHelper_Dealloc_Cuda`**(SUNMemoryHelper *helper*, SUNMemory *mem*)
: Deallocates the `mem->ptr` field if it is owned by `mem`, and then deallocates the `mem` object.

  Arguments:

- *helper* – the `SUNMemoryHelper` object
- *mem* – the `SUNMemory` object

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_Copy_Cuda**(SUNMemoryHelper *helper*, SUNMemory *dst*, SUNMemory *src*, size_t *mem_size*)

Synchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object should use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments:

- *helper* – the `SUNMemoryHelper` object
- *dst* – the destination memory to copy to
- *src* – the source memory to copy from
- *mem_size* – the number of bytes to copy

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

int **SUNMemoryHelper_CopyAsync**(SUNMemoryHelper *helper*, SUNMemory *dst*, SUNMemory *src*, size_t *mem_size*, void* *ctx*)

Asynchronously copies `mem_size` bytes from the the source memory to the destination memory. The copy can be across memory spaces, e.g. host to device, or within a memory space, e.g. host to host. The `helper` object should use the memory types of `dst` and `src` to determine the appropriate transfer type necessary.

Arguments:

- *helper* – the `SUNMemoryHelper` object
- *dst* – the destination memory to copy to
- *src* – the source memory to copy from
- *mem_size* – the number of bytes to copy
- *ctx* – the `cudaStream_t` handle for the stream that the copy will be performed on

Returns:

An `int` flag indicating success (zero) or failure (non-zero).

Chapter 15

ARKode Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form `SOLVER-X.Y.Z.tar.gz`, where `SOLVER` is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `X.Y.Z` represents the version number (of the SUNDIALS suite or of the individual solver). To begin the installation, first uncompress and expand the sources, by issuing

```
% tar -zxf SOLVER-X.Y.Z.tar.gz
```

This will extract source files under a directory `SOLVER-X.Y.Z`.

Starting with version 2.6.0 of SUNDIALS, CMake is the only supported method of installation. The explanations of the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

  `SOLVERDIR` is the directory `SOLVER-X.Y.Z` created above; i.e. the directory containing the SUNDIALS sources.

  `BUILDDIR` is the (temporary) directory under which SUNDIALS is built.

  `INSTDIR` is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory `INSTDIR/include` while libraries are installed under `INSTDIR/lib`, with `INSTDIR` specified at configuration time.

- For SUNDIALS' CMake-based installation, in-source builds are prohibited; in other words, the build directory `BUILDDIR` can **not** be the same as `SOLVERDIR` and such an attempt will lead to an error. This prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.
- The installation directory `INSTDIR` can not be the same as the source directory `SOLVERDIR`.
- By default, only the libraries and header files are exported to the installation directory `INSTDIR`. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.

Further details on the CMake-based installation procedures, instructions for manual compilation, and a roadmap of the resulting installed libraries and exported header files, are provided in the following subsections:

- *CMake-based installation*
- *Installed libraries and exported header files*

15.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version 3.0.2 or higher and a working C compiler. On Unix-like operating systems, it also requires Make (and `curses`, including its development libraries, for the GUI front end to CMake, `ccmake` or `cmake-gui`), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included may be out of date. Many new CMake features have been added recently, and you should download the latest version from http://www.cmake.org. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake` or `cmake-gui` (depending on the version of CMake), while Windows users will be able to use `CMakeSetup`.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a `make distclean` procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a `make clean` which will remove files generated by the compiler and linker.

15.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The INSTDIR defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command, or from a wxWidgets or QT based GUI by using the `cmake-gui` command. Examples for using both text and graphical methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
$ mkdir (...)/INSTDIR
$ mkdir (...)/BUILDDIR
$ cd (...)/BUILDDIR
```

15.1.1.1 Building with the GUI

Using CMake with the `ccmake` GUI follows the general process:

- Select and modify values, run configure (`c` key)
- New values are denoted with an asterisk

- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string
  - For file and directories, the <tab> key can be used to complete
- Repeat until all values are set as desired and the generate option is available (g key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (t key)
- To search for a variable press / key, and to repeat the search, press the n key

Using CMake with the cmake-gui GUI follows a similar process:

- Select and modify values, click Configure
- The first time you click Configure, make sure to pick the appropriate generator (the following will ssume generation of Unix Makfiles).
- New values are highlighted in red
- To set a variable, click on or move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will check/uncheck the box
  - If it is string or file, it will allow editing of the string. Additionally, an ellipsis button will appear ... on the far right of the entry. Clicking this button will bring up the file or directory selection dialog.
  - For files and directories, the <tab> key can be used to complete
- Repeat until all values are set as desired and click the Generate button
- Some variables (advanced variables) are not visible right away
- To see advanced variables, click the advanced button

To build the default configuration using the curses GUI, from the BUILDDIR enter the ccmake command and point to the SOLVERDIR:

```
$ ccmake (...)/SOLVERDIR
```

Similarly, to build the default configuration using the wxWidgets GUI, from the BUILDDIR enter the cmake-gui command and point to the SOLVERDIR:

```
$ cmake-gui (...)/SOLVERDIR
```

The default curses configuration screen is shown in the following figure.

The default INSTDIR for both SUNDIALS and corresponding examples can be changed by setting the CMAKE_INSTALL_PREFIX and the EXAMPLES_INSTALL_PATH as shown in the following figure.

Pressing the g key or clicking generate will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
$ make
```

or for a faster parallel build (e.g. using 4 threads), you can run

```
$ make -j 4
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

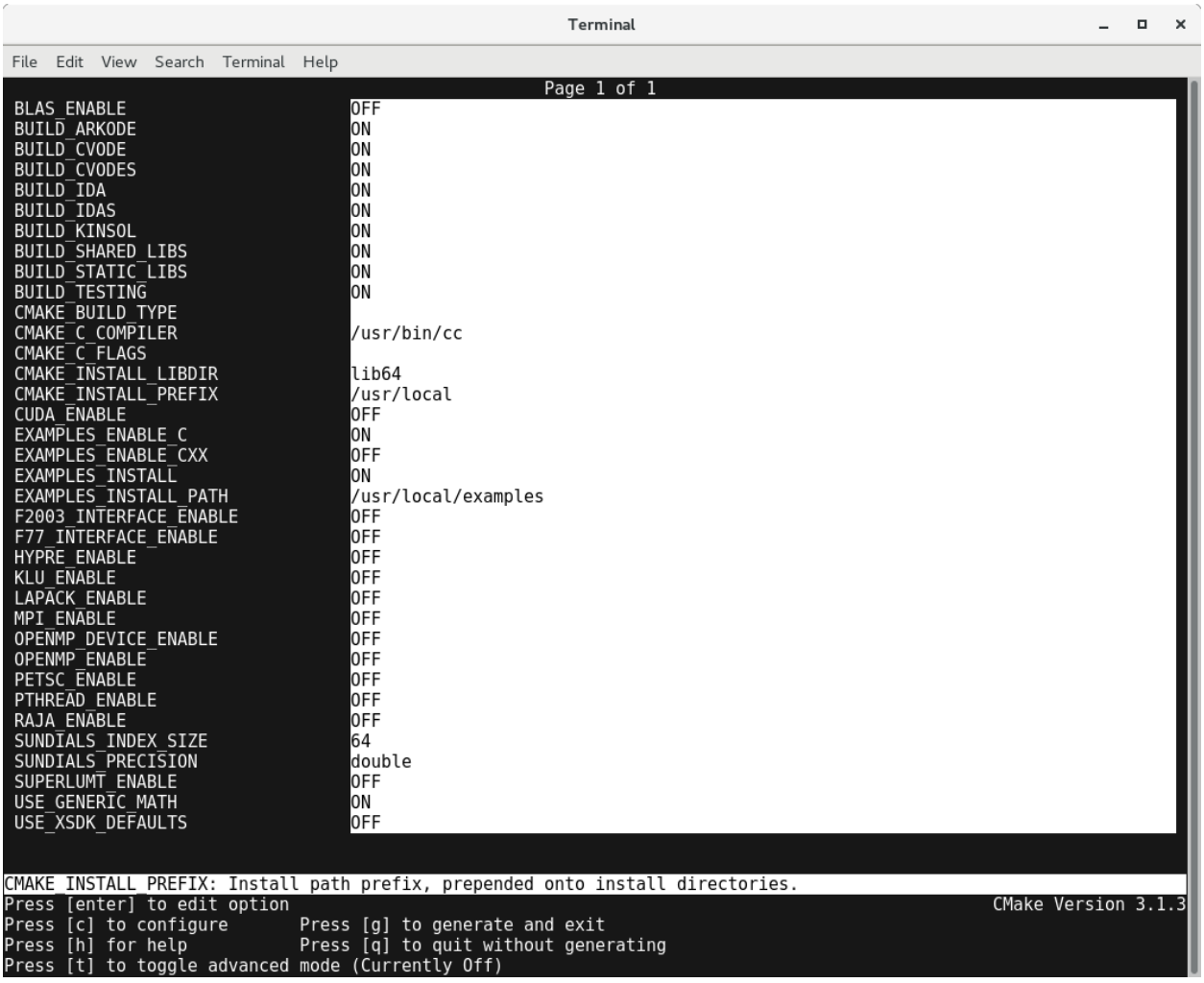


Fig. 15.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

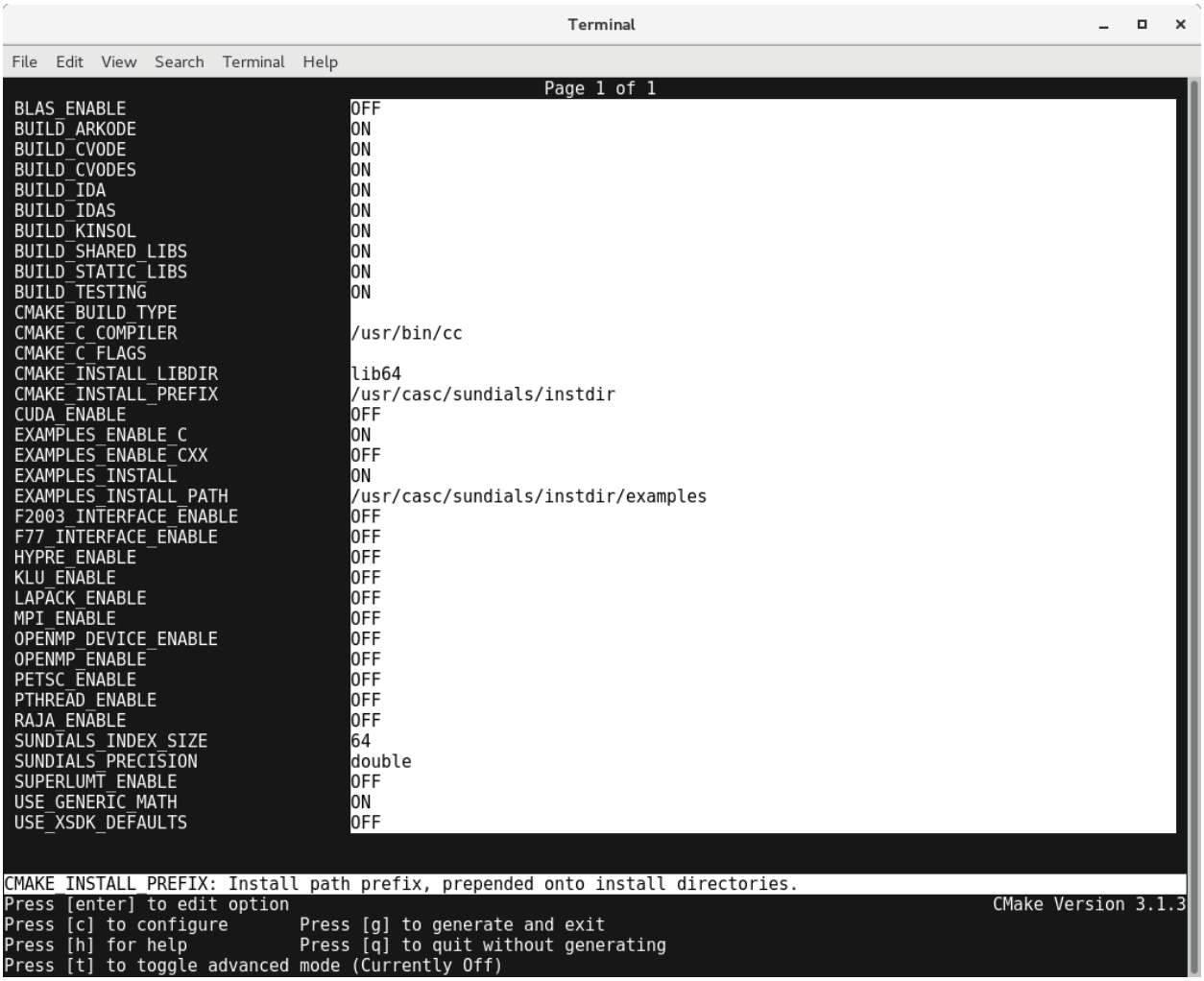


Fig. 15.2: Changing the INSTDIR for SUNDIALS and corresponding EXAMPLES.

```
$ make install
```

15.1.1.2 Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the `cmake` command. The following will build the default configuration:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
>  -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
>  ../srcdir
$ make
$ make install
```

15.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

BUILD_ARKODE Build the ARKODE library

Default: `ON`

BUILD_CVODE Build the CVODE library

Default: `ON`

BUILD_CVODES Build the CVODES library

Default: `ON`

BUILD_IDA Build the IDA library

Default: `ON`

BUILD_IDAS Build the IDAS library

Default: `ON`

BUILD_KINSOL Build the KINSOL library

Default: `ON`

BUILD_SHARED_LIBS Build shared libraries

Default: `ON`

BUILD_STATIC_LIBS Build static libraries

Default: `ON`

CMAKE_BUILD_TYPE Choose the type of build, options are: `None` (`CMAKE_C_FLAGS` used), `Debug`, `Release`, `RelWithDebInfo`, and `MinSizeRel`

Default:

Note: Specifying a build type will trigger the corresponding build type specific compiler flag options below which will be appended to the flags set by `CMAKE_<language>_FLAGS`.

CMAKE_C_COMPILER C compiler

Default: `/usr/bin/cc`

CMAKE_C_FLAGS Flags for C compiler

Default:

CMAKE_C_FLAGS_DEBUG Flags used by the C compiler during debug builds

Default: `-g`

CMAKE_C_FLAGS_MINSIZEREL Flags used by the C compiler during release minsize builds

Default: `-Os -DNDEBUG`

CMAKE_C_FLAGS_RELEASE Flags used by the C compiler during release builds

Default: `-O3 -DNDEBUG`

CMAKE_CXX_COMPILER C++ compiler

Default: `/usr/bin/c++`

Note: A C++ compiler (and all related options) are only are triggered if C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is ON). All SUNDIALS solvers can be used from C++ applications by default without setting any additional configuration options.

CMAKE_CXX_FLAGS Flags for C++ compiler

Default:

CMAKE_CXX_FLAGS_DEBUG Flags used by the C++ compiler during debug builds

Default: `-g`

CMAKE_CXX_FLAGS_MINSIZEREL Flags used by the C++ compiler during release minsize builds

Default: `-Os -DNDEBUG`

CMAKE_CXX_FLAGS_RELEASE Flags used by the C++ compiler during release builds

Default: `-O3 -DNDEBUG`

CMAKE_Fortran_COMPILER Fortran compiler

Default: `/usr/bin/gfortran`

Note: Fortran support (and all related options) are triggered only if either Fortran-C support is (`FCMIX_ENABLE` is ON) or LAPACK support is enabled (`ENABLE_LAPACK` is `ON`).

CMAKE_Fortran_FLAGS Flags for Fortran compiler

Default:

CMAKE_Fortran_FLAGS_DEBUG Flags used by the Fortran compiler during debug builds

Default: `-g`

CMAKE_Fortran_FLAGS_MINSIZEREL Flags used by the Fortran compiler during release minsize builds

Default: `-Os`

CMAKE_Fortran_FLAGS_RELEASE Flags used by the Fortran compiler during release builds

Default: `-O3`

CMAKE_INSTALL_PREFIX Install path prefix, prepended onto install directories

Default: `/usr/local`

Note:

The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories `include` and `lib` of

`CMAKE_INSTALL_PREFIX`, respectively.

ENABLE_CUDA Build the SUNDIALS CUDA modules.

Default: `OFF`

CMAKE_CUDA_ARCHITECTURES Specifies the CUDA architecture to compile for.

Default: `sm_30`

ENABLE_XBRAID Enable or disable the ARKStep + XBraid interface.

Default: `OFF`

Note: See additional information on building with *XBraid* enabled in *Working with external Libraries*.

EXAMPLES_ENABLE_C Build the SUNDIALS C examples

Default: `ON`

EXAMPLES_ENABLE_CXX Build the SUNDIALS C++ examples

Default: `OFF`

EXAMPLES_ENABLE_CUDA Build the SUNDIALS CUDA examples

Default: `OFF`

Note: You need to enable CUDA support to build these examples.

EXAMPLES_ENABLE_F77 Build the SUNDIALS Fortran77 examples

Default: `ON` (if `FCMIX_ENABLE` is `ON`)

EXAMPLES_ENABLE_F90 Build the SUNDIALS Fortran90 examples

Default: `ON` (if `BUILD_FORTRAN77_INTERFACE` is `ON`)

EXAMPLES_ENABLE_F2003 Build the SUNDIALS Fortran2003 examples

Default: `ON` (if `BUILD_FORTRAN_MODULE_INTERFACE` is `ON`)

EXAMPLES_INSTALL Install example files

Default: `ON`

Note: This option is triggered when any of the SUNDIALS example programs are enabled (`EXAMPLES_ENABLE_<language>` is `ON`). If the user requires installation of example programs then the

sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by `EXAMPLES_INSTALL_PATH`. A CMake configuration script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by `EXAMPLES_INSTALL_PATH`.

EXAMPLES_INSTALL_PATH Output directory for installing example files

Default: `/usr/local/examples`

Note: The actual default value for this option will be an `examples` subdirectory created under `CMAKE_INSTALL_PREFIX`.

BUILD_FORTRAN77_INTERFACE Enable Fortran77-C interface

Default: `OFF`

BUILD_FORTRAN_MODULE_INTERFACE Enable Fortran2003 interface

Default: `OFF`

ENABLE_HYPRE Flag to enable *hypre* support

Default: `OFF`

Note: See additional information on building with *hypre* enabled in *Working with external Libraries*.

HYPRE_INCLUDE_DIR Path to *hypre* header files

Default: none

HYPRE_LIBRARY Path to *hypre* installed library files

Default: none

ENABLE_KLU Enable KLU support

Default: `OFF`

Note: See additional information on building with KLU enabled in *Working with external Libraries*.

KLU_INCLUDE_DIR Path to SuiteSparse header files

Default: none

KLU_LIBRARY_DIR Path to SuiteSparse installed library files

Default: none

ENABLE_LAPACK Enable LAPACK support

Default: `OFF`

Note: Setting this option to `ON` will trigger additional CMake options. See additional information on building with LAPACK enabled in *Working with external Libraries*.

LAPACK_LIBRARIES LAPACK (and BLAS) libraries

Default: `/usr/lib/liblapack.so;/usr/lib/libblas.so`

Note: CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

ENABLE_MPI Enable MPI support. This will build the parallel nvector and the MPI-aware version of the ManyVector library.

Default: `OFF`

Note: Setting this option to `ON` will trigger several additional options related to MPI.

MPI_C_COMPILER `mpicc` program

Default:

MPI_CXX_COMPILER `mpicxx` program

Default:

Note: This option is triggered only if MPI is enabled (`ENABLE_MPI` is `ON`) and C++ examples are enabled (`EXAMPLES_ENABLE_CXX` is `ON`). All SUNDIALS solvers can be used from C++ MPI applications by default without setting any additional configuration options other than `ENABLE_MPI`.

MPI_Fortran_COMPILER `mpif77` or `mpif90` program

Default:

Note: This option is triggered only if MPI is enabled (`ENABLE_MPI` is `ON`) and Fortran-C support is enabled (`EXAMPLES_ENABLE_F77` or `EXAMPLES_ENABLE_F90` are `ON`).

MPIEXEC_EXECUTABLE Specify the executable for running MPI programs

Default: `mpirun`

Note: This option is triggered only if MPI is enabled (`ENABLE_MPI` is `ON`).

ENABLE_OPENMP Enable OpenMP support (build the OpenMP NVector)

Default: `OFF`

ENABLE_PETSC Enable PETSc support

Default: `OFF`

Note: See additional information on building with PETSc enabled in *Working with external Libraries*.

PETSC_DIR Path to PETSc installation

Default: none

PETSC_LIBRARIES (advanced option) Semi-colon separated list of PETSc link libraries. Unless provided by the user, this is autopopulated based on the PETSc installation found in `PETSC_DIR`.

Default: none

PETSC_INCLUDES (advanced option) Semi-colon separated list of PETSc include directroies. Unless provided by the user, this is autopopulated based on the PETSc installation found in `PETSC_DIR`.

Default: none

ENABLE_PTHREAD Enable Pthreads support (build the Pthreads NVector)

Default: `OFF`

ENABLE_RAJA Enable RAJA support.

Default: OFF

Note: You need to enable CUDA or HIP in order to build the RAJA vector module.

SUNDIALS_RAJA_BACKENDS If building SUNDIALS with RAJA support, this sets the RAJA backend to target. Values supported are CUDA and HIP.

Default: CUDA

ENABLE_SUPERLUDIST Enable SuperLU_DIST support

Default: `OFF`

Note: See additional information on building wtih SuperLU_DIST enabled in *Working with external Libraries*.

SUPERLUDIST_INCLUDE_DIR Path to SuperLU_DIST header files (under a typical SuperLU_DIST install, this is typically the SuperLU_DIST `SRC` directory)

Default: none

SUPERLUDIST_LIBRARY_DIR Path to SuperLU_DIST installed library files

Default: none

SUPERLUDIST_LIBRARIES Semi-colon separated list of libraries needed for SuperLU_DIST

Default: none

SUPERLUDIST_OpenMP Enable SUNDIALS support for SuperLU_DIST built with OpenMP

Default: none

Note: SuperLU_DIST must be built with OpenMP support for this option to function. Additionally the environment variable `OMP_NUM_THREADS` must be set to the desired number of threads.

ENABLE_SUPERLUMT Enable SuperLU_MT support

Default: `OFF`

Note: See additional information on building with SuperLU_MT enabled in *Working with external Libraries*.

SUPERLUMT_INCLUDE_DIR Path to SuperLU_MT header files (under a typical SuperLU_MT install, this is typically the SuperLU_MT `SRC` directory)

Default: none

SUPERLUMT_LIBRARY_DIR Path to SuperLU_MT installed library files

Default: none

SUPERLUMT_THREAD_TYPE Must be set to Pthread or OpenMP, depending on how SuperLU_MT was compiled.

Default: Pthread

SUNDIALS_BUILD_WITH_MONITORING Build SUNDIALS with capabilties for fine-grained monitoring of solver progress and statistics. This is primarily useful for debugging.

Default: OFF

Note: Building with monitoring may result in minor performance degradation even if monitoring is not utilized.

CMAKE_CXX_STANDARD The C++ standard to build C++ parts of SUNDIALS with.

Default: 11

Note: Options are 99, 11, 14, 17. This option only used when a C++ compiler is required.

SUNDIALS_F77_FUNC_CASE Specify the case to use in the Fortran name-mangling scheme, options are: `lower` or `upper`

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`lower`) scheme if one can not be determined. If used, `SUNDIALS_F77_FUNC_UNDERSCORES` must also be set.

SUNDIALS_F77_FUNC_UNDERSCORES Specify the number of underscores to append in the Fortran name-mangling scheme, options are: `none`, `one`, or `two`

Default:

Note: The build system will attempt to infer the Fortran name-mangling scheme using the Fortran compiler. This option should only be used if a Fortran compiler is not available or to override the inferred or default (`one`) scheme if one can not be determined. If used, `SUNDIALS_F77_FUNC_CASE` must also be set.

SUNDIALS_INDEX_TYPE (advanced) Integer type used for SUNDIALS indices. The size must match the size provided for the `SUNDIALS_INDEX_SIZE` option.

Default:

Note: In past SUNDIALS versions, a user could set this option to `INT64_T` to use 64-bit integers, or `INT32_T` to use 32-bit integers. Starting in SUNDIALS 3.2.0, these special values are deprecated. For SUNDIALS 3.2.0 and up, a user will only need to use the `SUNDIALS_INDEX_SIZE` option in most cases.

SUNDIALS_INDEX_SIZE Integer size (in bits) used for indices in SUNDIALS, options are: `32` or `64`

Default: `64`

Note: The build system tries to find an integer type of appropriate size. Candidate 64-bit integer types are (in order of preference): `int64_t`, `__int64`, `long long`, and `long`. Candidate 32-bit integers are (in order of preference): `int32_t`, `int`, and `long`. The advanced option, `SUNDIALS_INDEX_TYPE` can be used to provide a type not listed here.

SUNDIALS_PRECISION Precision used in SUNDIALS, options are: `double`, `single` or `extended`

Default: `double`

SUNDIALS_INSTALL_CMAKEDIR Installation directory for the SUNDIALS cmake files (relative to `CMAKE_INSTALL_PREFIX`).

Default: `CMAKE_INSTALL_PREFIX/cmake/sundials`

USE_GENERIC_MATH Use generic (`stdc`) math libraries

Default: `ON`

XBRAID_DIR The root directory of the XBraid installation.

Default: `OFF`

XBRAID_INCLUDES Semi-colon separated list of XBraid include directories. Unless provided by the user, this is autopopulated based on the XBraid installation found in `XBRAID_DIR`.

Default: none

XBRAID_LIBRARIES Semi-colon separated list of XBraid link libraries. Unless provided by the user, this is autopopulated based on the XBraid installation found in `XBRAID_DIR`.

Default: none

USE_XSDK_DEFAULTS Enable xSDK (see https://xsdk.info for more information) default configuration settings. This sets `CMAKE_BUILD_TYPE` to `Debug`, `SUNDIALS_INDEX_SIZE` to 32 and `SUNDIALS_PRECISION` to double.

Default: `OFF`

15.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.

To configure SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of `/home/myname/sundials/`, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_MPI=ON \
> -DFCMIX_ENABLE=ON \
> /home/myname/sundials/srcdir

% make install
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_MPI=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir

% make install
```

15.1.4 Working with external Libraries

The SUNDIALS suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries.

15.1.4.1 Building with LAPACK

To enable LAPACK, set the `ENABLE_LAPACK` option to `ON`. If the directory containing the LAPACK library is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `LAPACK_LIBRARIES` variable accordingly, otherwise CMake will attempt to find the LAPACK library in standard system locations. To explicitly tell CMake what library to use, the `LAPACK_LIBRARIES` variable can be set to the desired libraries required for LAPACK.

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DENABLE_LAPACK=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/libblas.so;/mylapackpath/lib/liblapack.so \
> /home/myname/sundials/srcdir

% make install
```

Note: If a working Fortran compiler is not available to infer the Fortran name-mangling scheme, the options `SUNDIALS_F77_FUNC_CASE` and `SUNDIALS_F77_FUNC_UNDERSCORES` *must* be set in order to bypass the check for a Fortran compiler and define the name-mangling scheme. The defaults for these options in earlier versions of SUNDIALS were `lower` and `one`, respectively.

15.1.4.2 Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: http://faculty.cse.tamu.edu/davis/suitesparse.html .

SUNDIALS has been tested with SuiteSparse version 5.7.2. To enable KLU, set `ENABLE_KLU` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`.

15.1.4.3 Building with SuperLU_DIST

The SuperLU_DIST libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/\protect\T1\textdollarsim\protect\T1\textdollarxiaoye/SuperLU/#superlu_dist.

SUNDIALS has been tested with SuperLU_DIST 6.1.1. To enable SuperLU_DIST, set `ENABLE_SUPERLUDIST` to `ON`, set `SUPERLUDIST_INCLUDE_DIR` to the `SRC` path of the SuperLU_DIST installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_DIST installation. At the same time, the variable `SUPERLUDIST_LIBRARIES` must be set to a semi-colon separated list of other libraries SuperLU_DIST depends on. For example, if SuperLU_DIST was built with LAPACK, then include the LAPACK library in this list. If SuperLU_DIST was built with OpenMP support, then you may set `SUPERLUDIST_OpenMP` to `ON` utilize the OpenMP functionality of SuperLU_DIST.

15.1.4.4 Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/\protect\T1\textdollarsim\protect\T1\textdollarxiaoye/SuperLU/#superlu_mt .

SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set `ENABLE_SUPERLUMT` to `ON`, set `SUPERLUMT_INCLUDE_DIR` to the `SRC` path of the SuperLU_MT installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_MT installation. At the same time, the variable `SUPERLUMT_LIBRARIES` must be set to a semi-colon separated list of other libraries SuperLU_MT depends on. For example, if SuperLU_MT was build with an external blas library, then include the full path to the blas library in this list. Additionally, the variable `SUPERLUMT_THREAD_TYPE` must be set to either `Pthread` or `OpenMP`.

Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either `ENABLE_OPENMP` or `ENABLE_PTHREAD` set to `ON` then SuperLU_MT should be set to use the same threading type.

15.1.4.5 Building with PETSc

The PETSc libraries are available for download from the Argonne National Laboratory website: http://www.mcs.anl.gov/petsc .

SUNDIALS has been tested with PETSc version 3.10.0 - 3.14.0. To enable PETSc, set `ENABLE_PETSC` to `ON`, and set `PETSC_DIR` to the path of the PETSc installation. Alternatively, a user can provide a list of inlcude paths in `PETSC_INCLUDES` and a list of complete paths to the PETSc libraries in `PETSC_LIBRARIES`.

15.1.4.6 Building with *hypre*

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: http://computing.llnl.gov/projects/hypre. SUNDIALS has been tested with *hypre* version 2.19.0. To enable *hypre*, set `ENABLE_HYPRE` to `ON`, set `HYPRE_INCLUDE_DIR` to the `include` path of the *hypre* installation, and set the variable `HYPRE_LIBRARY_DIR` to the `lib` path of the *hypre* installation.

Note: SUNDIALS must be configured so that `SUNDIALS_INDEX_SIZE` (or equivalently, `XSDK_INDEX_SIZE`) equals the precision of `HYPRE_BigInt` in the corresponding *hypre* installation.

15.1.4.7 Building with CUDA

SUNDIALS CUDA modules and examples have been tested with version 10 and 11 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from the NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set `ENABLE_CUDA` to `ON`. If CUDA is installed in a nonstandard location, you may be prompted to set the variable `CUDA_TOOLKIT_ROOT_DIR` with your CUDA Toolkit installation path. To enable CUDA examples, set `EXAMPLES_ENABLE_CUDA` to `ON`.

15.1.4.8 Building with RAJA

RAJA is a performance portability layer developed by Lawrence Livermore National Laboratory and can be obtained from https://github.com/LLNL/RAJA. SUNDIALS RAJA modules and examples have been tested with RAJA version 0.12.1. Building SUNDIALS RAJA modules requires a CUDA-enabled RAJA installation. To enable RAJA, set `ENABLE_CUDA` and `ENABLE_RAJA` to `ON`. If RAJA is installed in a nonstandard location you will be prompted to set the variable `RAJA_DIR` with the path to the RAJA CMake configuration file. To enable building the RAJA examples set `EXAMPLES_ENABLE_CUDA` to `ON`.

15.1.4.9 Building with XBraid

The XBraid library is available for download from the XBraid GitHub: https://github.com/XBraid/xbraid. SUNDIALS has been tested with XBraid version 3.0.0. To enable XBraid, set `ENABLE_XBRAID` to `ON`, set `XBRAID_DIR` to the root install location of XBraid or the location of the clone of the XBraid repository.

Note: At this time the XBraid types `braid_Int` and `braid_Real` are hard-coded to `int` and `double` respectively. As such SUNDIALS must be configured with `SUNDIALS_INDEX_SIZE` set to `32` and `SUNDIALS_PRECISION` set to `double`. Additionally, SUNDIALS must be configured with `ENABLE_MPI` set to `ON`.

15.1.5 Testing the build and installation

If SUNDIALS was configured with `EXAMPLES_ENABLE_<language>` options to `ON`, then a set of regression tests can be run after building with the `make` command by running:

```
% make test
```

Additionally, if `EXAMPLES_INSTALL` was also set to `ON`, then a set of smoke tests can be run after installing with the `make install` command by running:

```
% make test_install
```

15.1.6 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set at least of the `EXAMPLES_ENABLE_<language>` options to `ON`, and set `EXAMPLES_INSTALL` to `ON`. Specify the installation path for the examples with the variable `EXAMPLES_INSTALL_PATH`. CMake will generate `CMakeLists.txt` configuration files (and `Makefile` files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional `Makefile` may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied `Makefile` simply run `make` to compile and generate the executables. To use CMake from within the installed example directory, run `cmake` (or `ccmake` or `cmake-gui` to use the GUI) followed by `make` to compile the example code. Note that if CMake is used, it will overwrite the traditional `Makefile` with a new CMake-generated `Makefile`.

The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.

NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

15.1.7 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the `SOLVERDIR`
2. Create a separate `BUILDDIR`
3. Open a Visual Studio Command Prompt and cd to `BUILDDIR`
4. Run `cmake-gui ../SOLVERDIR`

(a) Hit Configure

(b) Check/Uncheck solvers to be built

(c) Change `CMAKE_INSTALL_PREFIX` to `INSTDIR`

(d) Set other options as desired

(e) Hit Generate

5. Back in the VS Command Window:

(a) Run `msbuild ALL_BUILD.vcxproj`

(b) Run `msbuild INSTALL.vcxproj`

The resulting libraries will be in the `INSTDIR`.

The SUNDIALS project can also now be opened in Visual Studio. Double click on the `ALL_BUILD.vcxproj` file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

15.2 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
$ make install
```

will install the libraries under `LIBDIR` and the public header files under `INCLUDEDIR`. The values for these directories are `INSTDIR/lib` and `INSTDIR/include`, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under `LIBDIR/lib`, the public header files are further organized into subdirectories under `INCLUDEDIR/include`.

The installed libraries and exported header files are listed for reference in the *Table: SUNDIALS libraries and header files*. The file extension `.LIB` is typically `.so` for shared libraries and `.a` for static libraries. Note that, in this table names are relative to `LIBDIR` for libraries and to `INCLUDEDIR` for header files.

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the `INCLUDEDIR/include/sundials` directory since they are explicitly included by the appropriate solver header files (e.g., `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so, and would be useful, for example, if the functions declared in `sundials_dense.h` are to be used in building a preconditioner.

15.2.1 Using SUNDIALS as a Third Party Library in other CMake Projects

The `make install` command will also install a CMake package configuration file that other CMake projects can load to get all the information needed to build against SUNDIALS. In the consuming project's CMake code, the `find_package` command may be used to search for the configuration file, which will be installed to `instdir/SUNDIALS_INSTALL_CMAKEDIR/SUNDIALSConfig.cmake` alongside a package version file `instdir/SUNDIALS_INSTALL_CMAKEDIR/SUNDIALSConfigVersion.cmake`. Together these files contain all the information the consuming project needs to use SUNDIALS, including exported CMake targets. The SUNDIALS exported CMake targets follow the same naming convention as the generated library binaries, e.g. the exported target for CVODE is `SUNDIALS::cvode`. The CMake code snipped below shows how a consuming project might leverage the SUNDIALS package configuration file to build against SUNDIALS in their own CMake project.

```
project(MyProject)

# Set the variable SUNDIALS_DIR to the SUNDIALS instdir.
# When using the cmake CLI command, this can be done like so:
#   cmake -D SUNDIALS_DIR=/path/to/sundials/installation

find_project(SUNDIALS REQUIRED)

add_executable(myexec main.c)

# Link to SUNDIALS libraries through the exported targets.
# This is just an example, users should link to the targets appropriate
# for their use case.
target_link_libraries(myexec PUBLIC SUNDIALS::cvode SUNDIALS::nvecpetsc)
```

Table 15.1: SUNDIALS shared libraries and header files

Shared	Headers	sundials/sundials_band.h
		sundials/sundials_config.h
		sundials/sundials_cuda_policies.hpp
		sundials/sundials_dense.h
		sundials/sundials_direct.h
		sundials/sundials_fconfig.h
		sundials/sundials_fnvector.h
		sundials/sundials_iterative.h
		sundials/sundials_linearsolver.h
		sundials/sundials_nonlinearsolver.h
		sundials/sundials_matrix.h
		sundials/sundials_math.h
		sundials/sundials_nvector.h
		sundials/sundials_types.h
		sundials/sundials_version.h
		sundials/sundials_xbraid.h
NVECTOR Modules		
SERIAL	Libraries	libsundials_nvecserial.LIB
		libsundials_fnvecserial.a
	Headers	nvector/nvector_serial.h
PARALLEL	Libraries	libsundials_nvecparallel.LIB
		libsundials_fnvecparallel.a
	Headers	nvector/nvector_parallel.h
OPENMP	Libraries	libsundials_nvecopenmp.LIB
		libsundials_fnvecopenmp.a
	Headers	nvector/nvector_openmp.h
PTHREADS	Libraries	libsundials_nvecpthreads.LIB
		libsundials_fnvecpthreads.a
	Headers	nvector/nvector_pthreads.h
PARHYP	Libraries	libsundials_nvecparhyp.LIB
	Headers	nvector/nvector_parhyp.h
PETSC	Libraries	libsundials_nvecpetsc.LIB
	Headers	nvector/nvector_petsc.h
CUDA	Libraries	libsundials_nveccuda.LIB
	Headers	nvector/nvector_cuda.h
RAJA	Libraries	libsundials_nveccudaraja.LIB libsundials_nvechipraja.LIB

Continued on next page

Table 15.1 – continued from previous page

	Headers	`nvector/nvector_raja.h`
MANYVECTOR	Libraries	`libsundials_nvecmanyvector.LIB`
	Headers	`nvector/nvector_manyvector.h`
MPIMANYVECTOR	Libraries	`libsundials_nvecmpimanyvector.LIB`
	Headers	`nvector/nvector_mpimanyvector.h`
MPIPLUSX	Libraries	`libsundials_nvecmpiplusx.LIB`
	Headers	`nvector/nvector_mpiplusx.h`
SUNMATRIX Modules		
BAND	Libraries	`libsundials_sunmatrixband.LIB`
		`libsundials_fsunmatrixband.a`
	Headers	`sunmatrix/sunmatrix_band.h`
DENSE	Libraries	`libsundials_sunmatrixdense.LIB`
		`libsundials_fsunmatrixdense.a`
	Headers	`sunmatrix/sunmatrix_dense.h`
SPARSE	Libraries	`libsundials_sunmatrixsparse.LIB`
		`libsundials_fsunmatrixsparse.a`
	Headers	`sunmatrix/sunmatrix_sparse.h`
SLUNRLOC	Libraries	`libsundials_sunmatrixslunrloc.LIB`
	Headers	`sunmatrix/sunmatrix_slunrloc.h`
CUSPARSE	Libraries	`libsundials_sunmatrixcusparse.LIB`
	Headers	`sunmatrix/sunmatrix_cusparse.h`
SUNLINSOL Modules		
BAND	Libraries	`libsundials_sunlinsolband.LIB`
		`libsundials_fsunlinsolband.a`
	Headers	`sunlinsol/sunlinsol_band.h`
DENSE	Libraries	`libsundials_sunlinsoldense.LIB`
		`libsundials_fsunlinsoldense.a`
	Headers	`sunlinsol/sunlinsol_dense.h`
KLU	Libraries	`libsundials_sunlinsolklu.LIB`
		`libsundials_fsunlinsolklu.a`
	Headers	`sunlinsol/sunlinsol_klu.h`
LAPACKBAND	Libraries	`libsundials_sunlinsollapackband.LIB`
		`libsundials_fsunlinsollapackband.a`
	Headers	`sunlinsol/sunlinsol_lapackband.h`
LAPACKDENSE	Libraries	`libsundials_sunlinsollapackdense.LIB`
		`libsundials_fsunlinsollapackdense.a`
	Headers	`sunlinsol/sunlinsol_lapackdense.h`
PCG	Libraries	`libsundials_sunlinsolpcg.LIB`
		`libsundials_fsunlinsolpcg.a`
	Headers	`sunlinsol/sunlinsol_pcg.h`
SPBCGS	Libraries	`libsundials_sunlinsolspbcgs.LIB`
		`libsundials_fsunlinsolspbcgs.a`
	Headers	`sunlinsol/sunlinsol_spbcgs.h`
SPFGMR	Libraries	`libsundials_sunlinsolspfgmr.LIB`
		`libsundials_fsunlinsolspfgmr.a`
	Headers	`sunlinsol/sunlinsol_spfgmr.h`
SPGMR	Libraries	`libsundials_sunlinsolspgmr.LIB`
		`libsundials_fsunlinsolspgmr.a`
	Headers	`sunlinsol/sunlinsol_spgmr.h`
SPTFQMR	Libraries	`libsundials_sunlinsolsptfqmr.LIB`
		`libsundials_fsunlinsolsptfqmr.a`

Continued on next page

Table 15.1 – continued from previous page

	Headers	sunlinsol/sunlinsol_sptfqmr.h
SUPERLUMT	Libraries	libsundials_sunlinsolsuperlumt.LIB
		libsundials_fsunlinsolsuperlumt.a
	Headers	sunlinsol/sunlinsol_superlumt.h
SUPERLUDIST	Libraries	libsundials_sunlinsolsuperludist.LIB
	Headers	sunlinsol/sunlinsol_superludist.h
CUSOLVERSP_BATCHQR	Libraries	libsundials_sunlinsolcusolversp.LIB
	Headers	sunlinsol/sunlinsol_cusolversp_batchqr.h
SUNNONLINSOL Modules		
NEWTON	Libraries	libsundials_sunnonlinsolnewton.LIB
		libsundials_fsunnonlinsolnewton.a
	Headers	sunnonlinsol/sunnonlinsol_newton.h
FIXEDPOINT	Libraries	libsundials_sunnonlinsolfixedpoint.LIB
		libsundials_fsunnonlinsolfixedpoint.a
	Headers	sunnonlinsol/sunnonlinsol_fixedpoint.h
PETSCSNES	Libraries	libsundials_sunnonlinsolpetscsnes.LIB
	Headers	sunnonlinsol/sunnonlinsol_petscsnes.h
SUNDIALS Packages		
CVODE	Libraries	libsundials_cvode.LIB
		libsundials_fcvode.a
	Headers	cvode/cvode.h
		cvode/cvode_bandpre.h
		cvode/cvode_bbdpre.h
		cvode/cvode_diag.h
		cvode/cvode_direct.h
		cvode/cvode_impl.h
		cvode/cvode_ls.h
		cvode/cvode_spils.h
CVODES	Libraries	libsundials_cvodes.LIB
	Headers	cvodes/cvodes.h
		cvodes/cvodes_bandpre.h
		cvodes/cvodes_bbdpre.h
		cvodes/cvodes_diag.h
		cvodes/cvodes_direct.h
		cvodes/cvodes_impl.h
		cvodes/cvodes_spils.h
ARKODE	Libraries	libsundials_arkode.LIB
		libsundials_farkode.a
		libsundials_xbraid.LIB
	Headers	arkode/arkode.h
		arkode/arkode_arkstep.h
		arkode/arkode_bandpre.h
		arkode/arkode_bbdpre.h
		arkode/arkode_butcher.h
		arkode/arkode_butcher_dirk.h
		arkode/arkode_butcher_erk.h
		arkode/arkode_erkstep.h
		arkode/arkode_impl.h
		arkode/arkode_ls.h
		arkode/arkode_xbraid.h
IDA	Libraries	libsundials_ida.LIB
		libsundials_fida.a

Continued on next page

Table 15.1 – continued from previous page

	Headers	`ida/ida.h`
		`ida/ida_bbdpre.h`
		`ida/ida_direct.h`
		`ida/ida_impl.h`
		`ida/ida_ls.h`
		`ida/ida_spils.h`
IDAS	Libraries	`libsundials_idas.LIB`
	Headers	`idas/idas.h`
		`idas/idas_bbdpre.h`
		`idas/idas_direct.h`
		`idas/idas_impl.h`
		`idas/idas_spils.h`
KINSOL	Libraries	`libsundials_kinsol.LIB`
		`libsundials_fkinsol.a`
	Headers	`kinsol/kinsol.h`
		`kinsol/kinsol_bbdpre.h`
		`kinsol/kinsol_direct.h`
		`kinsol/kinsol_impl.h`
		`kinsol/kinsol_ls.h`
		`kinsol/kinsol_spils.h`

Chapter 16

Appendix: ARKode Constants

Below we list all input and output constants used by the main solver, timestepper, and linear solver modules, together with their numerical values and a short description of their meaning.

16.1 ARKode input constants

16.1.1 Shared ARKode input constants

ARK_NORMAL (1): Solver returns at a specified output time.

ARK_ONE_STEP (2): Solver returns after each successful step.

16.1.2 Interpolation module input constants

ARK_INTERP_MAX_DEGREE (5): Maximum possible interpolating polynomial degree.

ARK_INTERP_HERMITE (0): Specifies use of the Hermite polynomial interpolation module (for non-stiff problems)

ARK_INTERP_LAGRANGE (1): Specifies use of the Lagrange polynomial interpolation module (for stiff problems)

16.1.3 Explicit Butcher table specification

HEUN_EULER_2_1_2 (0): Use the Heun-Euler-2-1-2 ERK method

BOGACKI_SHAMPINE_4_2_3 (1): Use the Bogacki-Shampine-4-2-3 ERK method

ARK324L2SA_ERK_4_2_3 (2): Use the ARK-4-2-3 ERK method

ZONNEVELD_5_3_4 (3): Use the Zonneveld-5-3-4 ERK method

ARK436L2SA_ERK_6_3_4 (4): Use the ARK-6-3-4 ERK method

SAYFY_ABURUB_6_3_4 (5): Use the Sayfy-Aburub-6-3-4 ERK method

CASH_KARP_6_4_5 (6): Use the Cash-Karp-6-4-5 ERK method

FEHLBERG_6_4_5 (7): Use the Fehlberg-6-4-5 ERK method

DORMAND_PRINCE_7_4_5 (8): Use the Dormand-Prince-7-4-5 ERK method

ARK548L2SA_ERK_8_4_5 (9): Use the ARK-8-4-5 ERK method

VERNER_8_5_6 (10): Use the Verner-8-5-6 ERK method

FEHLBERG_13_7_8 (11): Use the Fehlberg-13-7-8 ERK method

KNOTH_WOLKE_3_3 (12): Use the Knoth-Wolke-3-3 ERK method

DEFAULT_ERK_2 (HEUN_EULER_2_1_2): Use the default second-order ERK method

DEFAULT_ERK_3 (BOGACKI_SHAMPINE_4_2_3): Use the default third-order ERK method

DEFAULT_ERK_4 (ZONNEVELD_5_3_4): Use the default fourth-order ERK method

DEFAULT_ERK_5 (CASH_KARP_6_4_5): Use the default fifth-order ERK method

DEFAULT_ERK_6 (VERNER_8_5_6): Use the default sixth-order ERK method

DEFAULT_ERK_8 (FEHLBERG_13_7_8): Use the default eighth-order ERK method

16.1.4 Implicit Butcher table specification

SDIRK_2_1_2 (100): Use the SDIRK-2-1-2 SDIRK method

BILLINGTON_3_3_2 (101): Use the Billington-3-3-2 SDIRK method

TRBDF2_3_3_2 (102): Use the TRBDF2-3-3-2 ESDIRK method

KVAERNO_4_2_3 (103): Use the Kvaerno-4-2-3 ESDIRK method

ARK324L2SA_DIRK_4_2_3 (104): Use the ARK-4-2-3 ESDIRK method

CASH_5_2_4 (105): Use the Cash-5-2-4 SDIRK method

CASH_5_3_4 (106): Use the Cash-5-3-4 SDIRK method

SDIRK_5_3_4 (107): Use the SDIRK-5-3-4 SDIRK method

KVAERNO_5_3_4 (108): Use the Kvaerno-5-3-4 ESDIRK method

ARK436L2SA_DIRK_6_3_4 (109): Use the ARK-6-3-4 ESDIRK method

KVAERNO_7_4_5 (110): Use the Kvaerno-7-4-5 ESDIRK method

ARK548L2SA_DIRK_8_4_5 (111): Use the ARK-8-4-5 ESDIRK method

ARK437L2SA_DIRK_7_3_4 (112): Use the ARK-7-3-4 ESDIRK method

ARK548L2SAb_DIRK_8_4_5 (113): Use the ARK-8-4-5b ESDIRK method

DEFAULT_DIRK_2 (SDIRK_2_1_2): Use the default second-order DIRK method

DEFAULT_DIRK_3 (ARK324L2SA_DIRK_4_2_3): Use the default third-order DIRK method

DEFAULT_DIRK_4 (SDIRK_5_3_4): Use the default fourth-order DIRK method

DEFAULT_DIRK_5 (ARK548L2SA_DIRK_8_4_5): Use the default fifth-order DIRK method

16.1.5 ImEx Butcher table specification

ARK324L2SA_ERK_4_2_3 and ARK324L2SA_DIRK_4_2_3 (2 and 16): Use the ARK-4-2-3 ARK method

ARK436L2SA_ERK_6_3_4 and ARK436L2SA_DIRK_6_3_4 (4 and 21): Use the ARK-6-3-4 ARK method

ARK548L2SA_ERK_8_4_5 and ARK548L2SA_DIRK_8_4_5 (9 and 23): Use the ARK-8-4-5 ARK method

DEFAULT_ARK_ETABLE_3 and DEFAULT_ARK_ITABLE_3 (ARK324L2SA_[ERK,DIRK]_4_2_3): Use the default third-order ARK method

DEFAULT_ARK_ETABLE_4 and DEFAULT_ARK_ITABLE_4 (ARK436L2SA_[ERK,DIRK]_6_3_4): Use the default fourth-order ARK method

DEFAULT_ARK_ETABLE_5 and DEFAULT_ARK_ITABLE_5 (ARK548L2SA_[ERK,DIRK]_8_4_5): Use the default fifth-order ARK method

16.2 ARKode output constants

16.2.1 Shared ARKode output constants

ARK_SUCCESS (0): Successful function return.

ARK_TSTOP_RETURN (1): ARKode succeeded by reaching the specified stopping point.

ARK_ROOT_RETURN (2): ARKode succeeded and found one more more roots.

ARK_WARNING (99): ARKode succeeded but an unusual situation occurred.

ARK_TOO_MUCH_WORK (-1): The solver took `mxstep` internal steps but could not reach `tout`.

ARK_TOO_MUCH_ACC (-2): The solver could not satisfy the accuracy demanded by the user for some internal step.

ARK_ERR_FAILURE (-3): Error test failures occurred too many times during one internal time step, or the minimum step size was reached.

ARK_CONV_FAILURE (-4): Convergence test failures occurred too many times during one internal time step, or the minimum step size was reached.

ARK_LINIT_FAIL (-5): The linear solver's initialization function failed.

ARK_LSETUP_FAIL (-6): The linear solver's setup function failed in an unrecoverable manner.

ARK_LSOLVE_FAIL (-7): The linear solver's solve function failed in an unrecoverable manner.

ARK_RHSFUNC_FAIL (-8): The right-hand side function failed in an unrecoverable manner.

ARK_FIRST_RHSFUNC_ERR (-9): The right-hand side function failed at the first call.

ARK_REPTD_RHSFUNC_ERR (-10): The right-hand side function had repeated recoverable errors.

ARK_UNREC_RHSFUNC_ERR (-11): The right-hand side function had a recoverable error, but no recovery is possible.

ARK_RTFUNC_FAIL (-12): The rootfinding function failed in an unrecoverable manner.

ARK_LFREE_FAIL (-13): The linear solver's memory deallocation function failed.

ARK_MASSINIT_FAIL (-14): The mass matrix linear solver's initialization function failed.

ARK_MASSSETUP_FAIL (-15): The mass matrix linear solver's setup function failed in an unrecoverable manner.

ARK_MASSSOLVE_FAIL (-16): The mass matrix linear solver's solve function failed in an unrecoverable manner.

ARK_MASSFREE_FAIL (-17): The mass matrix linear solver's memory deallocation function failed.

ARK_MASSMULT_FAIL (-18): The mass matrix-vector product function failed.

ARK_CONSTR_FAIL (-19): The inequality constraint test failed repeatedly or failed with the minimum step size.

ARK_MEM_FAIL (-20): A memory allocation failed.

ARK_MEM_NULL (-21): The `arkode_mem` argument was `NULL`.

ARK_ILL_INPUT (-22): One of the function inputs is illegal.

ARK_NO_MALLOC (-23): The ARKode memory block was not allocated by a call to `ARKodeMalloc()`.

ARK_BAD_K (-24): The derivative order k is larger than allowed.

ARK_BAD_T (-25): The time t is outside the last step taken.

ARK_BAD_DKY (-26): The output derivative vector is `NULL`.

ARK_TOO_CLOSE (-27): The output and initial times are too close to each other.

ARK_VECTOROP_ERR (-28): An error occurred when calling an NVECTOR routine.

ARK_NLS_INIT_FAIL (-29): An error occurred when initializing a SUNNonlinearSolver module.

ARK_NLS_SETUP_FAIL (-30): A non-recoverable error occurred when setting up a SUNNonlinearSolver module.

ARK_NLS_SETUP_RECVR (-31): A recoverable error occurred when setting up a SUNNonlinearSolver module.

ARK_NLS_OP_ERR (-32): An error occurred when calling a set/get routine in a SUNNonlinearSolver module.

ARK_INNERSTEP_ATTACH_ERR (-33): An error occurred when attaching the inner stepper module.

ARK_INNERSTEP_FAIL (-34): An error occurred in the inner stepper module.

ARK_PREINNERFN_FAIL (-35): An error occurred in the MRIStep pre inner integrator function.

ARK_POSTINNERFN_FAIL (-36): An error occurred in the MRIStep post inner integrator function.

ARK_INTERP_FAIL (-40): An error occurred in the ARKode polynomial interpolation module.

ARK_INVALID_TABLE (-41): An invalid Butcher or MRI table was encountered.

ARK_UNRECOGNIZED_ERROR (-99): An unknown error was encountered.

16.2.2 ARKLS linear solver modules

ARKLS_SUCCESS (0): Successful function return.

ARKLS_MEM_NULL (-1): The `arkode_mem` argument was `NULL`.

ARKLS_LMEM_NULL (-2): The ARKLS linear solver interface has not been initialized.

ARKLS_ILL_INPUT (-3): The ARKLS solver interface is not compatible with the current NVECTOR module, or an input value was illegal.

ARKLS_MEM_FAIL (-4): A memory allocation request failed.

ARKLS_PMEM_NULL (-5): The preconditioner module has not been initialized.

ARKLS_MASSMEM_NULL (-6): The ARKLS mass-matrix linear solver interface has not been initialized.

ARKLS_JACFUNC_UNRECVR (-7): The Jacobian function failed in an unrecoverable manner.

ARKLS_JACFUNC_RECVR (-8): The Jacobian function had a recoverable error.

ARKLS_MASSFUNC_UNRECVR (-9): The mass matrix function failed in an unrecoverable manner.

ARKLS_MASSFUNC_RECVR (-10): The mass matrix function had a recoverable error.

ARKLS_SUNMAT_FAIL (-11): An error occurred with the current SUNMATRIX module.

ARKLS_SUNLS_FAIL (-12): An error occurred with the current SUNLINSOL module.

Chapter 17

Appendix: Butcher tables

Here we catalog the full set of Butcher tables included in ARKode. We group these into three categories: *explicit*, *implicit* and *additive*. However, since the methods that comprise an additive Runge Kutta method are themselves explicit and implicit, their component Butcher tables are listed within their separate sections, but are referenced together in the additive section.

In each of the following tables, we use the following notation (shown for a 3-stage method):

$$\begin{array}{c|ccc} c_1 & a_{1,1} & a_{1,2} & a_{1,3} \\ c_2 & a_{2,1} & a_{2,2} & a_{2,3} \\ c_3 & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline q & b_1 & b_2 & b_3 \\ p & \tilde{b}_1 & \tilde{b}_2 & \tilde{b}_3 \end{array}$$

where here the method and embedding share stage A and c values, but use their stages z_i differently through the coefficients b and $\tilde{b}$ to generate methods of orders q (the main method) and p (the embedding, typically $q = p + 1$, though sometimes this is reversed).

Method authors often use different naming conventions to categorize their methods. For each of the methods below with an embedding, we follow the uniform naming convention:

```
NAME-S-P-Q
```

where here

- `NAME` is the author or the name provided by the author (if applicable),
- `S` is the number of stages in the method,
- `P` is the global order of accuracy for the embedding,
- `Q` is the global order of accuracy for the method.

For methods without an embedding (e.g., fixed-step methods) `P` is omitted so that methods follow the naming convention `NAME-S-Q`.

In the code, unique integer IDs are defined inside `arkode_butcher_erk.h` and `arkode_butcher_dirk.h` for each method, which may be used by calling routines to specify the desired method. These names are specified in `fixed width font` at the start of each method's section below.

Additionally, for each method we provide a plot of the linear stability region in the complex plane. These have been computed via the following approach. For any Runge Kutta method as defined above, we may define the stability function

$$R(\eta) = 1 + \eta b[I - \eta A]^{-1} e,$$

where $e \in \mathbb{R}^s$ is a column vector of all ones, $\eta = h\lambda$ and h is the time step size. If the stability function satisfies $|R(\eta)| \leq 1$ for all eigenvalues, λ, of $\frac{\partial}{\partial y} f(t, y)$ for a given IVP, then the method will be linearly stable for that problem and step size. The stability region

$$S = \{\eta \in \mathbb{C} \ : \ |R(\eta)| \leq 1\}$$

is typically given by an enclosed region of the complex plane, so it is standard to search for the border of that region in order to understand the method. Since all complex numbers with unit magnitude may be written as $e^{i\theta}$ for some value of θ, we perform the following algorithm to trace out this boundary.

1. Define an array of values `Theta`. Since we wish for a smooth curve, and since we wish to trace out the entire boundary, we choose 10,000 linearly-spaced points from 0 to 16π. Since some angles will correspond to multiple locations on the stability boundary, by going beyond 2π we ensure that all boundary locations are plotted, and by using such a fine discretization the Newton method (next step) is more likely to converge to the root closest to the previous boundary point, ensuring a smooth plot.

2. For each value $\theta \in$ `Theta`, we solve the nonlinear equation

   $$0 = f(\eta) = R(\eta) - e^{i\theta}$$

   using a finite-difference Newton iteration, using tolerance 10^{-7}, and differencing parameter $\sqrt{\varepsilon}$ ($\approx 10^{-8}$).

   In this iteration, we use as initial guess the solution from the previous value of θ, starting with an initial-initial guess of $\eta = 0$ for $\theta = 0$.

3. We then plot the resulting η values that trace the stability region boundary.

We note that for any stable IVP method, the value $\eta_0 = -\varepsilon + 0i$ is always within the stability region. So in each of the following pictures, the interior of the stability region is the connected region that includes η_0. Resultingly, methods whose linear stability boundary is located entirely in the right half-plane indicate an *A-stable* method.

17.1 Explicit Butcher tables

In the category of explicit Runge-Kutta methods, ARKode includes methods that have orders 2 through 6, with embeddings that are of orders 1 through 5.

17.1.1 Heun-Euler-2-1-2

Accessible via the constant `HEUN_EULER_2_1_2` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()*. This is the default 2nd order explicit method.

0	0	0
1	1	0
2	$\frac{1}{2}$	$\frac{1}{2}$
1	1	0

17.1.2 Bogacki-Shampine-4-2-3

Accessible via the constant `BOGACKI_SHAMPINE_4_2_3` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()*. This is the default 3rd order ex-

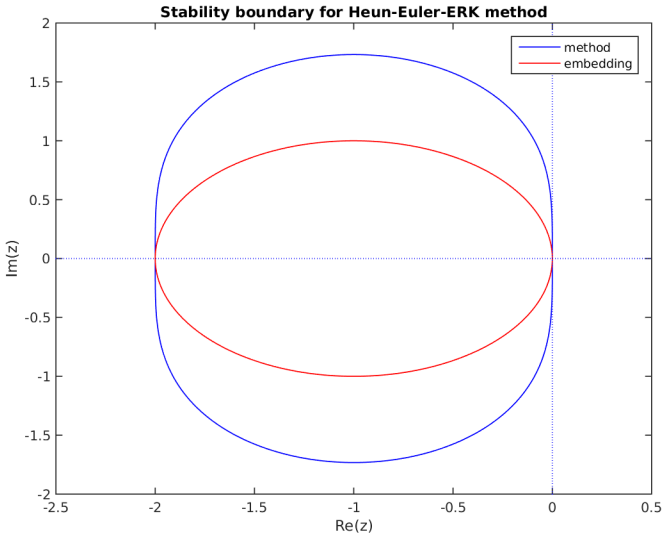


Fig. 17.1: Linear stability region for the Heun-Euler method. The method's region is outlined in blue; the embedding's region is in red.

plicit method (from *[BS1989]*).

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{3}{4} & 0 & \frac{3}{4} & 0 & 0 \\ 1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ \hline 3 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\ 2 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8} \end{array}$$

17.1.3 ARK-4-2-3 (explicit)

Accessible via the constant `ARK324L2SA_ERK_4_2_3` to *`ARKStepSetTableNum()`*, *`ERKStepSetTableNum()`* or *`ARKodeButcherTable_LoadERK()`*. This is the explicit portion of the default 3rd order additive method (from *[KC2003]*).

$$\begin{array}{r|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1767732205903}{2027836641118} & \frac{1767732205903}{2027836641118} & 0 & 0 & 0 \\ \frac{3}{5} & \frac{5535828885825}{10492691773637} & \frac{788022342437}{10882634858940} & 0 & 0 \\ 1 & \frac{6485989280629}{16251701735622} & -\frac{4246266847089}{9704473918619} & \frac{10755448449292}{10357097424841} & 0 \\ \hline 3 & \frac{1471266399579}{7840856788654} & -\frac{4482444167858}{7529755066697} & \frac{11266239266428}{11593286722821} & \frac{1767732205903}{4055673282236} \\ 2 & \frac{2756255671327}{12835298489170} & -\frac{10771552573575}{22201958757719} & \frac{9247589265047}{10645013368117} & \frac{2193209047091}{5459859503100} \end{array}$$

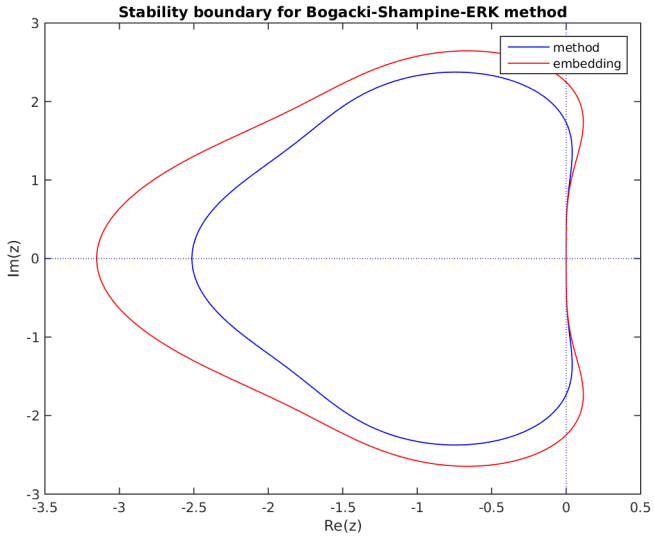


Fig. 17.2: Linear stability region for the Bogacki-Shampine method. The method's region is outlined in blue; the embedding's region is in red.

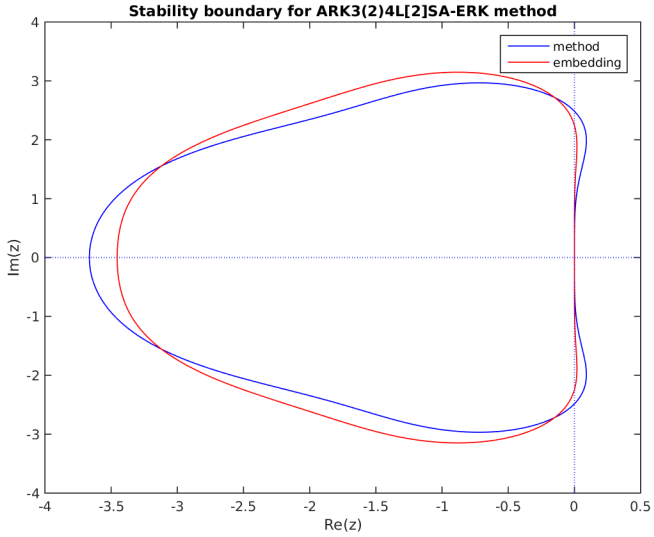


Fig. 17.3: Linear stability region for the explicit ARK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

17.1.4 Knoth-Wolke-3-3

Accessible via the constant `KNOTH_WOLKE_3_3` to `MRIStepSetMRITableNum()` and *ARKodeButcherTable_LoadERK()*. This is the default 3th order slow and fast MRIStep method (from *[KW1998]*).

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{3}{4} & -\frac{3}{16} & \frac{15}{16} & 0 \\ \hline 3 & \frac{1}{6} & \frac{3}{10} & \frac{8}{15} \end{array}$$

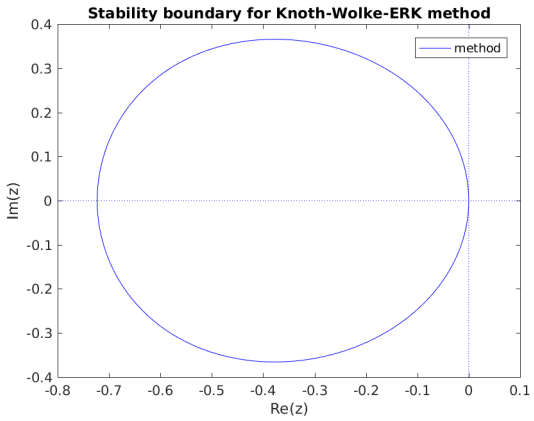


Fig. 17.4: Linear stability region for the Knoth-Wolke method

17.1.5 Zonneveld-5-3-4

Accessible via the constant `ZONNEVELD_5_3_4` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()*. This is the default 4th order explicit method (from *[Z1963]*).

$$\begin{array}{c|ccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ \frac{3}{4} & \frac{5}{32} & \frac{7}{32} & \frac{13}{32} & -\frac{1}{32} & 0 \\ \hline 4 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & 0 \\ 3 & -\frac{1}{2} & \frac{7}{3} & \frac{7}{3} & \frac{13}{6} & -\frac{16}{3} \end{array}$$

17.1.6 ARK-6-3-4 (explicit)

Accessible via the constant `ARK436L2SA_ERK_6_3_4` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()*. This is the explicit portion of the

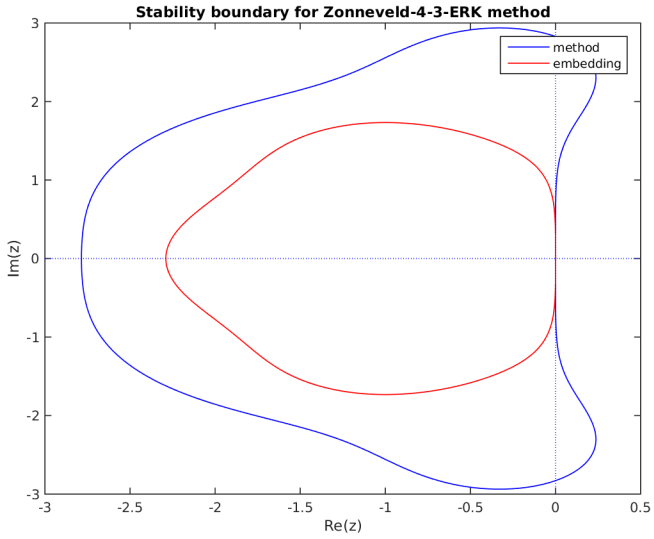


Fig. 17.5: Linear stability region for the Zonneveld method. The method's region is outlined in blue; the embedding's region is in red.

default 4th order additive method (from *[KC2003]*).

0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	0
$\frac{83}{250}$	$\frac{13861}{62500}$	$\frac{6889}{62500}$	0	0	0	0
$\frac{31}{50}$	$-\frac{116923316275}{2393684061468}$	$-\frac{2731218467317}{15368042101831}$	$\frac{9408046702089}{11113171139209}$	0	0	0
$\frac{17}{20}$	$-\frac{451086348788}{2902428689909}$	$-\frac{2682348792572}{7519795681897}$	$\frac{12662868775082}{11960479115383}$	$\frac{3355817975965}{11060851509271}$	0	0
1	$\frac{647845179188}{3216320057751}$	$\frac{73281519250}{8382639484533}$	$\frac{552539513391}{3454668386233}$	$\frac{3354512671639}{8306763924573}$	$\frac{4040}{17871}$	0
4	$\frac{82889}{524892}$	0	$\frac{15625}{83664}$	$\frac{69875}{102672}$	$-\frac{2260}{8211}$	$\frac{1}{4}$
3	$\frac{4586570599}{29645900160}$	0	$\frac{178811875}{945068544}$	$\frac{814220225}{1159782912}$	$-\frac{3700637}{11593932}$	$\frac{61727}{225920}$

17.1.7 ARK-7-3-4 (explicit)

Accessible via the constant `ARK437L2SA_ERK_7_3_4` to *`ARKStepSetTableNum()`*, *`ERKStepSetTableNum()`* or *`ARKodeButcherTable_LoadERK()`*. This is the explicit portion of the

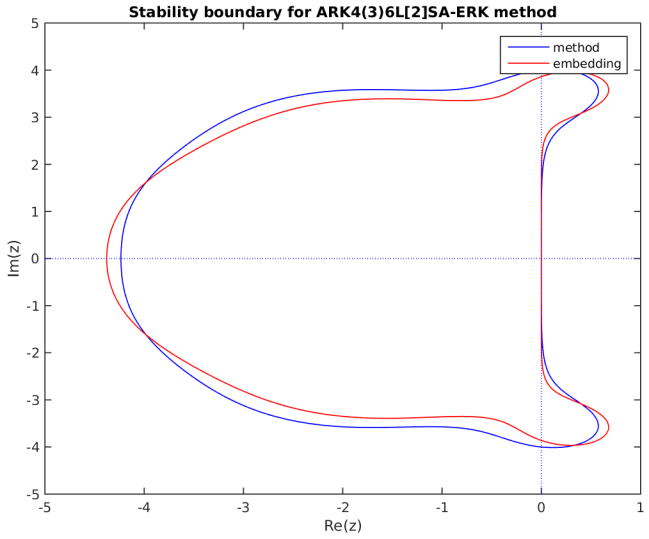


Fig. 17.6: Linear stability region for the explicit ARK-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

4th order additive method (from *[KC2019]*).

0	0	0	0	0	0	0	0
$\frac{247}{1000}$	$\frac{247}{1000}$	0	0	0	0	0	0
$\frac{4276536705230}{10142255878289}$	$\frac{247}{4000}$	$\frac{2694949928731}{7487940209513}$	0	0	0	0	0
$\frac{67}{200}$	$\frac{464650059369}{8764239774964}$	$\frac{878889893998}{2444806327765}$	$-\frac{952945855348}{12294611323341}$	0	0	0	0
$\frac{3}{40}$	$\frac{476636172619}{8159180917465}$	$-\frac{1271469283451}{7793814740893}$	$-\frac{859560642026}{4356155882851}$	$\frac{1723805262919}{4571918432560}$	0	0	0
$\frac{7}{10}$	$\frac{6338158500785}{11769362343261}$	$-\frac{4970555480458}{10924838743837}$	$\frac{3326578051521}{2647936831840}$	$-\frac{880713585975}{1841400956686}$	$-\frac{1428733748635}{8843423958496}$	0	0
1	$\frac{760814592956}{3276306540349}$	$\frac{760814592956}{3276306540349}$	$-\frac{47223648122716}{6934462133451}$	$\frac{71187472546993}{9669769126921}$	$-\frac{13330509492149}{9695768672337}$	$\frac{11565764226357}{8513123442827}$	0
4	0	0	$\frac{9164257142617}{17756377923965}$	$-\frac{10812980402763}{74029279521829}$	$\frac{1335994250573}{5691609445217}$	$\frac{2273837961795}{8368240463276}$	$\frac{247}{2000}$
3	0	0	$\frac{4469248916618}{8635866897933}$	$-\frac{621260224600}{4094290005349}$	$\frac{696572312987}{2942599194819}$	$\frac{1532940081127}{5565293938103}$	$\frac{2441}{20000}$

17.1.8 Sayfy-Aburub-6-3-4

Accessible via the constant `SAYFY_ABURUB_6_3_4` to *`ARKStepSetTableNum()`*, *`ERKStepSetTableNum()`* or *`ARKodeButcherTable_LoadERK()`* (from *[SA2002]*).

0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	0
1	-1	2	0	0	0	0
1	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	0	0	0
$\frac{1}{2}$	0.137	0.226	0.137	0	0	0
1	0.452	-0.904	-0.548	0	2	0
4	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{12}$	0	$\frac{1}{3}$	$\frac{1}{12}$
3	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	0	0	0

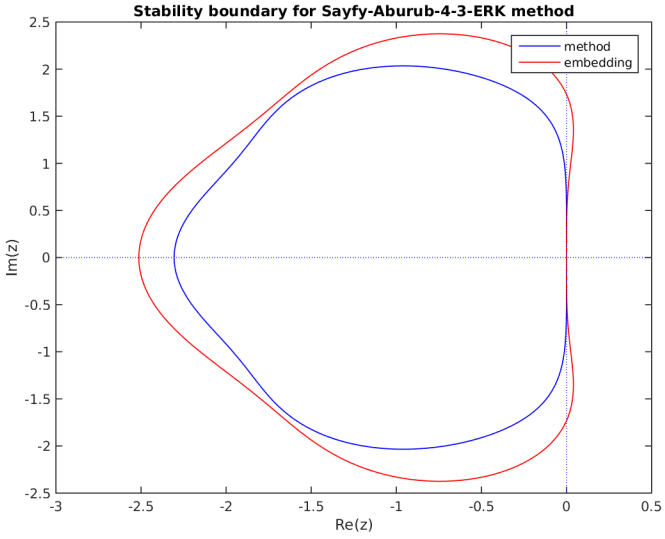


Fig. 17.7: Linear stability region for the Sayfy-Aburub-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

17.1.9 Cash-Karp-6-4-5

Accessible via the constant `CASH_KARP_6_4_5` to *`ARKStepSetTableNum()`*, *`ERKStepSetTableNum()`* or *`ARKodeButcherTable_LoadERK()`*. This is the default 5th order explicit method (from *[CK1990]*).

$$
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 & 0 & 0 \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & 0 & 0 & 0 & 0 \\
\frac{3}{5} & \frac{3}{10} & -\frac{9}{10} & \frac{6}{5} & 0 & 0 & 0 \\
1 & -\frac{11}{54} & \frac{5}{2} & -\frac{70}{27} & \frac{35}{27} & 0 & 0 \\
\frac{7}{8} & \frac{1631}{55296} & \frac{175}{512} & \frac{575}{13824} & \frac{44275}{110592} & \frac{253}{4096} & 0 \\
\hline
5 & \frac{37}{378} & 0 & \frac{250}{621} & \frac{125}{594} & 0 & \frac{512}{1771} \\
4 & \frac{2825}{27648} & 0 & \frac{18575}{48384} & \frac{13525}{55296} & \frac{277}{14336} & \frac{1}{4}
\end{array}
$$

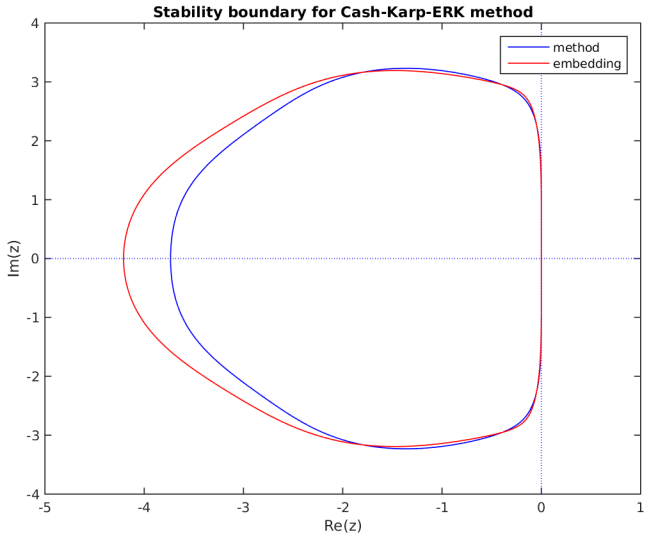


Fig. 17.8: Linear stability region for the Cash-Karp method. The method's region is outlined in blue; the embedding's region is in red.

17.1.10 Fehlberg-6-4-5

Accessible via the constant `FEHLBERG_6_4_5` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()* (from *[F1969]*).

$$
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0 \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & 0 & 0 & 0 \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & 0 & 0 \\
\frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & 0 \\
\hline
5 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55} \\
4 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0
\end{array}
$$

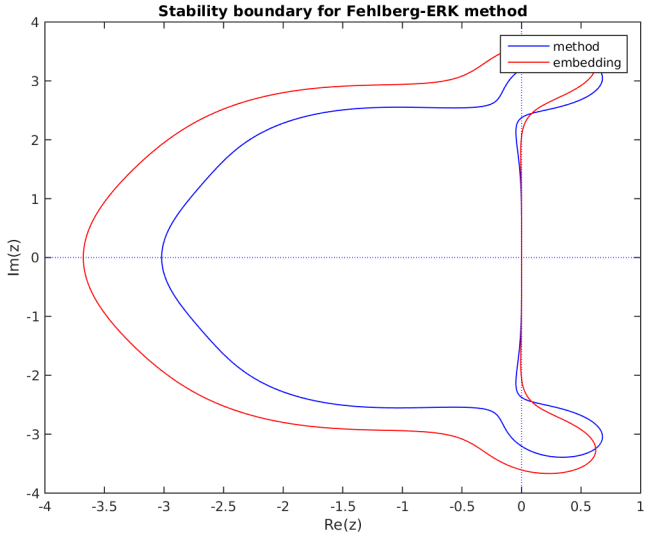


Fig. 17.9: Linear stability region for the Fehlberg method. The method's region is outlined in blue; the embedding's region is in red.

17.1.11 Dormand-Prince-7-4-5

Accessible via the constant `DORMAND_PRINCE_7_4_5` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()* (from *[DP1980]*).

0	0	0	0	0	0	0	0
$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0	0
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	0	0
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$	0	0	0	0
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$	0	0	0
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$	0	0
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
5	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
4	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

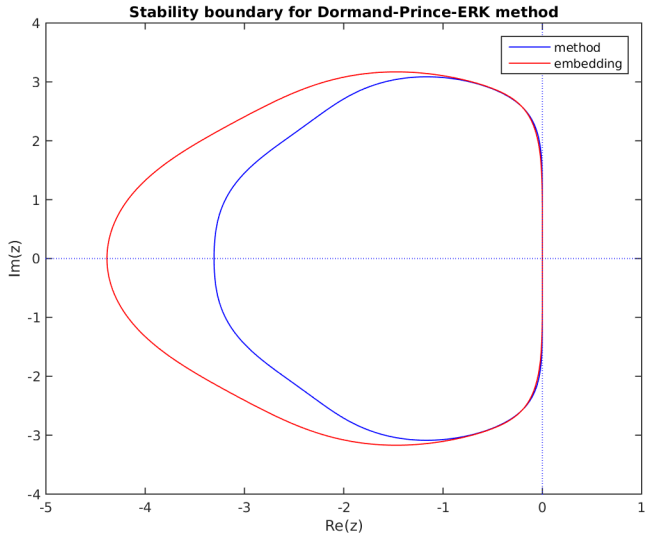


Fig. 17.10: Linear stability region for the Dormand-Prince method. The method's region is outlined in blue; the embedding's region is in red.

17.1.12 ARK-8-4-5 (explicit)

Accessible via the constant `ARK548L2SA_ERK_8_4_5` to *ARKStepSetTableNum()*, *ERKStepSetTableNum()* or *ARKodeButcherTable_LoadERK()*. This is the explicit portion of the

default 5th order additive method (from *[KC2003]*).

0	0	0	0	0	0	0	0
$\frac{41}{100}$	$\frac{41}{100}$	0	0	0	0	0	0
$\frac{2935347310677}{11292855782101}$	$\frac{367902744464}{2072280473677}$	$\frac{677623207551}{8224143866563}$	0	0	0	0	0
$\frac{1426016391358}{7196633302097}$	$\frac{1268023523408}{10340822734521}$	0	$\frac{1029933939417}{13636558850479}$	0	0	0	0
$\frac{92}{100}$	$\frac{14463281900351}{6315353703477}$	0	$\frac{66114435211212}{5879490589093}$	$-\frac{54053170152839}{4284798021562}$	0	0	0
$\frac{24}{100}$	$\frac{14090043504691}{34967701212078}$	0	$\frac{15191511035443}{11219624916014}$	$-\frac{18461159152457}{12425892160975}$	$-\frac{281667163811}{9011619295870}$	0	0
$\frac{3}{5}$	$\frac{19230459214898}{13134317526959}$	0	$\frac{21275331358303}{2942455364971}$	$-\frac{38145345988419}{4862620318723}$	$-\frac{1}{8}$	$-\frac{1}{8}$	0
1	$-\frac{19977161125411}{11928030595625}$	0	$-\frac{40795976796054}{6384907823539}$	$\frac{177454434618887}{12078138498510}$	$\frac{782672205425}{8267701900261}$	$-\frac{69563011059811}{9646580694205}$	$\frac{735662}{494218}$
5	$-\frac{872700587467}{9133579230613}$	0	0	$\frac{22348218063261}{9555858737531}$	$-\frac{1143369518992}{8141816002931}$	$-\frac{39379526789629}{19018526304540}$	$\frac{3272738}{4290004}$
4	$-\frac{975461918565}{9796059967033}$	0	0	$\frac{78070527104295}{32432590147079}$	$-\frac{548382580838}{3424219808633}$	$-\frac{33438840321285}{15594753105479}$	$\frac{362980}{465618}$

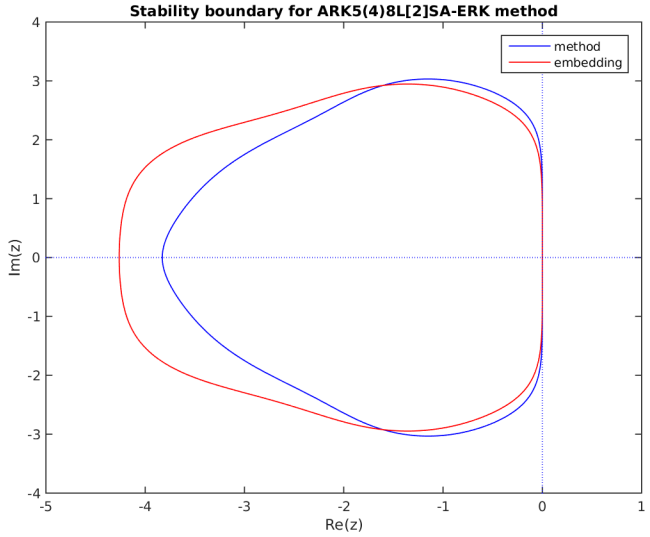


Fig. 17.11: Linear stability region for the explicit ARK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

17.1.13 ARK-8-4-5b (explicit)

Accessible via the constant `ARK548L2SAb_ERK_8_4_5` to *`ARKStepSetTableNum()`*, *`ERKStepSetTableNum()`* or *`ARKodeButcherTable_LoadERK()`*. This is the explicit portion of the

5th order additive method from *[KC2019]*.

0	0	0	0	0	0	0	0
$\frac{4}{9}$	$\frac{4}{9}$	0	0	0	0	0	0
$\frac{6456083330201}{8509243623797}$	$\frac{1}{9}$	$\frac{1183333538310}{1827251437969}$	0	0	0	0	0
$\frac{1632083962415}{14158861528103}$	$\frac{895379019517}{9750411845327}$	$\frac{477606656805}{13473228687314}$	$\frac{-112564739183}{9373365219272}$	0	0	0	0
$\frac{6365430648612}{17842476412687}$	$\frac{-4458043123994}{13015289567637}$	$\frac{-2500665203865}{9342069639922}$	$\frac{983347055801}{8893519644487}$	$\frac{2185051477207}{2551468980502}$	0	0	0
$\frac{18}{25}$	$\frac{-167316361917}{17121522574472}$	$\frac{1605541814917}{7619724128744}$	$\frac{991021770328}{13052792161721}$	$\frac{2342280609577}{11279663441611}$	$\frac{3012424348531}{12792462456678}$	0	0
$\frac{191}{200}$	$\frac{6680998715867}{14310383562358}$	$\frac{5029118570809}{3897454228471}$	$\frac{2415062538259}{6382199904604}$	$\frac{-3924368632305}{6964820224454}$	$\frac{-4331110370267}{15021686902756}$	$\frac{-3944303808049}{11994238218192}$	0
1	$\frac{2193717860234}{3570523412979}$	$\frac{2193717860234}{3570523412979}$	$\frac{5952760925747}{18750164281544}$	$\frac{-4412967128996}{6196664114337}$	$\frac{4151782504231}{36106512998704}$	$\frac{572599549169}{6265429158920}$	$\frac{-4578743561}{11306498036}$
5	0	0	$\frac{3517720773327}{20256071687669}$	$\frac{4569610470461}{17934693873752}$	$\frac{2819471173109}{11655438449929}$	$\frac{3296210113763}{10722700128969}$	$\frac{-1142099968}{57109839269}$
4	0	0	$\frac{520639020421}{8300446712847}$	$\frac{4550235134915}{17827758688493}$	$\frac{1482366381361}{6201654941325}$	$\frac{5551607622171}{13911031047899}$	$\frac{-5266607656}{367889688439}$

17.1.14 Verner-8-5-6

Accessible via the constant `VERNER_8_5_6` to `ARKStepSetTableNum()`, `ERKStepSetTableNum()` or `ARKodeButcherTable_LoadERK()`. This is the default 6th order explicit method (from *[V1978]*).

0	0	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{1}{6}$	0	0	0	0	0	0	0
$\frac{4}{15}$	$\frac{4}{75}$	$\frac{16}{75}$	0	0	0	0	0	0
$\frac{2}{3}$	$\frac{5}{6}$	$-\frac{8}{3}$	$\frac{5}{2}$	0	0	0	0	0
$\frac{5}{6}$	$-\frac{165}{64}$	$\frac{55}{6}$	$-\frac{425}{64}$	$\frac{85}{96}$	0	0	0	0
1	$\frac{12}{5}$	-8	$\frac{4015}{612}$	$-\frac{11}{36}$	$\frac{88}{255}$	0	0	0
$\frac{1}{15}$	$-\frac{8263}{15000}$	$\frac{124}{75}$	$-\frac{643}{680}$	$-\frac{81}{250}$	$\frac{2484}{10625}$	0	0	0
1	$\frac{3501}{1720}$	$-\frac{300}{43}$	$\frac{297275}{52632}$	$-\frac{319}{2322}$	$\frac{24068}{84065}$	0	$\frac{3850}{26703}$	0
6	$\frac{3}{40}$	0	$\frac{875}{2244}$	$\frac{23}{72}$	$\frac{264}{1955}$	0	$\frac{125}{11592}$	$\frac{43}{616}$
5	$\frac{13}{160}$	0	$\frac{2375}{5984}$	$\frac{5}{16}$	$\frac{12}{85}$	$\frac{3}{44}$	0	0

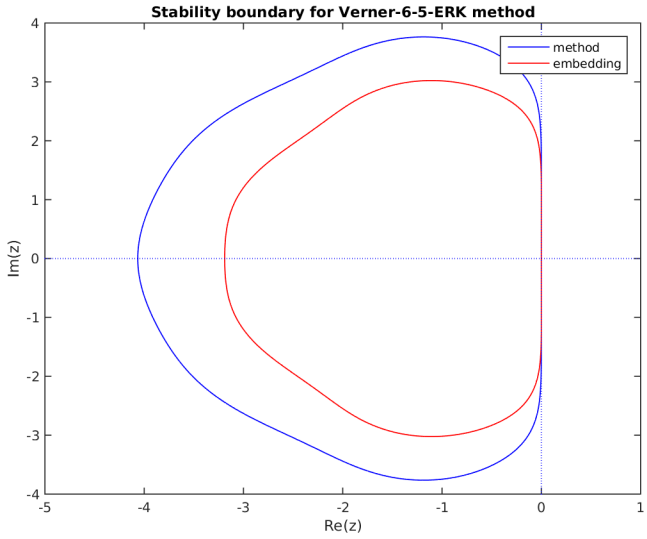


Fig. 17.12: Linear stability region for the Verner-8-5-6 method. The method's region is outlined in blue; the embedding's region is in red.

17.1.15 Fehlberg-13-7-8

Accessible via the constant `FEHLBERG_13_7_8` to *`ARKStepSetTableNum()`*, *`ERKStepSetTableNum()`* or *`ARKodeButcherTable_LoadERK()`*. This is the default 8th order explicit method (from *[B2008]*).

0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{2}{27}$	$\frac{2}{27}$	0	0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{12}$	0	0	0	0	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{1}{24}$	0	$\frac{1}{8}$	0	0	0	0	0	0	0	0	0	0
$\frac{5}{12}$	$\frac{5}{12}$	0	$-\frac{25}{16}$	$\frac{25}{16}$	0	0	0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{20}$	0	0	$\frac{1}{4}$	$\frac{1}{5}$	0	0	0	0	0	0	0	0
$\frac{5}{6}$	$-\frac{25}{108}$	0	0	$\frac{125}{108}$	$-\frac{65}{27}$	$\frac{125}{54}$	0	0	0	0	0	0	0
$\frac{1}{6}$	$\frac{31}{300}$	0	0	0	$\frac{61}{225}$	$-\frac{2}{9}$	$\frac{13}{900}$	0	0	0	0	0	0
$\frac{2}{3}$	2	0	0	$-\frac{53}{6}$	$\frac{704}{45}$	$-\frac{107}{9}$	$\frac{67}{90}$	3	0	0	0	0	0
$\frac{1}{3}$	$-\frac{91}{108}$	0	0	$\frac{23}{108}$	$-\frac{976}{135}$	$\frac{311}{54}$	$-\frac{19}{60}$	$\frac{17}{6}$	$-\frac{1}{12}$	0	0	0	0
1	$\frac{2383}{4100}$	0	0	$-\frac{341}{164}$	$\frac{4496}{1025}$	$-\frac{301}{82}$	$\frac{2133}{4100}$	$\frac{45}{82}$	$\frac{45}{164}$	$\frac{18}{41}$	0	0	0
0	$\frac{3}{205}$	0	0	0	0	$-\frac{6}{41}$	$-\frac{3}{205}$	$-\frac{3}{41}$	$\frac{3}{41}$	$\frac{6}{41}$	0	0	0
1	$-\frac{1777}{4100}$	0	0	$-\frac{341}{164}$	$\frac{4496}{1025}$	$-\frac{289}{82}$	$\frac{2193}{4100}$	$\frac{51}{82}$	$\frac{33}{164}$	$\frac{12}{41}$	0	1	0
8	0	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	0	$\frac{41}{840}$	$\frac{41}{840}$
7	$\frac{41}{840}$	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	$\frac{41}{840}$	0	0

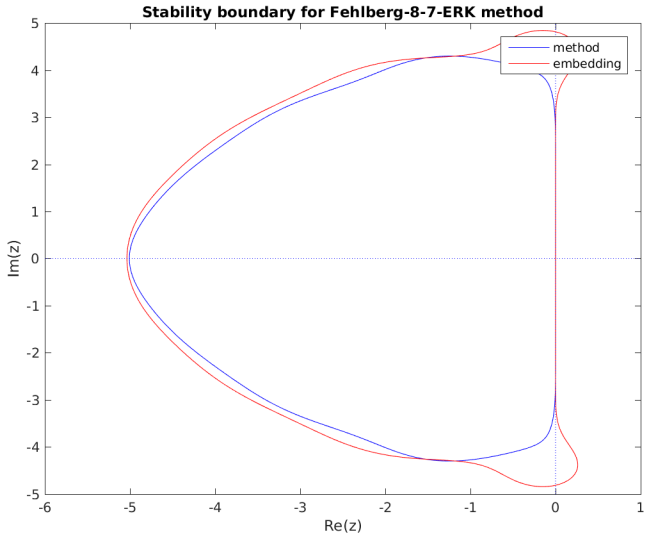


Fig. 17.13: Linear stability region for the Fehlberg-13-7-8 method. The method's region is outlined in blue; the embedding's region is in red.

17.2 Implicit Butcher tables

In the category of diagonally implicit Runge-Kutta methods, ARKode includes methods that have orders 2 through 5, with embeddings that are of orders 1 through 4.

17.2.1 SDIRK-2-1-2

Accessible via the constant `SDIRK_2_1_2` to *ARKStepSetTableNum()* or *ARKodeButcherTable_LoadDIRK()*. This is the default 2nd order implicit method. Both the method and embedding are A- and B-stable.

$$\begin{array}{c|cc} 1 & 1 & 0 \\ 0 & -1 & 1 \\ \hline 2 & \frac{1}{2} & \frac{1}{2} \\ 1 & 1 & 0 \end{array}$$

17.2.2 Billington-3-3-2

Accessible via the constant `BILLINGTON_3_3_2` to *ARKStepSetTableNum()* or *ARKodeButcherTable_LoadDIRK()*. Here, the higher-order embedding is less stable than the lower-order

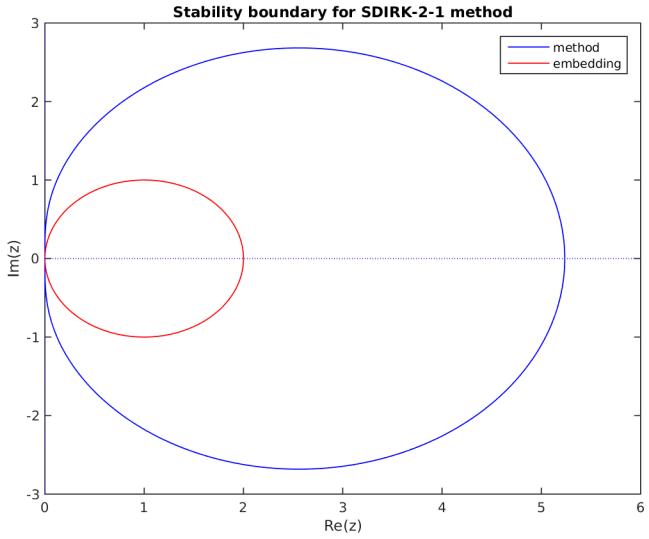


Fig. 17.14: Linear stability region for the SDIRK-2-1-2 method. The method's region is outlined in blue; the embedding's region is in red.

method (from *[B1983]*).

0.292893218813	0.292893218813	0	0
1.091883092037	0.798989873223	0.292893218813	0
1.292893218813	0.740789228841	0.259210771159	0.292893218813
2	0.740789228840	0.259210771159	0
3	0.691665115992	0.503597029883	−0.195262145876

17.2.3 TRBDF2-3-3-2

Accessible via the constant `TRBDF2_3_3_2` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. As with Billington, here the higher-order embedding is less stable than the lower-order method (from *[B1985]*).

0	0	0	0
$2-\sqrt{2}$	$\frac{2-\sqrt{2}}{2}$	$\frac{2-\sqrt{2}}{2}$	0
1	$\frac{\sqrt{2}}{4}$	$\frac{\sqrt{2}}{4}$	$\frac{2-\sqrt{2}}{2}$
2	$\frac{\sqrt{2}}{4}$	$\frac{\sqrt{2}}{4}$	$\frac{2-\sqrt{2}}{2}$
3	$\frac{1-\frac{\sqrt{2}}{4}}{3}$	$\frac{\frac{3\sqrt{2}}{4}+1}{3}$	$\frac{2-\sqrt{2}}{6}$

17.2.4 Kvaerno-4-2-3

Accessible via the constant `KVAERNO_4_2_3` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. Both the method and embedding are A-stable; additionally the

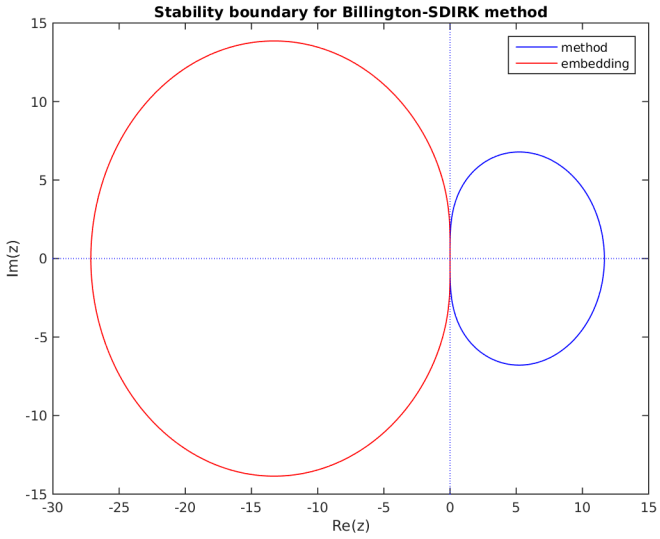


Fig. 17.15: Linear stability region for the Billington method. The method's region is outlined in blue; the embedding's region is in red.

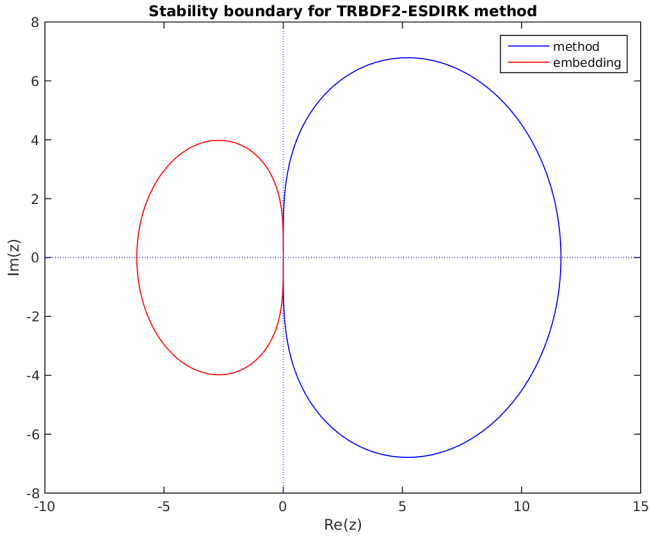


Fig. 17.16: Linear stability region for the TRBDF2 method. The method's region is outlined in blue; the embedding's region is in red.

method is L-stable (from *[K2004]*).

0	0	0	0	0
0.871733043	0.4358665215	0.4358665215	0	0
1	0.490563388419108	0.073570090080892	0.4358665215	0
1	0.308809969973036	1.490563388254106	−1.235239879727145	0.4358665215
3	0.308809969973036	1.490563388254106	−1.235239879727145	0.4358665215
2	0.490563388419108	0.073570090080892	0.4358665215	0

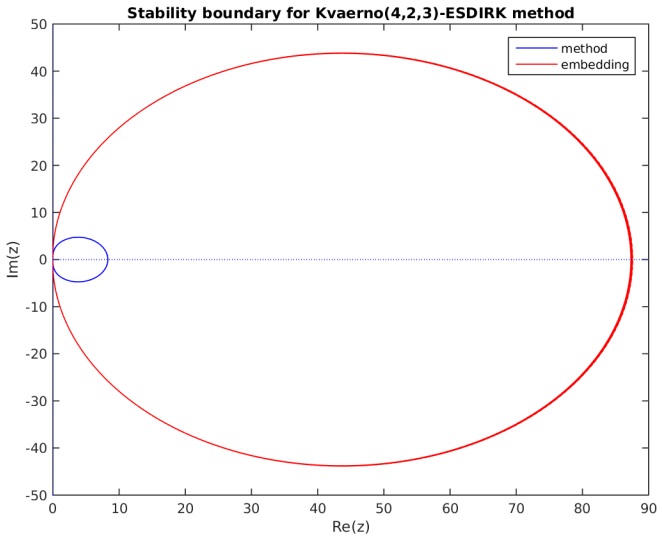


Fig. 17.17: Linear stability region for the Kvaerno-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

17.2.5 ARK-4-2-3 (implicit)

Accessible via the constant `ARK324L2SA_DIRK_4_2_3` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. This is the default 3rd order implicit method, and the implicit portion of the default 3rd order additive method. Both the method and embedding are A-stable; additionally the method is L-stable (from *[KC2003]*).

0	0	0	0	0
$\frac{1767732205903}{2027836641118}$	$\frac{1767732205903}{4055673282236}$	$\frac{1767732205903}{4055673282236}$	0	0
$\frac{3}{5}$	$\frac{2746238789719}{10658868560708}$	$-\frac{640167445237}{6845629431997}$	$\frac{1767732205903}{4055673282236}$	0
1	$\frac{1471266399579}{7840856788654}$	$-\frac{4482444167858}{7529755066697}$	$\frac{11266239266428}{11593286722821}$	$\frac{1767732205903}{4055673282236}$
3	$\frac{1471266399579}{7840856788654}$	$-\frac{4482444167858}{7529755066697}$	$\frac{11266239266428}{11593286722821}$	$\frac{1767732205903}{4055673282236}$
2	$\frac{2756255671327}{12835298489170}$	$-\frac{10771552573575}{22201958757719}$	$\frac{9247589265047}{10645013368117}$	$\frac{2193209047091}{5459859503100}$

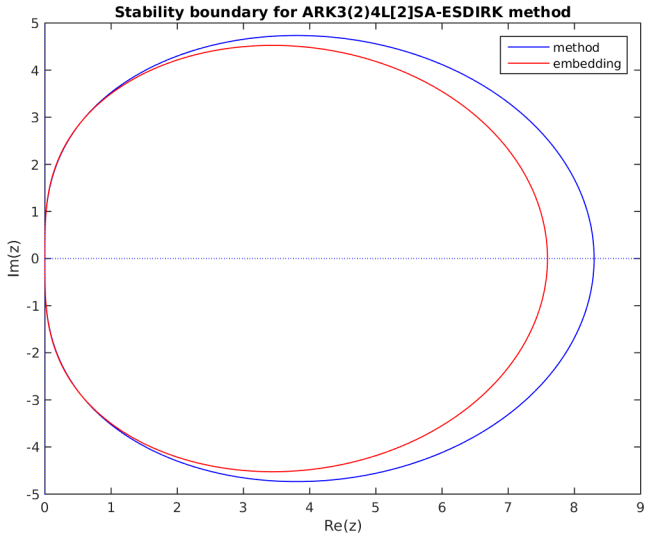


Fig. 17.18: Linear stability region for the implicit ARK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

17.2.6 Cash-5-2-4

Accessible via the constant `CASH_5_2_4` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. Both the method and embedding are A-stable; additionally the method is L-stable (from *[C1979]*).

0.435866521508	0.435866521508	0	0	0	0
−0.7	−1.13586652150	0.435866521508	0	0	0
0.8	1.08543330679	−0.721299828287	0.435866521508	0	0
0.924556761814	0.416349501547	0.190984004184	−0.118643265417	0.435866521508	0
1	0.896869652944	0.0182725272734	−0.0845900310706	−0.266418670647	0.435866521508
4	0.896869652944	0.0182725272734	−0.0845900310706	−0.266418670647	0.435866521508
2	1.05646216107052	−0.0564621610705236	0	0	0

17.2.7 Cash-5-3-4

Accessible via the constant `CASH_5_3_4` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. Both the method and embedding are A-stable; additionally the

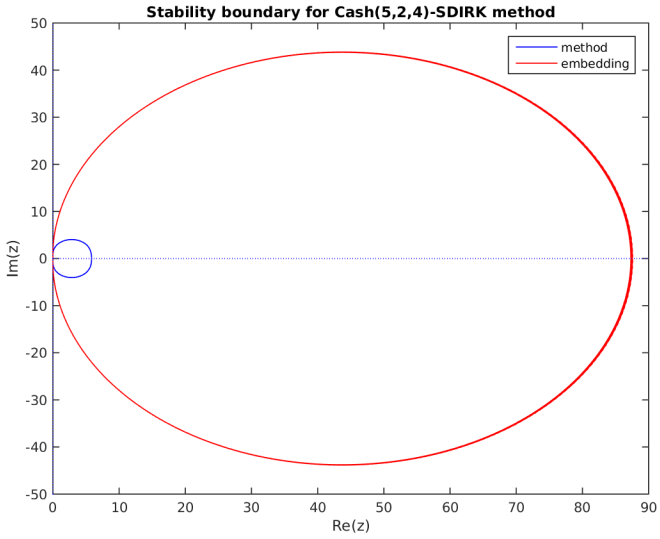


Fig. 17.19: Linear stability region for the Cash-5-2-4 method. The method's region is outlined in blue; the embedding's region is in red.

method is L-stable (from *[C1979]*).

0.435866521508	0.435866521508	0	0	0	0
−0.7	−1.13586652150	0.435866521508	0	0	0
0.8	1.08543330679	−0.721299828287	0.435866521508	0	0
0.924556761814	0.416349501547	0.190984004184	−0.118643265417	0.435866521508	0
1	0.896869652944	0.0182725272734	−0.0845900310706	−0.266418670647	0.435866521508
4	0.896869652944	0.0182725272734	−0.0845900310706	−0.266418670647	0.435866521508
3	0.776691932910	0.0297472791484	−0.0267440239074	0.220304811849	0

17.2.8 SDIRK-5-3-4

Accessible via the constant `SDIRK_5_3_4` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. This is the default 4th order implicit method. Here, the method is both A- and L-stable, although the embedding has reduced stability (from *[HW1996]*).

$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0
$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{4}$	0	0	0
$\frac{11}{20}$	$\frac{17}{50}$	$-\frac{1}{25}$	$\frac{1}{4}$	0	0
$\frac{1}{2}$	$\frac{371}{1360}$	$-\frac{137}{2720}$	$\frac{15}{544}$	$\frac{1}{4}$	0
1	$\frac{25}{24}$	$-\frac{49}{48}$	$\frac{125}{16}$	$-\frac{85}{12}$	$\frac{1}{4}$
4	$\frac{25}{24}$	$-\frac{49}{48}$	$\frac{125}{16}$	$-\frac{85}{12}$	$\frac{1}{4}$
3	$\frac{59}{48}$	$-\frac{17}{96}$	$\frac{225}{32}$	$-\frac{85}{12}$	0

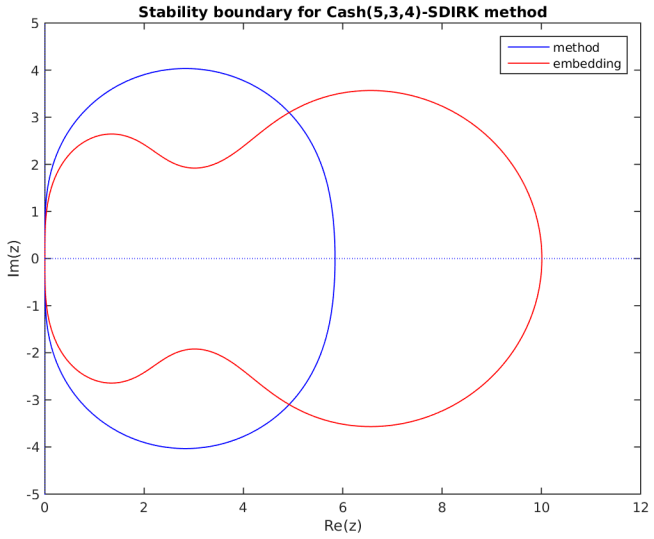


Fig. 17.20: Linear stability region for the Cash-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

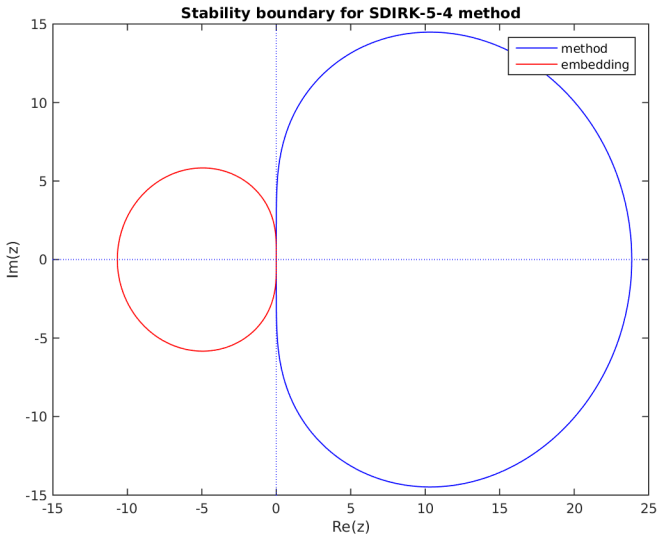


Fig. 17.21: Linear stability region for the SDIRK-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

17.2.9 Kvaerno-5-3-4

Accessible via the constant `KVAERNO_5_3_4` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. Both the method and embedding are A-stable (from *[K2004]*).

0	0	0	0	0	0
0.871733043	0.4358665215	0.4358665215	0	0	0
0.468238744853136	0.140737774731968	−0.108365551378832	0.4358665215	0	0
1	0.102399400616089	−0.376878452267324	0.838612530151233	0.4358665215	0
1	0.157024897860995	0.117330441357768	0.61667803039168	−0.326899891110444	0.4358665215
4	0.157024897860995	0.117330441357768	0.61667803039168	−0.326899891110444	0.4358665215
3	0.102399400616089	−0.376878452267324	0.838612530151233	0.4358665215	0

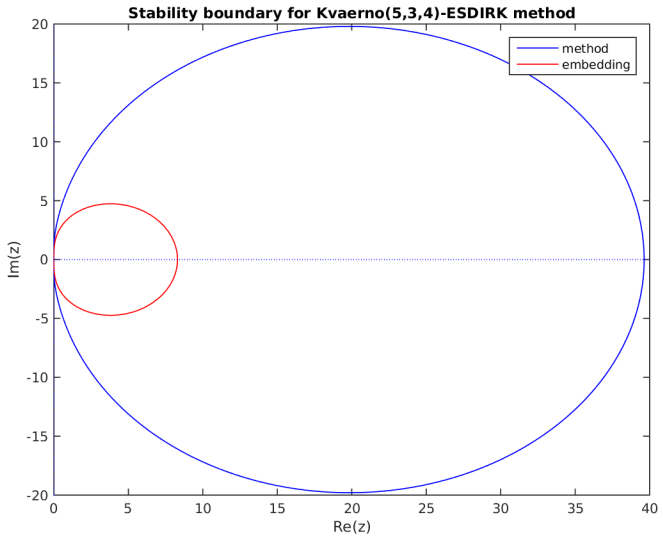


Fig. 17.22: Linear stability region for the Kvaerno-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

17.2.10 ARK-6-3-4 (implicit)

Accessible via the constant `ARK436L2SA_DIRK_6_3_4` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. This is the implicit portion of the default 4th order additive method.

Both the method and embedding are A-stable; additionally the method is L-stable (from *[KC2003]*).

0	0	0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0
$\frac{83}{250}$	$\frac{8611}{62500}$	$-\frac{1743}{31250}$	$\frac{1}{4}$	0	0	0
$\frac{31}{50}$	$\frac{5012029}{34652500}$	$-\frac{654441}{2922500}$	$\frac{174375}{388108}$	$\frac{1}{4}$	0	0
$\frac{17}{20}$	$\frac{15267082809}{155376265600}$	$-\frac{71443401}{120774400}$	$\frac{730878875}{902184768}$	$\frac{2285395}{8070912}$	$\frac{1}{4}$	0
1	$\frac{82889}{524892}$	0	$\frac{15625}{83664}$	$\frac{69875}{102672}$	$-\frac{2260}{8211}$	$\frac{1}{4}$
4	$\frac{82889}{524892}$	0	$\frac{15625}{83664}$	$\frac{69875}{102672}$	$-\frac{2260}{8211}$	$\frac{1}{4}$
3	$\frac{4586570599}{29645900160}$	0	$\frac{178811875}{945068544}$	$\frac{814220225}{1159782912}$	$-\frac{3700637}{11593932}$	$\frac{61727}{225920}$

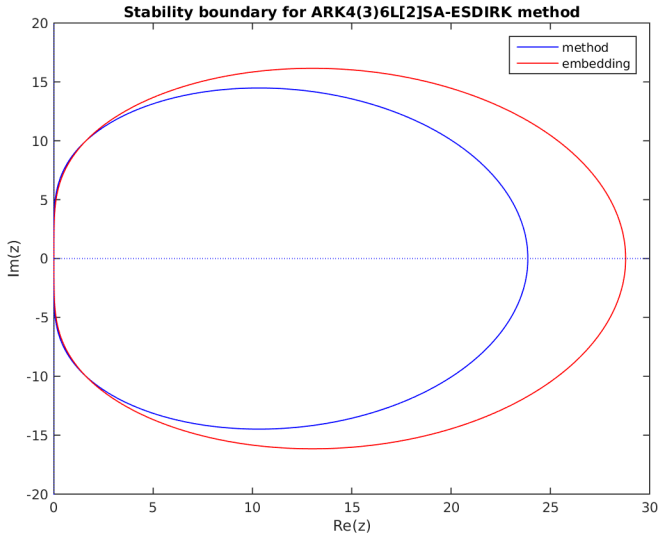


Fig. 17.23: Linear stability region for the implicit ARK-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

17.2.11 ARK-7-3-4 (implicit)

Accessible via the constant `ARK437L2SA_DIRK_7_3_4` to *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. This is the implicit portion of the 4th order additive method from

[KC2019].

0	0	0	0	0	0	0	0
$\frac{247}{1000}$	$\frac{1235}{10000}$	$\frac{1235}{10000}$	0	0	0	0	0
$\frac{4276536705230}{10142255878289}$	$\frac{624185399699}{4186980696204}$	$\frac{624185399699}{4186980696204}$	$\frac{1235}{10000}$	0	0	0	0
$\frac{67}{200}$	$\frac{1258591069120}{10082082980243}$	$\frac{1258591069120}{10082082980243}$	$-\frac{322722984531}{8455138723562}$	$\frac{1235}{10000}$	0	0	0
$\frac{3}{40}$	$-\frac{436103496990}{5971407786587}$	$-\frac{436103496990}{5971407786587}$	$-\frac{2689175662187}{11046760208243}$	$\frac{4431412449334}{12995360898505}$	$\frac{1235}{10000}$	0	0
$\frac{7}{10}$	$-\frac{2207373168298}{14430576638973}$	$-\frac{2207373168298}{14430576638973}$	$\frac{242511121179}{3358618340039}$	$\frac{3145666661981}{7780404714551}$	$\frac{5882073923981}{14490790706663}$	$\frac{1235}{10000}$	0
1	0	0	$\frac{9164257142617}{17756377923965}$	$-\frac{10812980402763}{74029279521829}$	$\frac{1335994250573}{5691609445217}$	$\frac{2273837961795}{8368240463276}$	$\frac{1235}{10000}$
4	0	0	$\frac{9164257142617}{17756377923965}$	$-\frac{10812980402763}{74029279521829}$	$\frac{1335994250573}{5691609445217}$	$\frac{2273837961795}{8368240463276}$	$\frac{1235}{10000}$
3	0	0	$\frac{4469248916618}{8635866897933}$	$-\frac{621260224600}{4094290005349}$	$\frac{696572312987}{2942599194819}$	$\frac{1532940081127}{5565293938103}$	$\frac{2441}{20000}$

17.2.12 Kvaerno-7-4-5

Accessible via the constant `KVAERNO_7_4_5` to *ARKStepSetTableNum()* or *ARKodeButcherTable_LoadDIRK()*. Both the method and embedding are A-stable; additionally the method is L-stable (from *[K2004]*).

0	0	0	0	0	
0.52	0.26	0.26	0	0	
1.230333209967908	0.13	0.84033320996790809	0.26	0	
0.895765984350076	0.22371961478320505	0.47675532319799699	-0.06470895363112615	0.26	
0.436393609858648	0.16648564323248321	0.10450018841591720	0.03631482272098715	-0.13090704451073998	
1	0.13855640231268224	0	-0.04245337201752043	0.02446657898003141	0.61
1	0.13659751177640291	0	-0.05496908796538376	-0.04118626728321046	0.62
5	0.13659751177640291	0	-0.05496908796538376	-0.04118626728321046	0.62
4	0.13855640231268224	0	-0.04245337201752043	0.02446657898003141	0.61

17.2.13 ARK-8-4-5 (implicit)

Accessible via the constant `ARK548L2SA_DIRK_8_4_5` for *ARKStepSetTableNum()* or *ARKodeButcherTable_LoadDIRK()*. This is the default 5th order implicit method, and the implicit portion of the default 5th order additive method. Both the method and embedding are A-stable; additionally the method is

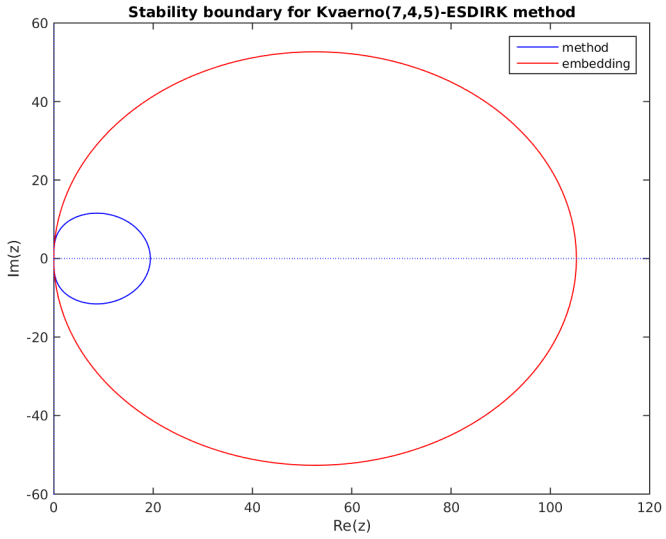


Fig. 17.24: Linear stability region for the Kvaerno-7-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

L-stable (from *[KC2003]*).

0	0	0	0	0	0	0	
$\frac{41}{100}$	$\frac{41}{200}$	$\frac{41}{200}$	0	0	0	0	
$\frac{2935347310677}{11292855782101}$	$\frac{41}{400}$	$-\frac{567603406766}{11931857230679}$	$\frac{41}{200}$	0	0	0	
$\frac{1426016391358}{7196633302097}$	$\frac{683785636431}{9252920307686}$	0	$-\frac{110385047103}{1367015193373}$	$\frac{41}{200}$	0	0	
$\frac{92}{100}$	$\frac{3016520224154}{10081342136671}$	0	$\frac{30586259806659}{12414158314087}$	$-\frac{22760509404356}{11113319521817}$	$\frac{41}{200}$	0	
$\frac{24}{100}$	$\frac{218866479029}{1489978393911}$	0	$\frac{638256894668}{5436446318841}$	$-\frac{1179710474555}{5321154724896}$	$-\frac{60928119172}{8023461067671}$	$\frac{41}{200}$	
$\frac{3}{5}$	$\frac{1020004230633}{5715676835656}$	0	$\frac{25762820946817}{25263940353407}$	$-\frac{2161375909145}{9755907335909}$	$-\frac{211217309593}{5846859502534}$	$-\frac{4269925059573}{7827059040749}$	
1	$-\frac{872700587467}{9133579230613}$	0	0	$\frac{22348218063261}{9555858737531}$	$-\frac{1143369518992}{8141816002931}$	$-\frac{39379526789629}{19018526304540}$	$\frac{327273}{429000}$
5	$-\frac{872700587467}{9133579230613}$	0	0	$\frac{22348218063261}{9555858737531}$	$-\frac{1143369518992}{8141816002931}$	$-\frac{39379526789629}{19018526304540}$	$\frac{327273}{429000}$
4	$-\frac{975461918565}{9796059967033}$	0	0	$\frac{78070527104295}{32432590147079}$	$-\frac{548382580838}{3424219808633}$	$-\frac{33438840321285}{15594753105479}$	$\frac{362980}{465618}$

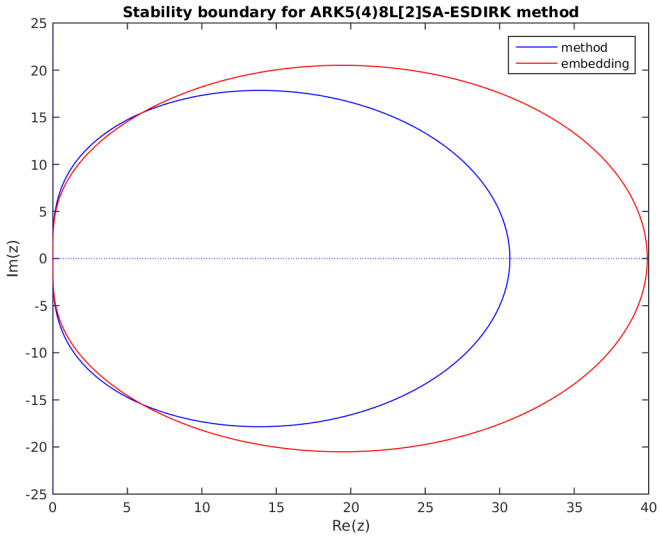


Fig. 17.25: Linear stability region for the implicit ARK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

17.2.14 ARK-8-4-5b (implicit)

Accessible via the constant `ARK548L2SAb_DIRK_8_4_5` for *`ARKStepSetTableNum()`* or *`ARKodeButcherTable_LoadDIRK()`*. This is the 5th order implicit method from *[KC2019]*.

0	0	0	0	0	0	0	
$\frac{4}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	0	0	0	0	
$\frac{6456083330201}{8509243623797}$	$\frac{2366667076620}{8822750406821}$	$\frac{2366667076620}{8822750406821}$	$\frac{2}{9}$	0	0	0	
$\frac{1632083962415}{14158861528103}$	$-\frac{257962897183}{4451812247028}$	$-\frac{257962897183}{4451812247028}$	$\frac{128530224461}{14379561246022}$	$\frac{2}{9}$	0	0	
$\frac{6365430648612}{17842476412687}$	$-\frac{486229321650}{11227943450093}$	$-\frac{486229321650}{11227943450093}$	$-\frac{225633144460}{6633558740617}$	$\frac{1741320951451}{6824444397158}$	$\frac{2}{9}$	0	
$\frac{18}{25}$	$\frac{621307788657}{4714163060173}$	$\frac{621307788657}{4714163060173}$	$-\frac{125196015625}{3866852212004}$	$\frac{940440206406}{7593089888465}$	$\frac{961109811699}{6734810228204}$	$\frac{2}{9}$	
$\frac{191}{200}$	$\frac{2036305566805}{6583108094622}$	$\frac{2036305566805}{6583108094622}$	$-\frac{3039402635899}{4450598839912}$	$-\frac{1829510709469}{31102090912115}$	$-\frac{286320471013}{6931253422520}$	$\frac{8651533662697}{9642993110008}$	
1	0	0	$\frac{3517720773327}{20256071687669}$	$\frac{4569610470461}{17934693873752}$	$\frac{2819471173109}{11655438449929}$	$\frac{3296210113763}{10722700128969}$	$-\frac{1142}{5710}$
5	0	0	$\frac{3517720773327}{20256071687669}$	$\frac{4569610470461}{17934693873752}$	$\frac{2819471173109}{11655438449929}$	$\frac{3296210113763}{10722700128969}$	$-\frac{1142}{5710}$
4	0	0	$\frac{520639020421}{8300446712847}$	$\frac{4550235134915}{17827758688493}$	$\frac{1482366381361}{6201654941325}$	$\frac{5551607622171}{13911031047899}$	$-\frac{5266}{3678}$

17.3 Additive Butcher tables

In the category of additive Runge-Kutta methods for split implicit and explicit calculations, ARKode includes methods that have orders 3 through 5, with embeddings that are of orders 2 through 4. These Butcher table pairs are as follows:

- 3rd-order pair: *ARK-4-2-3 (explicit)* with *ARK-4-2-3 (implicit)*, corresponding to Butcher tables `ARK324L2SA_ERK_4_2_3` and `ARK324L2SA_DIRK_4_2_3` for *`ARKStepSetTableNum()`*.

- 4th-order pair: *ARK-6-3-4 (explicit)* with *ARK-6-3-4 (implicit)*, corresponding to Butcher tables `ARK436L2SA_ERK_6_3_4` and `ARK436L2SA_DIRK_6_3_4` for *ARKStepSetTableNum()*.
- 4th-order pair: *ARK-7-3-4 (explicit)* with *ARK-7-3-4 (implicit)*, corresponding to Butcher tables `ARK437L2SA_ERK_7_3_4` and `ARK437L2SA_DIRK_7_3_4` for *ARKStepSetTableNum()*.
- 5th-order pair: *ARK-8-4-5 (explicit)* with *ARK-8-4-5 (implicit)*, corresponding to Butcher tables `ARK548L2SA_ERK_8_4_5` and `ARK548L2SA_ERK_8_4_5` for *ARKStepSetTableNum()*.
- 5th-order pair: *ARK-8-4-5b (explicit)* with *ARK-8-4-5b (implicit)*, corresponding to Butcher tables `ARK548L2SAb_ERK_8_4_5` and `ARK548L2SAb_ERK_8_4_5` for *ARKStepSetTableNum()*.

Chapter 18

Appendix: SUNDIALS Release History

Date	SUNDIALS	ARKode	CVODE	CVODES	IDA	IDAS	KINSOL
Dec 2020	5.6.0	4.6.0	5.6.0	5.6.0	5.6.0	4.6.0	5.6.0
Oct 2020	5.5.0	4.5.0	5.5.0	5.5.0	5.5.0	4.5.0	5.5.0
Sep 2020	5.4.0	4.4.0	5.4.0	5.4.0	5.4.0	4.4.0	5.4.0
May 2020	5.3.0	4.3.0	5.3.0	5.3.0	5.3.0	4.3.0	5.3.0
Mar 2020	5.2.0	4.2.0	5.2.0	5.2.0	5.2.0	4.2.0	5.2.0
Jan 2020	5.1.0	4.1.0	5.1.0	5.1.0	5.1.0	4.1.0	5.1.0
Oct 2019	5.0.0	4.0.0	5.0.0	5.0.0	5.0.0	4.0.0	5.0.0
Feb 2019	4.1.0	3.1.0	4.1.0	4.1.0	4.1.0	3.1.0	4.1.0
Jan 2019	4.0.2	3.0.2	4.0.2	4.0.2	4.0.2	3.0.2	4.0.2
Dec 2018	4.0.1	3.0.1	4.0.1	4.0.1	4.0.1	3.0.1	4.0.1
Dec 2018	4.0.0	3.0.0	4.0.0	4.0.0	4.0.0	3.0.0	4.0.0
Oct 2018	3.2.1	2.2.1	3.2.1	3.2.1	3.2.1	2.2.1	3.2.1
Sep 2018	3.2.0	2.2.0	3.2.0	3.2.0	3.2.0	2.2.0	3.2.0
Jul 2018	3.1.2	2.1.2	3.1.2	3.1.2	3.1.2	2.1.2	3.1.2
May 2018	3.1.1	2.1.1	3.1.1	3.1.1	3.1.1	2.1.1	3.1.1
Nov 2017	3.1.0	2.1.0	3.1.0	3.1.0	3.1.0	2.1.0	3.1.0
Sep 2017	3.0.0	2.0.0	3.0.0	3.0.0	3.0.0	2.0.0	3.0.0
Sep 2016	2.7.0	1.1.0	2.9.0	2.9.0	2.9.0	1.3.0	2.9.0
Aug 2015	2.6.2	1.0.2	2.8.2	2.8.2	2.8.2	1.2.2	2.8.2
Mar 2015	2.6.1	1.0.1	2.8.1	2.8.1	2.8.1	1.2.1	2.8.1
Mar 2015	2.6.0	1.0.0	2.8.0	2.8.0	2.8.0	1.2.0	2.8.0
Mar 2012	2.5.0	–	2.7.0	2.7.0	2.7.0	1.1.0	2.7.0
May 2009	2.4.0	–	2.6.0	2.6.0	2.6.0	1.0.0	2.6.0
Nov 2006	2.3.0	–	2.5.0	2.5.0	2.5.0	–	2.5.0
Mar 2006	2.2.0	–	2.4.0	2.4.0	2.4.0	–	2.4.0
May 2005	2.1.1	–	2.3.0	2.3.0	2.3.0	–	2.3.0
Apr 2005	2.1.0	–	2.3.0	2.2.0	2.3.0	–	2.3.0
Mar 2005	2.0.2	–	2.2.2	2.1.2	2.2.2	–	2.2.2
Jan 2005	2.0.1	–	2.2.1	2.1.1	2.2.1	–	2.2.1
Dec 2004	2.0.0	–	2.2.0	2.1.0	2.2.0	–	2.2.0
Jul 2002	1.0.0	–	2.0.0	1.0.0	2.0.0	–	2.0.0
Mar 2002	–	–	1.0.0[3]	–	–	–	–
Feb 1999	–	–	–	–	1.0.0[4]	–	–
Aug 1998	–	–	–	–	–	–	1.0.0[5]

Continued on next page

Table 18.1 – continued from previous page

Date	SUNDIALS	ARKode	CVODE	CVODES	IDA	IDAS	KINSOL
Jul 1997	–	–	1.0.0[2]	–	–	–	–
Sep 1994	–	–	1.0.0[1]	–	–	–	–

CVODE and PVODE combined
IDA written
KINSOL written
PVODE written
CVODE written

Bibliography

[A1965] D.G. Anderson, Iterative Procedures for Nonlinear Integral Equations, *J. Assoc. Comput. Machinery*, 12:547-560, 1965.

[AP1998] U.M Ascher and L.R. Petzold, Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations, SIAM, Philadelphia, 1998.

[B1985] Bank et al., Transient Simulation of Silicon Devices and Circuits, *IEEE Trans. CAD*, 4:436-451, 1985.

[B1983] S.R. Billington, Type-Insensitive Codes for the Solution of Stiff and Nonstiff Systems of Ordinary Differential Equations, *in: Master Thesis, University of Manchester, United Kingdom*, 1983.

[BS1989] P. Bogacki and L.F. Shampine. A 3(2) pair of Runge–Kutta formulas, *Appl. Math. Lett.*, 2:321–325, 1989.

[B1987] P.N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24:407-434, 1987.

[BBH1989] P.N. Brown, G.D. Byrne and A.C. Hindmarsh. VODE, a Variable-Coefficient ODE Solver. *SIAM J. Sci. Stat. Comput.*, 10:1038-1051, 1989.

[BH1989] P.N. Brown and A.C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49-91, 1989.

[BS1990] P.N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450-481, 1990.

[B2008] J.C. Butcher, Numerical Methods for Ordinary Differential Equations. Wiley, 2nd edition, Chicester, England, 2008.

[B1992] G.D. Byrne. Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting. In J.R. Cash and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pp. 323-356, Oxford University Press, 1992.

[C1979] J.R. Cash. Diagonally Implicit Runge-Kutta Formulae with Error Estimates. *IMA J Appl Math*, 24:293-301, 1979.

[CK1990] J.R. Cash and A.H. Karp. A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides, *ACM Trans. Math. Soft.*, 16:201-222, 1990.

[CGM2014] J. Cheng, M. Grossman and T. McKercher. Professional Cuda C Programming. John Wiley & Sons, 2014.

[CUDA] NVIDIA CUDA Programming Guide.

[cuSOLVER] NVIDIA cuSOLVER Documentation.

[cuSPARSE] NVIDIA cuSPARSE Documentation.

[DFWBT2010] M.R. Dorr, J.-L. Fattebert, M.E. Wickett, J.F. Belak and P.E.A Turchi. A numerical algorithm for the solution of a phase-field model of polycrystalline materials. *J. Comput. Phys.*, 229(3):626-641, 2010.

[DP1980] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae, *J. Comput. Appl. Math.* 6:19–26, 1980.

[DP2010] T. Davis and E. Palamadai Natarajan. Algortithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Soft.*, 37, 2010.

[DES1982] R.S. Dembo, S.C. Eisenstat and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400-408, 1982.

[DGL1999] J.W. Demmel, J.R. Gilbert and X.S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Analysis and Applications*, 20:915-952, 1999.

[DS1996] J.E. Dennis and R.B. Schnabel. Numerical Methods for Unconstrained Optimization and Nonlinear Equations. SIAM, Philadelphia, 1996.

[F2014] R.D. Falgout, S. Friedhoff, TZ.V. Kolev, S.P. MacLachlan, and J.B. Schroder, Parallel Time Integration with Multigrid, *SIAM J. Sci. Comput.*, 36:C635-C661, 2014.

[F2015] R. Falgout and U.M. Yang. Hypre user's manual. *LLNL Technical Report*, 2015.

[FS2009] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197-21, 2009.

[F1969] E. Fehlberg. Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems. *NASA Technical Report 315*, 1969.

[F1993] R.W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470-482, 1993.

[G1991] K. Gustafsson. Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods. *ACM Trans. Math. Soft.*, 17:533-554, 1991.

[G1994] K. Gustafsson. Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods. *ACM Trans. Math. Soft.* 20:496-512, 1994.

[GDL2007] L. Grigori, J.W. Demmel, and X.S. Li. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM J. Scientific Comptuing*, 29:1289-1314, 2007.

[HW1993] E. Hairer, S. Norsett and G. Wanner. Solving Ordinary Differential Equations I. *Springer Series in Computational Mathematics*, vol. 8, 1993.

[HW1996] E. Hairer and G. Wanner. Solving Ordinary Differential Equations II. *Springer Series in Computational Mathematics*, vol. 14, 1996.

[HS1952] M.R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Research of the National Bureau of Standards*, 49:409-436, 1952.

[HS1980] K.L. Hiebert and L.F. Shampine. Implicitly Defined Output Points for Solutions of ODEs. Technical Report SAND80-0180, Sandia National Laboratories, February 1980.

[H2000] A.C. Hindmarsh. The PVODE and IDA Algorithms. Technical Report UCRL-ID-141558, LLNL, 2000.

[HS2017] A.C. Hindmarsh and R. Serban. User Documentation for CVODE v5.6.0. Technical Report UCRL-SM-208108, LLNL, 2020.

[HSR2017] A.C. Hindmarsh, R. Serban and D.R. Reynolds. Example Programs for CVODE v5.6.0. Technical Report UCRL-SM-208110, LLNL, 2020.

[HT1998] A.C. Hindmarsh and A.G. Taylor. PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems. Technical Report UCRL-IL-129739, LLNL, February 1998.

[HK2014] R.D. Hornung and J.A. Keasler. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, LLNL, September 2014.

[JPE2019] S.R. Johnson, A. Prokopenko, and K. J. Evans. Automated Fortran-C++ bindings for Large-Scale Scientific Applications. arXiv:1904.02546 [cs], Apr. 2019.

[K1995] C.T. Kelley. Iterative Methods for Solving Linear and Nonlinear Equations. SIAM, Philadelphia, 1995.

[KC2003] C.A. Kennedy and M.H. Carpenter. Additive Runge-Kutta schemes for convection-diffusion-reaction equations. *Appl. Numer. Math.*, 44:139-181, 2003.

[KC2019] C.A. Kennedy and M.H. Carpenter. Higher-order additive Runge–Kutta schemes for ordinary differential equations. *Appl. Numer. Math.*, 136:183-205, 2019.

[K2004] A. Kv{ae}rno. Singly Diagonally Implicit Runge-Kutta Methods with an Explicit First Stage. *BIT Numer. Math.*, 44:489-502, 2004.

[KLU] KLU Sparse Matrix Factorization Library.

[L2005] X.S. Li. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Soft.*, 31:302-325, 2005.

[LD2003] X.S. Li. and J.W. Demmel. A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Soft.*, 29:110-140, 2003.

[LWWY2012] P.A. Lott, H.F. Walker, C.S. Woodward and U.M. Yang. An Accelerated Picard Method for Nonlinear Systems Related to Variably Saturated Flow, *Adv. Wat. Resour.*, 38:92-101, 2012.

[oneAPI] Intel oneAPI Programming Guide.

[R2018] D.R. Reynolds. ARKode Example Documentation. Technical Report, Southern Methodist University Center for Scientific Computation, 2020.

[ROCm] AMD ROCm Documentation.

[SS1986] Y. Saad and M.H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856-869, 1986.

[S1993] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14:461-469, 1993.

[S2019] A. Sandu, A Class of Multirate Infinitesimal GARK Methods. *SIAM J. Numer. Anal.*, 57:2300-2327, 2019.

[SA2002] A. Sayfy and A. Aburub. Embedded Additive Runge-Kutta Methods. *Intern. J. Computer Math.*, 79:945-953, 2002.

[SKAW2009] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Multirate Runge–Kutta schemes for advection equations. *J. Comput. Appl. Math.*, 226:345-357, 2009.

[SKAW2012a] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Implementation of multirate time integration methods for air pollution modelling. *GMD*, 5:1395-1405, 2012.

[SKAW2012b] M. Schlegel, O. Knoth, M. Arnold, and R. Wolke. Numerical solution of multiscale problems in atmospheric modeling. *Appl. Numer. Math.*, 62:1531-1542, 2012.

[S1998] G. Soderlind. The automatic control of numerical integration. *CWI Quarterly*, 11:55-74, 1998.

[S2003] G. Soderlind. Digital filters in adaptive time-stepping. *ACM Trans. Math. Soft.*, 29:1-26, 2003.

[S2006] G. Soderlind. Time-step selection algorithms: Adaptivity, control and signal processing. *Appl. Numer. Math.*, 56:488-502, 2006.

[SLUUG1999] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, M. Shao and I. Yamazaki. SuperLU Users' Guide. 1999.

[SuperLUDIST] SuperLU_DIST Parallel Sparse Matrix Factorization Library.

[SuperLUMT] SuperLU_MT Threaded Sparse Matrix Factorization Library.

[V1992] H.A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631-644, 1992.

[V1978] J.H. Verner. Explicit Runge-Kutta methods with estimates of the local truncation error. *SIAM J. Numer. Anal.*, 15:772-790, 1978.

[WN2011] H.F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM J. Numer. Anal.*, 49:1715-1735, 2011.

[KW1998] O. Knoth and R. Wolke. Implicit-explicit Runge-Kutta methods for computing atmospheric reactive flows. *Appl. Numer. Math.*, 28(2):327-341, 1998.

[XBraid] XBraid: Parallel multigrid in time..

[Z1963] J.A. Zonneveld. Automatic integration of ordinary differential equations. *Report R743, Mathematisch Centrum*, Postbus 4079, 1009AB Amsterdam, 1963.

Index