User Documentation for ARKode v2.2.1 (SUNDIALS v3.2.1)

Daniel R. Reynolds[1]**, David J. Gardner**[2]**,
Alan C. Hindmarsh**[2]**, Carol S. Woodward**[2]
and Jean M. Sexton[1]**,**

[1]*Department of Mathematics
Southern Methodist University*

[2]*Center for Applied Scientific Computing
Lawrence Livermore National Laboratory*

October 11, 2018

DISCLAIMER

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). 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 and using either shared-memory parallelism (via OpenMP or Pthreads) or distributed-memory parallelism (via MPI). The default integration and solver options should apply to most users, though complete control over all internal parameters and time adaptivity algorithms is enabled through optional interface routines.

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

Due to its similarities in both function and design with the CVODE package, this documentation is highly similar with the corresponding CVODE user guide *[HS2017]*.

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.

INTRODUCTION

The ARKode solver library provides an adaptive-step time integration package for stiff, nonstiff and mixed stiff/nonstiff systems of ordinary differential equations (ODEs) given in explicit 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 $\mathbb{R}^N$ (possibly time dependent, but independent of y), and the right-hand side function is partitioned into two components:

- $f_E(t, y)$ contains the "slow" time scale components to be integrated explicitly, and

- $f_I(t, y)$ contains the "fast" 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 methods used in ARKode are adaptive-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 system into explicit and implicit components (1.1), such methods have the potential to enable accurate and efficient time integration of 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 solvers.

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.

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 Newton method, modified Newton method, an 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 one of the distributed parallel NVECTOR modules, including PETSc and *hypre* vectors, 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, there is a banded preconditioner module called ARKBANDPRE for use with the Krylov solvers, while for the distributed memory 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 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.

1.1.2 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 deprecated. 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., `LAPACK_ENABLE` 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.

1.1.3 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`.

1.1.4 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.).

1.1.5 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.

1.1.6 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 SUNLIN-EARSOLVER/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 th 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.

1.1.7 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 teh 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* that relate these algorithms together.

- We follow this with overview of how the source code for ARKode is *organized*.

- The largest section follows, providing a full account of the ARKode user interface, including a description of all user-accessible functions and outlines for ARKode usage for serial and parallel applications. Since ARKode is written in C, we first present *the C and C++ interface*, followed with a separate section on *using ARKode within Fortran applications*.

- The following 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

The SUNDIALS packages are released open source, under a BSD license. The only requirements of the BSD license are preservation of copyright and a standard disclaimer of liability. Our Copyright notice is below along with the license.

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 LGPL license (which is what KLU is released with) and *not* the SUNDIALS BSD license anymore.

1.3.1 Copyright Notices

All SUNDIALS packages except ARKode are subject to the following Copyright notice.

SUNDIALS Copyright

Copyright (c) 2002-2016, Lawrence Livermore National Security. Produced at the Lawrence Livermore National Laboratory. Written by A.C. Hindmarsh, D.R. Reynolds, R. Serban, C.S. Woodward, S.D. Cohen, A.G. Taylor, S. Peles, L.E. Banks, and D. Shumaker.

UCRL-CODE-155951 (CVODE)

UCRL-CODE-155950 (CVODES)

UCRL-CODE-155952 (IDA)

UCRL-CODE-237203 (IDAS)

LLNL-CODE-665877 (KINSOL)

All rights reserved.

ARKode Copyright

ARKode is subject to the following joint Copyright notice. Copyright (c) 2015-2017, Southern Methodist University and Lawrence Livermore National Security Written by D.R. Reynolds, D.J. Gardner, A.C. Hindmarsh, C.S. Woodward, and J.M. Sexton.

LLNL-CODE-667205 (ARKODE)

All rights reserved.

1.3.2 BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.

3. Neither the name of the LLNS/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

- This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.

- Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

- Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

MATHEMATICAL CONSIDERATIONS

ARKode solves ODE initial value problems (IVPs) in $\mathbb{R}^N$. These problems should be posed in explicit form, as

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

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 $\frac{dy}{dt}$.

M is a user-specified nonsingular operator from $\mathbb{R}^N \to \mathbb{R}^N$. This operator may depend on t but is currently assumed to be 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 or finite volume methods, M is typically the identity matrix, I. For PDEs using a finite-element spatial semi-discretization M is typically a well-conditioned mass matrix.

The two right-hand side functions may be described as:

- $f_E(t, y)$ contains the "slow" time scale components of the system. This will be integrated using explicit methods.

- $f_I(t, y)$ contains the "fast" time scale components of the system. This will be integrated using implicit methods.

ARKode may be used to solve stiff, nonstiff and mixed stiff/nonstiff problems. Roughly speaking, stiffness is characterized by the presence of at least one rapidly damped mode, whose time constant is small compared to the time scale of the solution itself. In the implicit/explicit (ImEx) splitting above, these stiff components should be included in the right-hand side function $f_I(t, y)$.

In the sub-sections that follow, we elaborate on the numerical methods that comprise the ARKode solvers. We first discuss the general *formulation of additive Runge-Kutta methods*, including the resulting implicit systems that must be solved at each stage. We then discuss the solver strategies that ARKode uses in solving these systems: *nonlinear solvers*, *linear solvers* and *preconditioners*. We then describe our approaches for *error control* within the iterative nonlinear and linear solvers, including discussion on our choice of norms used within ARKode for measuring errors within various components of the solver. We then discuss specific enhancements available in ARKode, including an array of *prediction algorithms* for the solution at each stage, *adaptive error controllers*, *mass-matrix handling*, and *rootfinding capabilities*.

2.1 Additive Runge-Kutta methods

The methods used in ARKode are variable-step, embedded, additive Runge-Kutta methods (ARK), based on formulas of the form

$$
Mz_i = My_{n-1} + h_n \sum_{j=1}^{i-1} A_{i,j}^E f_E(t_{n,j}^E, z_j) + h_n \sum_{j=1}^{i} A_{i,j}^I f_I(t_{n,j}^I, z_j), \quad i = 1, \ldots, s,
$$

$$
My_n = My_{n-1} + h_n \sum_{i=1}^{s} \left(b_i^E f_E(t_{n,i}^E, z_i) + b_i^I f_I(t_{n,i}^I, z_i) \right), \tag{2.2}
$$

$$
M\tilde{y}_n = My_{n-1} + h_n \sum_{i=1}^{s} \left(\tilde{b}_i^E f_E(t_{n,i}^E, z_i) + \tilde{b}_i^I f_I(t_{n,i}^I, z_i) \right).
$$

Here the y_n are computed approximations to $y(t_n)$, $\tilde{y}_n$ are [typically] lower-order embedded solutions (used in error estimation), and $h_n \equiv t_n - t_{n-1}$ is the step size. The internal stage times are abbreviated using the notation $t_{n,j}^E = t_{n-1} + c_j^E h_n$ and $t_{n,j}^I = t_{n-1} + c_j^I h_n$. 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 $\tilde{b}^I \in \mathbb{R}^s$ may be used to enable an *embedded solution* that is used to estimate error for adaptive time-stepping. We note that ARKode currently enforces the constraint that these tables must share the same number of stages s between the explicit and implicit methods in an ARK pair.

The user of ARKode must choose appropriately between one of three classes of methods: *ImEx*, *explicit* and *implicit*. All of ARKode's available 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, ARKode currently implements the ARK methods proposed in *[KC2003]*, allowing for methods having order $q = \{3, 4, 5\}$. The tables for these methods are given in the section *Additive Butcher tables*.

For nonstiff problems, a user may specify that $f_I = 0$, i.e. the equation (2.1) reduces to the non-split IVP

$$
M\dot{y} = f_E(t, y), \qquad y(t_0) = y_0. \tag{2.3}
$$

In this scenario, the coefficients $A^I = 0$, $c^I = 0$, $b^I = 0$ and $\tilde{b}^I = 0$ in (2.2), and the ARK methods reduce to classical explicit Runge-Kutta methods (ERK). For these classes of methods, ARKode allows 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.

Finally, for stiff problems the user may specify that $f_E = 0$, so the equation (2.1) reduces to the non-split IVP

$$
M\dot{y} = f_I(t, y), \qquad y(t_0) = y_0. \tag{2.4}
$$

Similarly to ERK methods, in this scenario the coefficients $A^E = 0$, $c^E = 0$, $b^E = 0$ and $\tilde{b}^E = 0$ in (2.2), and the ARK methods reduce to classical diagonally-implicit Runge-Kutta methods (DIRK). For these classes of methods, ARKode allows 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.

2.2 Nonlinear solver methods

For both the DIRK and ARK methods corresponding to (2.1) and (2.4), an implicit system

$$
G(z_i) \equiv Mz_i - h_n A_{i,i}^I f_I(t_{n,i}^I, z_i) - a_i = 0 \tag{2.5}
$$

must be solved for each stage $z_i, i = 1, \ldots, s$, where we have the data

$$a_i \equiv My_{n-1} + h_n \sum_{j=1}^{i-1} \left[A_{i,j}^E f_E(t_{n,j}^E, z_j) + A_{i,j}^I f_I(t_{n,j}^I, z_j) \right]$$

for the ARK methods, or

$$a_i \equiv My_{n-1} + h_n \sum_{j=1}^{i-1} A_{i,j}^I f_I(t_{n,j}^I, z_j)$$

for the DIRK methods. Here, if $f_I(t, y)$ depends nonlinearly on y then (2.5) corresponds to a nonlinear system of equations; if $f_I(t, y)$ depends linearly on y then this is a linear system of equations.

For systems of either type, ARKode allows a choice of solution strategy. The default solver choice is a variant of Newton's method,

$$z_i^{(m+1)} = z_i^{(m)} + \delta^{(m+1)}, \tag{2.6}$$

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

$$\mathcal{A}\left(z_i^{(m)}\right) \delta^{(m+1)} = -G\left(z_i^{(m)}\right), \tag{2.7}$$

in which

$$\mathcal{A} \approx M - \gamma J, \quad J = \frac{\partial f_I}{\partial y}, \quad \text{and} \quad \gamma = h_n A_{i,i}^I. \tag{2.8}$$

As an alternate to Newton's method, ARKode may solve for each stage $z_i, i = 1, \ldots, s$ using an Anderson-accelerated fixed point iteration

$$z_i^{(m+1)} = g(z_i^{(m)}), \quad m = 0, 1, \ldots \tag{2.9}$$

Unlike with Newton's method, this method *does not* require the solution of a linear system at each iteration, instead opting for solution of a low-dimensional least-squares solution to construct the nonlinear update. For details on how this iteration is performed, we refer the reader to the reference *[WN2011]*.

Finally, if the user specifies that $f_I(t, y)$ depends linearly on y (via a call to *ARKodeSetLinear()* in C/C++, or the *LINEAR* argument to *FARKSETIIN()* in Fortran), and if the Newton-based nonlinear solver is chosen, then the problem (2.5) will be solved using only a single Newton iteration. In this case, an additional argument to the respective function denotes whether this Jacobian is time-dependent or not, indicating whether the Jacobian or preconditioner needs to be recomputed at each step.

The optimal solver (Newton vs fixed-point) is highly problem-dependent. Since fixed-point solvers do not require the solution of any linear systems, 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. However, these fixed-point solvers do allow for user specification of the Anderson-accelerated subspace size, m_k. While the required amount of solver memory grows proportionately to $m_k N$, larger values of m_k may result in faster convergence. In our experience, this improvement may be significant even for "small" values, e.g. $1 \leq m_k \leq 5$, and that convergence may not improve (or even deteriorate) for larger values of m_k.

While ARKode uses a Newton-based iteration as its default solver due to its increased robustness on very stiff problems, it is highly recommended that users also consider the fixed-point solver for their 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 depends on the choice of a good initial guess. In ARKode, the initial guess for either nonlinear solution method is a predicted value $z_i^{(0)}$ that is computed explicitly from the previously-computed data (e.g. y_{n-2}, y_{n-1}, and z_j where $j < i$). Additional information on the specific predictor algorithms implemented in ARKode is provided in the following section, *Implicit predictors*.

2.3 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]* (serial or threaded vector modules only) [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 Flexible GMRES (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 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 solvers provided by SUNDIALS (dense, band and sparse), as well as the direct linear solvers accessible through LAPACK, can only be used with the serial and threaded vector representations.

In the case that a direct linear solver is used, ARKode utilizes either a Newton or a *modified Newton iteration*. The difference between these is that in a modified Newton method, the matrix $\mathcal{A}$ is held fixed for multiple Newton iterations. More precisely, each Newton iteration is computed from the modified equation

$$\tilde{\mathcal{A}}\left(z_i^{(m)}\right)\delta^{(m+1)} = -G\left(z_i^{(m)}\right), \tag{2.10}$$

in which

$$\tilde{\mathcal{A}} \approx M - \tilde{\gamma}\tilde{J}, \quad \tilde{J} = \frac{\partial f_I}{\partial y}(\tilde{y}), \quad \text{and} \quad \tilde{\gamma} = \tilde{h}A_{i,i}^I. \tag{2.11}$$

Here, the solution $\tilde{y}$ and step size $\tilde{h}$ upon which the modified Jacobian rely, are merely values of the solution and step size from a previous iteration. In other words, the matrix $\tilde{\mathcal{A}}$ is only computed rarely, and reused for repeated stage solves. The frequency at which $\tilde{\mathcal{A}}$ is recomputed, and hence the choice between normal and modified Newton iterations, is determined by the input parameter *msbp* to the input function *ARKodeSetMaxStepsBetweenLSet()* in C/C++, or with the *LSETUP_MSBP* argument to *FARKSETIIN()* in Fortran.

When using the direct and band solvers for the linear systems (2.10), the Jacobian may be supplied by a user routine or approximated by finite-differences. In the case of differencing, we use the standard approximation

$$J_{i,j}(t,y) = \frac{f_{I,i}(t, y + \sigma_j e_j) - f_{I,i}(t,y)}{\sigma_j},$$

where e_j is the jth unit vector, and the increments σ_j are given by

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

Here U is the unit roundoff, σ_0 is a dimensionless value, and w_j is the error weight defined in (2.13). In the dense case, this approach requires N evaluations of f_I, 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_I evaluations equal to the bandwidth.

We note that with the sparse direct solvers, the Jacobian *must* be supplied by a user routine.

In the case that an iterative linear solver is chosen, ARKode utilizes a Newton method variant called an *Inexact Newton iteration*. 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 (2.7) is not solved completely, since these linear solvers are iterative (hence the "inexact" in the name). Resultingly. for these linear solvers $\mathcal{A}$ is applied in a matrix-free manner,

$$\mathcal{A}v = Mv - \gamma Jv.$$

The matrix-vector products Jv are obtained by either calling an optional user-supplied routine, or through directional differencing using the formula

$$Jv = \frac{f_I(t, y + \sigma v) - f_I(t, y)}{\sigma},$$

where the increment $\sigma = 1/\|v\|$ to ensure that $\|\sigma v\| = 1$.

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

Alternately, for some preconditioning algorithms that do not rely on costly matrix construction and factorization operations (e.g. when using an iterative multigrid method as preconditioner), a user may specify that $\mathcal{A}$ and/or P should be recomputed at every Newton iteration, since the increased rate of convergence may more than account for the additional cost of Jacobian/preconditioner construction. To indicate this, a user need only supply a negative value for the *msbp* argument to `ARKodeSetMaxStepsBetweenLSet()` in C/C++, or the *LSETUP_MSBP* argument to `FARKSETIIN()` in Fortran.

However, in cases where recomputation of the Newton matrix $\tilde{\mathcal{A}}$ or preconditioner matrix P is lagged, ARKode will force recomputation of these structures 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 changed via the *msbp* argument to `ARKodeSetMaxStepsBetweenLSet()` in C/C++, or the *LSETUP_MSBP* argument to `FARKSETIIN()` in Fortran),

- when the value $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| > 0.2$ (this tolerance may be changed via the *dgmax* argument to `ARKodeSetDeltaGammaMax()` in C/C++, or the *LSETUP_DGMAX* argument to `FARKSETRIN()` in Fortran),

- 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 reevaluation 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 reevaluate J (or instruct the user to reevaluate Jacobian data in 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 $\bar{\gamma}$ of γ at the last update satisfies $|\gamma/\bar{\gamma} - 1| > 0.2$,

- 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.

As will be further discussed in the section *Preconditioning*, in the case of a Krylov method, 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.

2.4 Iteration Error Control

2.4.1 Choice of norm

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

$$\|v\|_{\text{WRMS}} = \left(\frac{1}{N} \sum_{i=1}^{N} (v_i \, w_i)^2 \right)^{1/2}. \tag{2.12}$$

The power of this choice of norm arises in the specification of the weighting vector w, that combines the units of the problem with the user-supplied measure of "acceptable" error. To this end, ARKode constructs and error weight vector using the most-recent step solution and the relative and absolute tolerances input by the user, namely

$$w_i = \frac{1}{RTOL \cdot |y_i| + ATOL_i}. \tag{2.13}$$

Since $1/w_i$ represents a tolerance in the component y_i, a vector whose WRMS norm is 1 is regarded as "small." For brevity, we will typically 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 (2.1) may differ from the units of the solution y. In this case, ARKode may also construct a residual weight vector,

$$w_i = \frac{1}{RTOL \cdot |My_i| + ATOL'_i}, \tag{2.14}$$

where the user may specify a separate absolute residual tolerance value or array, $ATOL'_i$. The choice of weighting vector used in any given norm is determined by the quantity being measured: values having solution units use (2.13), whereas values having equation units use (2.14). Obviously, for problems with $M = I$, the weighting vectors are identical.

2.4.2 Nonlinear iteration error control

The stopping test for all of ARKode's nonlinear solvers is related to the subsequent 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 (2.5) 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 $\tilde{A}$ 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

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

where the factor 0.3 is user-modifiable as the *crdown* input to the the function *ARKodeSetNonlinCRDown()* in C/C++, or the *NONLIN_CRDOWN* argument to *FARKSETRIN()* in Fortran.

Denoting the true time step solution as $y(t_n)$, and the computed time step solution (computed using the stage solutions $z_i^{(m)}$) as y_n, we use the estimate

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

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

$$R_i \left\| \delta^{(m)} \right\| < \epsilon, \tag{2.15}$$

where the factor ϵ has default value 0.1, and is user-modifiable as the *nlscoef* input to the the function `ARKodeSetNonlinConvCoef()` in C/C++, or the *NLCONV_COEF* input to the function `FARKSETRIN()` in Fortran. We allow at most 3 nonlinear iterations (modifiable through `ARKodeSetMaxNonlinIters()` in C/C++, or as the *MAX_NSTEPS* argument to `FARKSETIIN()` in Fortran). We also declare the nonlinear iteration to be divergent if any of the ratios $\|\delta^{(m)}\|/\|\delta^{(m-1)}\| > 2.3$ with $m > 0$ (the value 2.3 may be modified as the *rdiv* input to `ARKodeSetNonlinRDiv()` in C/C++, or the *NONLIN_RDIV* input to `FARKSETRIN()` in Fortran). If convergence fails in the fixed point iteration, or in the Newton iteration with J or $\mathcal{A}$ current, we must then reduce the step size by a factor of 0.25 (modifiable via the *etacf* input to the `ARKodeSetMaxCFailGrowth()` function in C/C++, or the *ADAPT_ETACF* input to `FARKSETRIN()` in Fortran). The integration is halted after 10 convergence failures (modifiable via the `ARKodeSetMaxConvFails()` function in C/C++, or the *MAX_CONVFAIL* argument to `FARKSETIIN()` in Fortran).

2.4.3 Linear iteration error control

When a Krylov method is used to solve the linear systems (2.7), 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 on 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

$$\|r\| \leq \frac{\epsilon_L \epsilon}{10}. \tag{2.16}$$

Here ϵ is the same value as that 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 and stiff ODE systems, while easier ODE systems may benefit from a value closer to 1; by default $\epsilon_L = 0.05$, which may be modified by the user through the `ARKSpilsSetEpsLin()` function in C/C++, or through the `FARKSPILSSETEPSLIN()` in Fortran. We note that for linearly implicit problems the same tolerance (2.16) is used for the single Newton iteration.

2.5 Preconditioning

When using an inexact Newton method to solve the nonlinear system (2.5), ARKode makes repeated use of a linear solver 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 linear system solve is done with one of the scaled preconditioned iterative linear solvers, the efficiency of such solvers may benefit tremendously from preconditioning. A system $\mathcal{A}x = b$ can be preconditioned as one of:

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

The Krylov method is 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 $\mathcal{A}$. Yet at the same time, 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 tradeoff 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 preconditioning either side, or on both sides, 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, and the user should experiment with both choices. Performance will differ between these choices because 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 used with ARKode 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 feature(s) 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.

2.6 Implicit predictors

For problems with implicit components, ARKode will employ a prediction algorithm for constructing the initial guesses for each 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 nonlinear solve, enabling the difference between rapid quadratic convergence versus divergence of the iteration. To this end, ARKode implements a variety of prediction algorithms that may be selected by the user. 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 subsequent stage times to provide an inexpensive but (hopefully) reasonable prediction of the subsequent solution value. Specifically, for most Runge-Kutta methods each stage solution satisfies

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

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

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

Denoting $[a, b]$ as the interval containing the data used to construct $p_q(t)$, and assuming forward integration from $a \to b$, it is typically the case that $t_{n,j}^I > b$. 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 various prediction algorithms therefore construct different types of interpolant $p_q(t)$, as described below.

2.6.1 Trivial predictor

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

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

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 either highly stiff problems, or problems with implicit constraints whose violation may cause illegal solution values (e.g. a negative density or temperature).

2.6.2 Maximum order predictor

At the opposite end of the spectrum, ARKode can construct an interpolant $p_q(t)$ of polynomial order up to $q = 3$. Here, the function $p_q(t)$ is identical to the one used for interpolation of output solution values between time steps, i.e. for "dense output" of $y(t)$ for $t_{n-1} < t < t_n$. The order of this polynomial, q, may be specified by the user with the function *ARKodeSetDenseOrder()* in C/C++, or with the *DENSE_ORDER* argument to *FARKSETIIN()* in Fortran.

The interpolants generated are either of Lagrange or Hermite form, and use the data $\{y_{n-2}, f_{n-2}, y_{n-1}, f_{n-1}\}$, where we use f_k to denote $M^{-1}(f_E(t_k, y_k) + f_I(t_k, y_k))$. Defining a scaled and shifted "time" variable τ for the interval $[t_{n-2}, t_{n-1}]$ as

$$\tau(t) = (t - t_{n-1})/h_{n-1},$$

we may denote the predicted stage times in the subsequent time interval $[t_{n-1}, t_n]$ as

$$\tau_i = c_i^I \frac{h_n}{h_{n-1}}.$$

We then construct the interpolants $p(t)$ as follows:

- $q = 0$: constant interpolant

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

- $q = 1$: linear Lagrange interpolant

$$p_1(\tau) = -\tau \, y_{n-2} + (1 + \tau) \, y_{n-1}.$$

- $q = 2$: quadratic Hermite interpolant

$$p_2(\tau) = \tau^2 \, y_{n-2} + (1 - \tau^2) \, y_{n-1} + h(\tau + \tau^2) \, f_{n-1}.$$

- $q = 3$: cubic Hermite interpolant

$$p_3(\tau) = (3\tau^2 + 2\tau^3) \, y_{n-2} + (1 - 3\tau^2 - 2\tau^3) \, y_{n-1} + h(\tau^2 + \tau^3) \, f_{n-2} + h(\tau + 2\tau^2 + \tau^3) \, f_{n-1}.$$

These higher-order predictors may be useful when using lower-order methods in which h_n is not too large. We further note that although interpolants of order > 3 are possible, these are not implemented due to their increased computing and storage costs, along with their diminishing returns due to increased extrapolation error.

2.6.3 Variable order predictor

This predictor attempts to use higher-order interpolations $p_q(t)$ for predicting earlier stages in the subsequent time interval, and lower-order interpolants for later stages. It uses the same formulas as described above, but chooses q adaptively based on the stage index i, under the (rather tenuous) assumption that the stage times are increasing, i.e. $c_j^I < c_k^I$ for $j < k$:

$$q = \max\{q_{\max} - i, \, 1\}.$$

2.6.4 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:

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

2.6.5 Bootstrap predictor

This predictor does not use any information from within the preceding step, instead using information only within the current step $[t_{n-1}, t_n]$ (including y_{n-1} and f_{n-1}). Instead, 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 in which $c_j^I = 0$ for all previous stages $j = 0, \ldots, i-1$, and for the first stage of any time step ($i = 0$), this method reduces to using the trivial predictor $z_i^{(0)} = y_{n-1}$. For stages having multiple precdding nonzero c_j^I, we choose the stage having largest c_j^I value, to minimize the amount of extrapolation induced through the prediction.

We note that in general, each stage solution z_j has signicantly worse accuracy than the time step solutions y_{n-1}, due to the difference between the *stage order* and the *method order* in typical 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 $[t_{n-2}, t_{n-1}]$.

We further note that although this method could be used with non-identity mass matrix $M \neq I$, support for that mode is not currently implemented, so selection of this predictor in the case that $M \neq I$ will result in use of the *Trivial predictor*.

2.6.6 Minimum correction predictor

This 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 (2.2) and (2.5), 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

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

$$\Leftrightarrow$$

$$z_i = a_i.$$

We note that although this method could be also used with non-identity mass matrix $M \neq I$, support for that mode is not currently implemented, so selection of this predictor in the case that $M \neq I$ will result in use of the *Trivial predictor*.

2.7 Time step adaptivity

A critical component of ARKode, making it an IVP "solver" rather than just an integrator, is its adaptive control of local truncation error. 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, every Runge-Kutta method packaged within ARKode admit an embedded solution $\tilde{y}_n$, as shown in equation (2.2). Generally, these embedded solutions attain a lower order of accuracy than the computed solution y_n. Denoting these orders of accuracy as p and q, where p corresponds to the embedding and q corresponds to the method, for the majority of embedded methods $p = q - 1$. These values of p and q correspond to the global order of accuracy for the method and embedding, hence each admit local errors satisfying *[HW1993]*

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

(2.17)

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

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

We therefore use this difference norm as an estimate for the local truncation error at the step n,

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

(2.18)

Here, $\beta > 0$ is an error *bias* to help account for the error constant D; the default value of this is $\beta = 1.5$, and may be modified by the user through the function `ARKodeSetErrorBias()` in C/C++, or through the input *ADAPT_BIAS* to `FARKSETRIN()` in Fortran.

With this LTE estimate, the local error test is simply $\|T_n\| < 1$, where we remind that this norm includes the user-specified relative and absolute tolerances. If this error test passes, the step is considered successful, and the estimate is subsequently used to estimate the next step size, as will be 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 error control algorithms described in *Asymptotic error control*. A new attempt at the step is made, and the error test is repeated. If it fails multiple times (as specified through the *small_nef* input to `ARKodeSetSmallNumEFails()` in C/C++, or the *ADAPT_SMALL_NEF* argument to `FARKSETIIN()` in Fortran, which defaults to 2), then h'/h is limited above to 0.3 (this is modifiable via the *etamxf* argument to `ARKodeSetMaxEFailGrowth()` in C/C++, or the *ADAPT_ETAMXF* argument to `FARKSETRIN()` in Fortran), and limited below to 0.1 after an additional step failure. After seven error test failures (modifiable via the function `ARKodeSetMaxErrTestFails()` in C/C++, or the *MAX_ERRFAIL* argument to `FARKSETIIN()` in Fortran), ARKode returns to the user with a give-up message.

We define the step size ratio between a prospective step h' and a completed step h as η, i.e.

$$\eta = h'/h.$$

This is bounded above by $\eta_{\max}$ to ensure that step size adjustments are not overly aggressive. This value is modified according to the step and history,

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

Here, the values of *etamx1* and *growth* may be modified by the user in the functions `ARKodeSetMaxFirstGrowth()` and `ARKodeSetMaxGrowth()` in C/C++, respectively, or through the inputs *ADAPT_ETAMX1* and *ADAPT_GROWTH* to the function `FARKSETRIN()` in Fortran.

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.
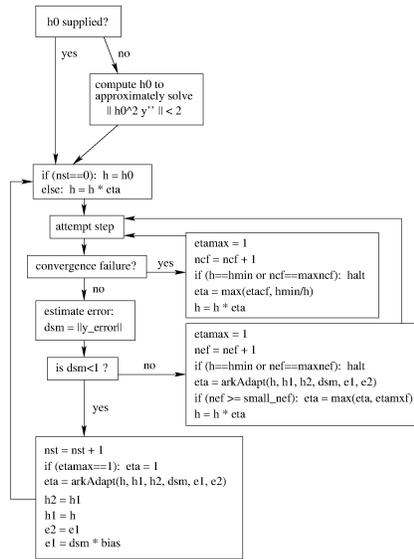
For some problems it may be preferrable to avoid small step size adjustments. This can be especially true for problems that construct and factor the Newton Jacobian matrix $\mathcal{A}$ from equation (2.8) for either a direct solve, or as a preconditioner for an iterative solve, where this construction is computationally expensive, and where Newton convergence can be seriously hindered through use of a somewhat incorrect $\mathcal{A}$. In these scenarios, the step is not changed when $\eta \in [\eta_L, \eta_U]$. The default values for these parameters are $\eta_L = 1$ and $\eta_U = 1.5$, though these are modifiable through the function *ARKodeSetFixedStepBounds()* in C/C++, or through the input *ADAPT_BOUNDS* to the function *FARKSETRIN()* in Fortran.

The user may supply external bounds on the step sizes within ARKode, through defining the values $h_{\min}$ and $h_{\max}$ with the functions *ARKodeSetMinStep()* and *ARKodeSetMaxStep()* in C/C++, or through the inputs *MIN_STEP* and *MAX_STEP* to the function *FARKSETRIN()* in Fortran, respectively. These default to $h_{\min} = 0$ and $h_{\max} = \infty$.

Normally, ARKode takes steps until a user-defined output value $t = t_{\text{out}}$ is overtaken, and then it computes $y(t_{\text{out}})$ by interpolation (using the same dense output routines described in the section *Maximum order predictor*). However, a "one step" mode option is available, where control returns to the calling program after each step. There are also options to force ARKode not to integrate past a given stopping point $t = t_{\text{stop}}$, through the function *ARKodeSetStopTime()* in C/C++, or through the input *STOP_TIME* to *FARKSETRIN()* in Fortran.

2.7.1 Asymptotic error control

As mentioned above, ARKode adapts 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 1.0 upon program initialization, to accomodate the few initial time steps of a calculation where some of these error estimates are undefined. With these estimates, ARKode implements a variety of error control algorithms, as specified in the subsections below.

PID controller

This is the default time adaptivity controller used by ARKode. It derives from those found in *[KC2003]*, *[S1998]*, *[S2003]* and *[S2006]*. It 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, though each may be changed via a call to the C/C++ function *ARKodeSetAdaptivityMethod()* in C/C++, or to the Fortran function *FARKSETADAPTIVITYMETHOD()* in Fortran. In this estimate, a floor of $\varepsilon > 10^{-10}$ is enforced to avoid division-by-zero errors.

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 via a call to *ARKodeSetAdaptivityMethod()* in C/C++, or *FARKSETADAPTIVITYMETHOD()* in Fortran. As with the previous controller, at initialization $k_1 = k_2 = 1.0$ and the floor of 10^{-10} is enforced on the local error estimates.

I controller

The so-called I controller is the standard time adaptivity control algorithm in use by most available ODE solvers. 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 overridden by the user with the function *ARKodeSetAdaptivityMethod()* in C/C++, or the function *FARKSETADAPTIVITYMETHOD()* in Fortran.

Explicit Gustafsson controller

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

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

The default values of k_1 and k_2 are 0.367 and 0.268, respectively, which may be changed by calling either *ARKodeSetAdaptivityMethod()* in C/C++, or *FARKSETADAPTIVITYMETHOD()* in Fortran.

Implicit Gustafsson controller

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

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

The algorithm parameters default to $k_1 = 0.98$ and $k_2 = 0.95$, but may be modified by the user with *ARKodeSetAdaptivityMethod()* in C/C++, or *FARKSETADAPTIVITYMETHOD()* in Fortran.

ImEx Gustafsson controller

An ImEx version of these two preceding controllers is available in ARKode. 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 three of these parameters may be modified with the C/C++ function `ARKodeSetAdaptivityMethod()` in C/C++, or the Fortran function `FARKSETADAPTIVITYMETHOD()` in Fortran.

User-supplied controller

Finally, ARKode allows the user to define their own time step adaptivity function,

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

via a call to the C/C++ routine `ARKodeSetAdaptivityFn()` or the Fortran routine `FARKADAPTSET()`.

2.8 Explicit stability

For problems that involve a nonzero explicit component, $f_E(t, y) \neq 0$, explicit and ImEx Runge-Kutta methods may benefit from addition user-supplied information regarding the explicit stability region. All ARKode adaptivity methods utilize estimates of the local error. 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 may be costlier than desired. In these scenarios, a stability-based time step controller may also be useful.

Since the explicit stability region for any method depends on the problem under consideration, as it results from the eigenvalues of the linearized operator $\frac{\partial f_E}{\partial y}$, information on the maximum stable step size is not computed internally within ARKode. However, for many problems such information is readily available. For example, 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,

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

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, $|h_{\text{exp}}|$, by calling the C/C++ function `ARKodeSetStabilityFn()` or the Fortran function `FARKEXPSTABSET()`. 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 step used by ARKode will satisfy

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

Here the explicit stability step factor (often called the "CFL factor") $c > 0$ may be modified through the function `ARKodeSetCFLFraction()` in C/C++, or through the input *ADAPT_CFL* to the function `FARKSETRIN()` in Fortran, and has a default value of $1/2$.

2.8.1 Fixed time stepping

While ARKode is designed for time step adaptivity, it may additionally be called in "fixed-step" mode, typically used for debugging purposes or for verification against hand-coded Runge-Kutta methods. In this mode, all time step adaptivity is disabled:

- temporal error control is disabled,

- nonlinear or linear solver non-convergence results 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 section *Optional input functions*.

2.9 Mass matrix solver

Within the algorithms described above, there are three locations where a linear solve of the form

$$Mx = b$$

is required: (a) in constructing the time-evolved solution y_n, (b) in estimating the local temporal truncation error, and (c) in constructing predictors for the implicit solver iteration (see section *Maximum order predictor*). Specifically, to construct the time-evolved solution y_n from equation (2.2) we must solve

$$My_n = My_{n-1} + h_n \sum_{i=1}^{s} \left(b_i^E f_E(t_{n,i}^E, z_i) + b_i^I f_I(t_{n,i}^I, z_i) \right),$$

$$\Leftrightarrow$$

$$M(y_n - y_{n-1}) = h_n \sum_{i=1}^{s} \left(b_i^E f_E(t_{n,i}^E, z_i) + b_i^I f_I(t_{n,i}^I, z_i) \right),$$

$$\Leftrightarrow$$

$$M\nu = h_n \sum_{i=1}^{s} \left(b_i^E f_E(t_{n,i}^E, z_i) + b_i^I f_I(t_{n,i}^I, z_i) \right),$$

for the update $\nu = y_n - y_{n-1}$. Similarly, in computing the local temporal error estimate T_n from equation (2.18) we must solve systems of the form

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

Lastly, in constructing dense output and implicit predictors of order 2 or higher (as in the section *Maximum order predictor* above), we must compute the derivative information f_k from the equation

$$M f_k = f_E(t_k, y_k) + f_I(t_k, y_k).$$

Of course, for problems in which $M = I$ these solves are not required; however for problems with non-identity M, ARKode may use either an iterative linear solver or a direct linear solver, in the same manner as described in the section *Linear solver methods* for solving the linear Newton systems. We note that at present, the matrix M may depend on time t but must be independent of the solution y, since we assume that each of the above systems are linear.

At present, for DIRK and ARK problems using a direct solver for the Newton nonlinear iterations, the type of matrix (dense, band or sparse) for the Newton systems $\mathcal{A}\delta = -G$ must match the type of linear solver used for these mass-matrix systems, since M is included inside $\mathcal{A}$. When direct methods are employed, the user must supply a routine to compute M in either dense, band or sparse form to match the structure of $\mathcal{A}$, with a user-supplied routine of type `ARKDlsMassFn()`.

When iterative methods are used, a routine must be supplied to perform the mass-matrix-vector product, Mv, through a call to the routine *ARKSpilsMassTimesVecFn()*. As with iterative solvers for the Newton systems, preconditioning may be applied to aid in solution of the mass matrix systems $Mx = b$. When using an iterative mass matrix linear solver, we require that the norm of the preconditioned linear residual satisfies

$$\|r\| \leq \epsilon_L \epsilon, \tag{2.21}$$

where again, ϵ is the nonlinear solver tolerance parameter from (2.15). When using iterative system and mass matrix linear solvers, ϵ_L may be specified separately for both tolerances (2.16) and (2.21); the mass matrix linear solver value of ϵ_L may be modified using *ARKSpilsSetMassEpsLin()* in C/C++, or *FARKSPILSSETMASSEPSLIN()* in Fortran.

2.10 Rootfinding

The ARKode solver has been augmented to include a rootfinding feature. This means that, while integrating the IVP (2.1), ARKode 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 probably be missed by ARKode. 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 computed, 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 $(t_{lo}, t_{hi}]$ 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 $g_i(t_{lo}) \neq 0$. 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 $g(t_{hi})$ for zeros, and it checks for sign changes in (t_{lo}, t_{hi}). If no sign changes are found, then either a root is reported (if some $g_i(t_{hi}) = 0$) 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 $|g_i(t_{hi})| / |g_i(t_{hi}) - g_i(t_{lo})|$, 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 $g_i(t_{mid})$ 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_{lo}, t_{mid}) but some $g_i(t_{mid}) = 0$, then that root is reported. The loop continues until $|t_{hi} - t_{lo}| < \tau$, 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

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

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, the ARKode 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.

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*, *SUNDIALS tree*, and *SUNDIALS examples*). 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 solver for ODE systems $M\dot{y} = f_E(t, y) + f_I(t, y)$ based on additive Runge-Kutta methods;

- 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 $F(u) = 0$.
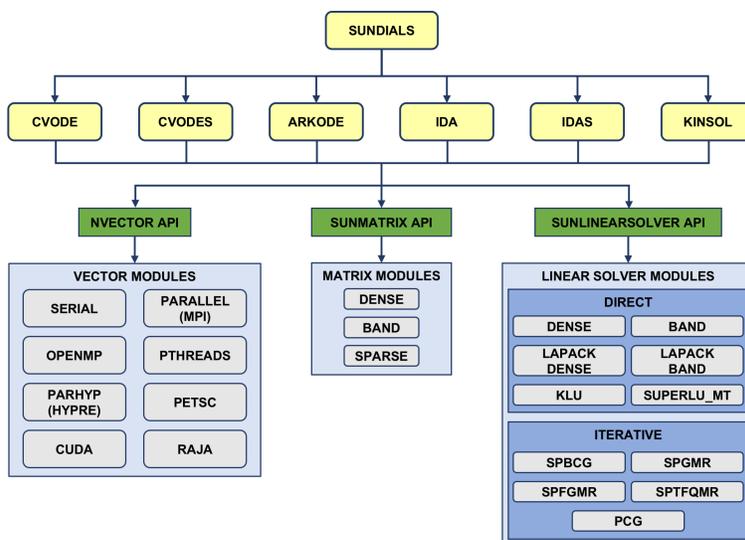

Fig. 3.1: *SUNDIALS organization*: High-level diagram of the SUNDIALS structure
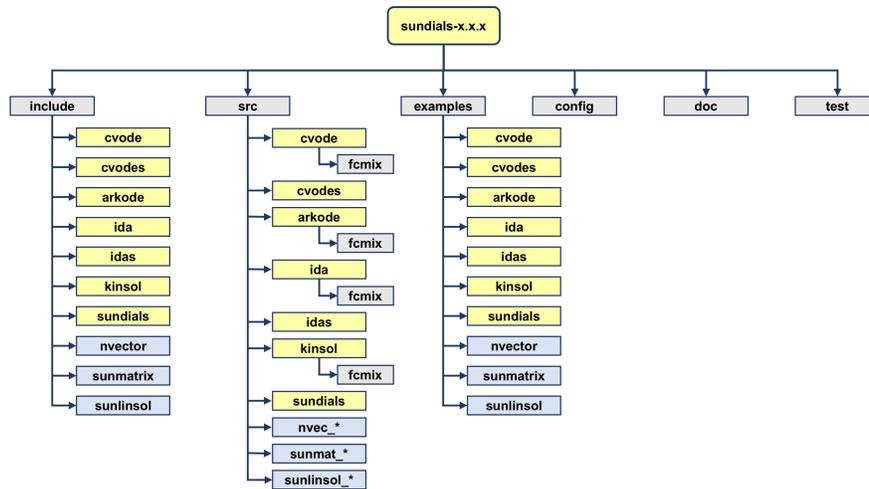
Fig. 3.2: *SUNDIALS tree*: Directory structure of the source tree.
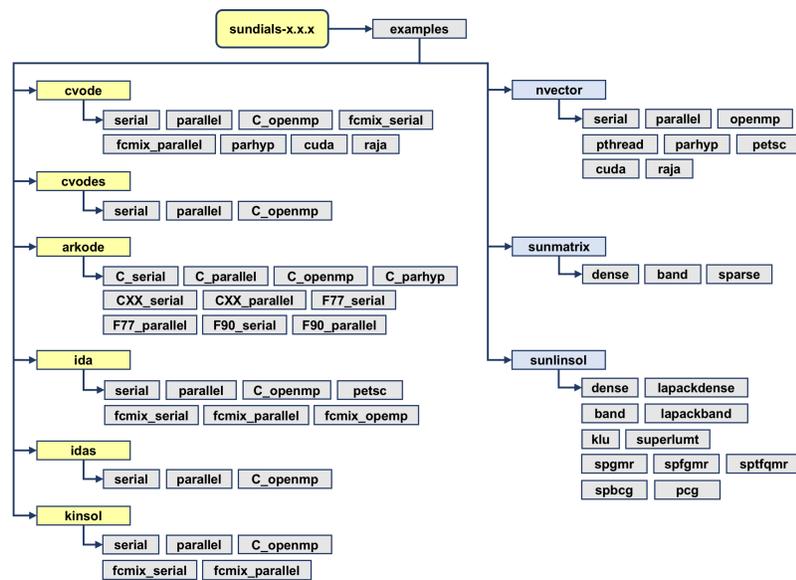
Fig. 3.3: *SUNDIALS examples*: Directory structure of the examples.

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 module, implemented in the files `arkode.h`, `arkode_impl.h` and `arkode.c`, deals with the evaluation of integration stages, the nonlinear solver (if $f_I(t, y) \neq 0$), 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 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, 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.
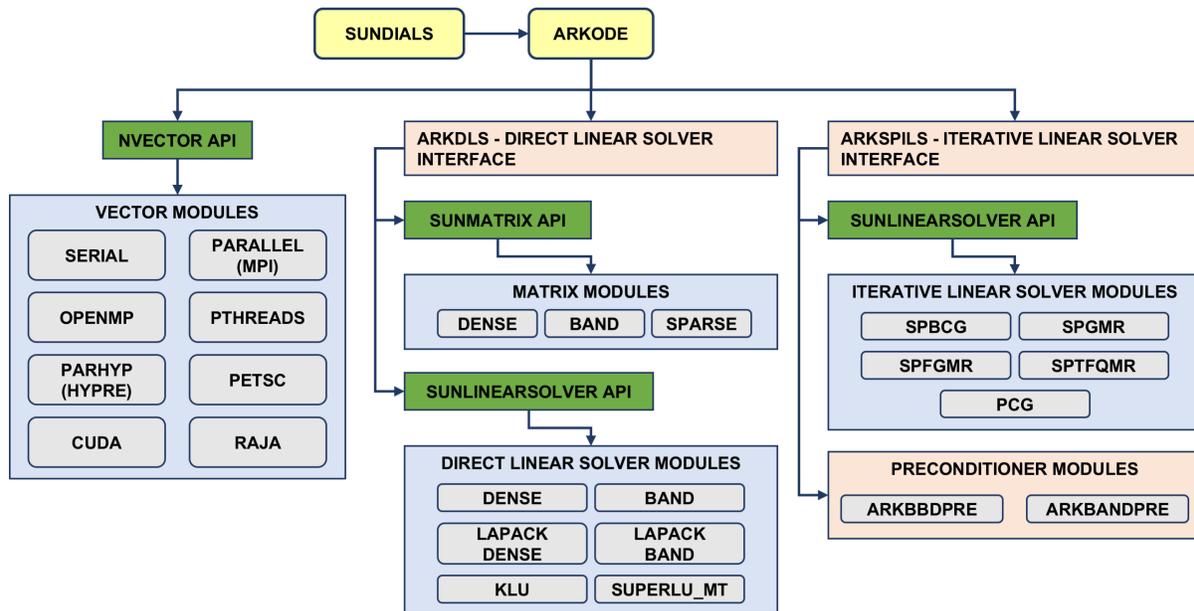

Fig. 3.4: *ARKode organization*: Overall structure of the ARKode package. Modules specific to ARKode begin with "ARK" (ARKDLS, ARKSPILS, ARKBBDPRE), all other items correspond to generic solver and auxiliary modules. Note also that the LAPACK, KLU and SuperLU_MT support is through interfaces to external packages. Users will need to download and compile those packages independently.

For solving these linear systems, ARKode presently includes two linear solver interfaces. The *direct* linear solver interface, ARKDLS, supports SUNLINSOL implementations with type `SUNLINSOL_DIRECT` (see *Linear Solver Data Structures*). These linear solvers utilize direct methods for the solution of linear systems stored using one of the SUNDIALS generic SUNMATRIX implementations (dense, banded or sparse; see *Matrix Data Structures*). It is assumed that the dominant cost for such solvers occurs in factorization of the linear system matrix A, so ARKode utilizes these solvers within its modified Newton nonlinear solve. The *iterative* linear solver interface, ARKSPILS, supports SUNLINSOL implementations with type `SUNLINSOL_ITERATIVE` (see *Linear Solver Data Structures*).

These linear solvers utilize scaled preconditioned iterative methods. It is assumed that these methods are implemented in a "matrix-free" manner, wherein only the action of the matrix-vector product Av is required. Since ARKode can operate on any valid SUNLINSOL implementation of `SUNLINSOL_DIRECT` or `SUNLINSOL_ITERATIVE` types, the set of linear solver modules available to ARKode will expand as new SUNLINSOL modules are developed.

Within the ARKDLS interface, the package includes algorithms for the approximation of dense or banded Jacobians through difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. This user-supplied routine is required when using sparse Jacobian matrices, since standard difference quotient approximations do not leverage the inherent sparsity of the problem. Additionally, when solving problems with non-identity mass matrices using the ARKDLS interface, a user-supplied routine is required for providing the mass matrix.

Within the ARKSPILS interface, the package 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 using ARKSPILS to solve problems with non-identity mass matrices, corresponding user-supplied routines for computing the product Mv are required. 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.

Each ARKode 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 is saved in a separate data structure, a pointer to which resides in the ARKode memory structure. We note that the ARKode Fortran interface, however, currently uses global variables.

USING ARKODE FOR C AND C++ APPLICATIONS

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

The example programs described in the companion document *[R2013]* 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, and may look to the Fortran example programs also described in the companion document *[R2013]*. These codes are also located in the ARKode package examples directory.

The user should be aware that not all SUNLINSOL, SUNMATRIX preconditioning modules are compatible with all NVECTOR implementations. Details on compatability are given in the documentation for each SUNMATRIX (see *Matrix Data Structures*) and each SUNLINSOL module (see *Linear Solver Data Structures*). For example, NVEC-TOR_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 *Linear Solver Data Structures* to verify compatability 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.

ARKode uses various constants for both input and output. 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 ARKode'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` (one or two files),

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`

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.

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*).

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` 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*).

4.3 Header Files

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.h`, the main header file for ARKode, which defines the several types and various constants, and includes function prototypes.

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.

The calling program must also include an NVECTOR implementation header file, of the form `nvector/nvector_***.h`. 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 includes a non-trivial implicit component to their ODE system, then each time step will require a nonlinear solver for the resulting systems of equations. ARKode allows an accelerated fixed point iteration and Newton-based iterations for this solver; if a Newton method is used then a linear solver module header file may also be required. Similarly, if the ODE system

$$My' = f_I(t, y) + f_E(t, y)$$

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 various linear solver interfaces and linear solver modules available for use with ARKode for either the Newton solver or for mass-matrix solves, are:

- `arkode/arkode_direct.h`, which is used with the ARKDLS direct linear solver interface to access direct solvers with the following header files:

  - `sunlinsol/sunlinsol_dense.h`, which is used with the dense linear solver module, SUNLIN-SOL_DENSE;

  - `sunlinsol/sunlinsol_band.h`, which is used with the banded linear solver module, SUNLIN-SOL_BAND;

  - `sunlinsol/sunlinsol_lapackdense.h`, which is used with the LAPACK dense linear solver interface module, SUNLINSOL_LAPACKDENSE;

  - `sunlinsol/sunlinsol_lapackband.h`, which is used with the LAPACK banded linear solver interface module, SUNLINSOL_LAPACKBAND;

  - `sunlinsol/sunlinsol_klu.h`, which is used with the {klu} sparse linear solver interface module, SUNLINSOL_KLU;

  - `sunlinsol/sunlinsol_superlumt.h`, which is used with the SuperLU_MT sparse linear solver interface module, SUNLINSOL_SUPERLUMT;

- `arkode/arkode_spils.h`, which is used with the ARKSPILS iterative linear solver interface to access iterative solvers with the following header files:

  - `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 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 a block-diagonal matrix, the header `sundials/sundials_dense.h` may need to be included for access to the underlying generic dense matrix arithmetic routines used in the preconditioner solve.

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. Most of the steps are independent of the NVECTOR, SUNMATRIX, and SUNLINSOL implementations used. For the steps that are not, refer to the sections *Vector Data Structures*, *Matrix Data Structures* and *Linear Solver 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 ARKode object

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

5. Initialize ARKode solver

   Call *ARKodeInit()* to provide required problem specifications, allocate internal memory for ARKode, and initialize ARKode. *ARKodeInit()* returns a flag, the value of which indicates either success or an illegal argument value. See the section *ARKode initialization and deallocation functions* for details.

6. Specify integration tolerances

   Call *ARKodeSStolerances()* or *ARKodeSVtolerances()* 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 *ARKodeWFtolerances()* to specify a function which sets directly the weights used in evaluating WRMS vector norms. See the section *ARKode 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 *ARKodeResStolerance()*, *ARKodeResVtolerance()* or *ARKodeResFtolerance()*.

7. Set optional inputs

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

8. Create matrix object

   If a direct linear solver is to be used within a Newton iteration or for solving non-identity mass matrix systems, then a template Jacobian and/or 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.

9. Create linear solver object

   If a Newton iteration is chosen, 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.

10. 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 *Linear Solver Data Structures* for details.

11. Attach linear solver module

    If a Newton iteration is chosen for implicit or ImEx methods, initialize the ARKDLS or ARKSPILS linear solver interface by attaching the linear solver object (and Jacobian matrix object, if applicable) with one of the following calls (for details see the section *Linear solver interface functions*):

    ```
    ier = ARKDlsSetLinearSolver(...);

    ier = ARKSpilsSetLinearSolver(...);
    ```

    Similarly, if the problem involves a non-identity mass matrix, initialize the ARKDLS or ARKSPILS mass matrix linear solver interface by attaching the mass linear solver object (and mass matrix object, if applicable) with one of the following calls (for details see the section *Linear solver interface functions*):

    ```
    ier = ARKDlsSetMassLinearSolver(...);

    ier = ARKSpilsSetMassLinearSolver(...);
    ```

12. Set linear solver interface optional inputs

    Call `ARKDlsSet*` or `ARKSpilsSet*` functions to change optional inputs specific to that linear solver interface. See the section *Optional input functions* for details.

13. Specify rootfinding problem

    Optionally, call *ARKodeRootInit()* 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.

14. Advance solution in time

    For each point at which output is desired, call

    ```
    ier = ARKode(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 $y(t_{out})$. See the section *ARKode solver function* for details.

15. Get optional outputs

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

16. Deallocate memory for solution vector

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

    ```
    N_VDestroy_***(y);
    ```

17. Free solver memory

    Call `ARKodeFree(&arkode_mem)` to free the memory allocated for ARKode.

18. 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.

19. 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 *Linear Solver Data Structures* 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_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 User-callable functions

This section describes the ARKode functions that are called by the user to setup and then solve an IVP. 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. 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 and sends an error message to the error handler routine, which prints the message on 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 ARKode initialization and deallocation functions

void* **ARKodeCreate** ()
> This function creates an internal memory block for a problem to be solved by ARKode.
>
> **Arguments:** None
>
> **Return value:** If successful, a pointer to initialized problem memory of type void*, to be passed to *ARKodeInit()*. If unsuccessful, a NULL pointer will be returned, and an error message will be printed to stderr.

int **ARKodeInit** (void* *arkode_mem*, *ARKRhsFn fe*, *ARKRhsFn fi*, realtype *t0*, N_Vector *y0*)
> This function allocates and initializes memory for a problem to be solved by ARKode.
>
> **Arguments:**
>
> > - *arkode_mem* – pointer to the ARKode memory block (that was returned by *ARKodeCreate()*)
> >
> > - *fe* – the name of the C function (of type *ARKRhsFn()*) defining the explicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$
> >
> > - *fi* – the name of the C function (of type *ARKRhsFn()*) defining the implicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$
> >
> > - *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 ARKode memory was NULL
> >
> > - *ARK_MEM_FAIL* if a memory allocation failed
> >
> > - *ARK_ILL_INPUT* if an argument has an illegal value.

void **ARKodeFree** (void* *arkode_mem*)
> This function frees the problem memory *arkode_mem* created by *ARKodeCreate()* and allocated by *ARKodeInit()*.
>
> **Arguments:**
>
> > - *arkode_mem* – pointer to the ARKode memory block.
>
> **Return value:** None

4.5.2 ARKode tolerance specification functions

These functions specify the integration tolerances. One of them **should** be called before the first call to `ARKode()`; 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 `ARKodeSStolerances()`, this vector has components

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

whereas in the case of `ARKodeSVtolerances()` 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 `ARKodeWFtolerances()`.

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

Arguments:

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

- *reltol* – scalar relative tolerance

- *abstol* – scalar absolute tolerance

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

- *ARK_NO_MALLOC* if the ARKode memory was not allocated by `ARKodeInit()`

- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKodeSVtolerances** (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 ARKode 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 ARKode memory was `NULL`

- *ARK_NO_MALLOC* if the ARKode memory was not allocated by `ARKodeInit()`

- *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKodeWFtolerances** (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 ARKode 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 ARKode memory was NULL

>> - *ARK_NO_MALLOC* if the ARKode memory was not allocated by *ARKodeInit()*

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, ARKode defines a vector of residual weights, rwt for measuring convergence of these iterative solvers. In the case of *ARKodeResStolerance()*, this vector has components

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

whereas in the case of *ARKodeResVtolerance()* 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 *ARKodeResFtolerance()*. Further information on all three of these functions is provided below.

int **ARKodeResStolerance** (void* *arkode_mem*, realtype *abstol*)
> This function specifies a scalar absolute residual tolerance.

> **Arguments:**

>> - *arkode_mem* – pointer to the ARKode memory block.

>> - *rabstol* – scalar absolute residual tolerance

> **Return value:**

>> - *ARK_SUCCESS* if successful

>> - *ARK_MEM_NULL* if the ARKode memory was NULL

>> - *ARK_NO_MALLOC* if the ARKode memory was not allocated by *ARKodeInit()*

>> - *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

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

> **Arguments:**
>
> - *arkode_mem* – pointer to the ARKode memory block.
>
> - *rabstol* – vector containing the absolute residual tolerances for each solution component
>
> **Return value:**
>
> - *ARK_SUCCESS* if successful
>
> - *ARK_MEM_NULL* if the ARKode memory was NULL
>
> - *ARK_NO_MALLOC* if the ARKode memory was not allocated by *ARKodeInit()*
>
> - *ARK_ILL_INPUT* if an argument has an illegal value (e.g. a negative tolerance).

int **ARKodeResFtolerance** (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 ARKode 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 ARKode memory was NULL
>
> - *ARK_NO_MALLOC* if the ARKode memory was not allocated by *ARKodeInit()*

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 *[R2013]*. In that problem, the three components vary betwen 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. I.e. if you want .01% relative accuracy (globally), a good choice for `reltol` is 10^{-5}. But 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.

Advice on controlling unphysical 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 (unphysical) 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 ARKode, 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, ARKode 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 ARKode requires 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

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

There are two ARKode linear solver interfaces currently available for this task: ARKDLS and ARKSPILS.

The first corresponds to the use of Direct Linear Solvers, and utilizes `SUNMatrix` objects to store the approximate Jacobian J, the Newton matrix $\mathcal{A}$, the mass matrix M, and factorizations used throughout the solution process.

The second corresponds to the use of Scaled, Preconditioned, Iterative Linear Solvers, utilizing matrix-free Krylov methods to solve the Newton systems of equations. 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 ARKode to use for the Newton systems, after the call to `ARKodeCreate()` but before any calls to `ARKode()`, the user's program must create the appropriate SUNLinearSolver object and call either of the functions `ARKDlsSetLinearSolver()` or `ARKSpilsSetLinearSolver()`, as documented below. The first argument passed to these functions is the ARKode memory pointer returned by `ARKodeCreate()`; the second argument passed to these functions is the desired `SUNLinearSolver` object to use for solving Newton systems. A call to one of these functions initializes the appropriate ARKode linear solver interface, linking this to the main ARKode integrator, and allows the user to specify parameters which are specific to a particular solver interface.

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 *Linear Solver Data Structures*.

int **ARKDlsSetLinearSolver** (void* *arkode_mem*, SUNLinearSolver *LS*, SUNMatrix *J*)
>   This function specifies the direct `SUNLinearSolver` object that ARKode should use, as well as a template Jacobian `SUNMatrix` object. Its use requires inclusion of the header file `arkode/arkode_direct.h`.
>
>   **Arguments:**
>
>   - *arkode_mem* – pointer to the ARKode memory block.
>
>   - *LS* – the `SUNLinearSolver` object to use.
>
>   - *J* – the template Jacobian `SUNMatrix` object to use.
>
>   **Return value:**
>
>   - *ARKDLS_SUCCESS* if successful
>
>   - *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`
>
>   - *ARKDLS_MEM_FAIL* if there was a memory allocation failure
>
>   - *ARKDLS_ILL_INPUT* if ARKDLS is incompatible with the provided *LS* or *J* input objects, or the current `N_Vector` module.
>
>   **Notes:** 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.
>
>   The ARKDLS linear solver interface is not compatible with all implementations of the `SUNLinearSolver` and `N_Vector` modules. Specifically, ARKDLS requires use of a *direct* `SUNLinearSolver` object and a serial or threaded `N_Vector` module. Additional compatibility limitations for each `SUNLinearSolver` object (i.e. `SUNMatrix` and `N_Vector` object compatibility) are described in the section *Linear Solver Data Structures*.

int **ARKSpilsSetLinearSolver** (void* *arkode_mem*, SUNLinearSolver *LS*)
>   This function specifies the iterative `SUNLinearSolver` object that ARKode should use, initializing the ARK-SPILS scaled, preconditioned, iterative linear solver interface. Its use requires inclusion of the header file `arkode/arkode_spils.h`.
>
>   **Arguments:**
>
>   - *arkode_mem* – pointer to the ARKode memory block.

- *LS* – the `SUNLinearSolver` object to use.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure

- *ARKSPILS_ILL_INPUT* if ARKSPILS is incompatible with the provided *LS* input objects, or the current `N_Vector` module.

Notes: The ARKSPILS linear solver interface is not compatible with all implementations of the `SUNLinearSolver` and `N_Vector` modules. Specifically, ARKSPILS requires use of an *iterative* `SUNLinearSolver` object, and a minimum required set of vector operations must be provided by the current `N_Vector` module. Additional compatibility limitations for each `SUNLinearSolver` object (i.e. required `N_Vector` routines) are described in the section *Linear Solver Data Structures*.

4.5.4 Mass matrix solver specification functions

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

$$Mx = b.$$

The same solver interfaces listed above in the section *Linear solver interface functions* may be used for this purpose: ARKDLS and ARKSPILS. With the ARKSPILS interface 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 pre-conditioner 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 ARKode to use for mass matrix systems, after the call to *ARKodeCreate()* but before any calls to *ARKode()*, the user's program must create the appropriate `SUNLinearSolver` object and call either of the functions *ARKDlsSetMassLinearSolver()* or *ARKSpilsSetMassLinearSolver()*, as documented below. The first argument passed to these functions is the ARKode memory pointer returned by *ARKodeCreate()*; the second argument passed to these functions is the desired `SUNLinearSolver` object to use for solving mass matrix systems. A call to one of these functions initializes the appropriate ARKode mass matrix linear solver interface, linking this to the main ARKode integrator, and allows the user to specify parameters which are specific to a particular solver interface.

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 *Linear Solver Data Structures*.

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 *direct*, then they must utilize the same `SUNMatrix` type. In this case, both the Newton and mass matrix linear solver interfaces can use the same `SUNLinearSolver` object, although different objects (e.g. with different solver parameters) are also allowed.

- If both are *iterative*, 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 **ARKDlsSetMassLinearSolver** (void* *arkode_mem*, SUNLinearSolver *LS*, SUNMatrix *M*, boolean-type *time_dep*)

This function specifies the direct `SUNLinearSolver` object that ARKode should use for mass matrix systems, as well as a template `SUNMatrix` object. Its use requires inclusion of the header file `arkode/arkode_direct.h`.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ($M \neq M(t)$). Use `SUNTRUE` to indicate time-dependence of the mass matrix.

Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKDLS_MEM_FAIL* if there was a memory allocation failure

- *ARKDLS_ILL_INPUT* if ARKDLS is incompatible with the provided *LS* or *M* input objects, or the current `N_Vector` module.

Notes: 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, with the results reused throughout the entire ARKode simulation.

Unlike the system Jacobian, the system mass matrix cannot be approximated using finite-differences of any functions provided to ARKode. Hence, use of the ARKDLS mass matrix solver interface requires the user to provide a mass-matrix constructor routine (see *ARKDlsMassFn* and *ARKDlsSetMassFn()*).

The ARKDLS linear solver interface is not compatible with all implementations of the `SUNLinearSolver` and `N_Vector` modules. Specifically, ARKDLS requires use of a *direct* `SUNLinearSolver` object and a serial or threaded `N_Vector` module. Additional compatibility limitations for each `SUNLinearSolver` object (i.e. `SUNMatrix` and `N_Vector` object compatibility) are described in the section *Linear Solver Data Structures*.

int **ARKSpilsSetMassLinearSolver** (void* *arkode_mem*, SUNLinearSolver *LS*, booleantype *time_dep*)

This function specifies the iterative `SUNLinearSolver` object that ARKode should use for mass matrix systems, initializing the ARKSPILS scaled, preconditioned, iterative mass matrix linear solver interface. Its use requires inclusion of the header file `arkode/arkode_spils.h`.

Arguments:

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

- *LS* – the `SUNLinearSolver` object to use.

- *time_dep* – flag denoting whether the mass matrix depends on the independent variable ($M = M(t)$) or not ($M \neq M(t)$). Use `SUNTRUE` to indicate time-dependence of the mass matrix.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_MEM_FAIL* if there was a memory allocation failure

- *ARKSPILS_ILL_INPUT* if ARKSPILS is incompatible with the provided *LS* input objects, or the current `N_Vector` module.

Notes: If called with *time_dep* set to `SUNFALSE`, then the mass matrix-vector-product (if supplied) is only set up once, and the mass matrix preconditioner (if supplied) is only set up once, with the results reused throughout the entire ARKode simulation.

Unlike the system Jacobian, the system mass matrix-vector-product cannot be approximated using finite-differences of any functions provided to ARKode. Hence, use of the ARKSPILS mass matrix solver interface requires the user to provide a mass-matrix-times-vector product routine (see *ARKSpilsMassTimesVecFn* and *ARKSpilsSetMassTimes()*).

The ARKSPILS linear solver interface is not compatible with all implementations of the `SUNLinearSolver` and `N_Vector` modules. Specifically, ARKSPILS requires use of an *iterative* `SUNLinearSolver` object, and a minimum required set of vector operations must be provided by the current `N_Vector` module. Additional compatibility limitations for each `SUNLinearSolver` object (i.e. required `N_Vector` routines) are described in the section *Linear Solver Data Structures*.

4.5.5 Rootfinding initialization function

As described in the section *Rootfinding*, while solving the IVP ARKode has 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 *ARKode()*, but if the rootfinding problem is to be changed during the solution, *ARKodeRootInit()* can also be called prior to a continuation call to *ARKode()*.

int **ARKodeRootInit** (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 *ARKodeCreate()*, and before *ARKode()*.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode 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 ARKode's rootfinding module, call *ARKodeRootInit* with *nrtfn = 0*.

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

4.5.6 ARKode 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 ARKode 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 *ARKodeSetStopTime()*) or has requested rootfinding.

int **ARKode** (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 ARKode 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 reached or just passed the user-specified *tout* parameter. The solver then interpolates in order to return an approximate value of $y(tout)$. This interpolation may be slightly less accurate than the full time step solutions produced by the solver, since the interpolation uses a cubic Hermite polynomial even when the RK method is of higher order.

>> To ensure that this returned value has full method accuracy, issue a call to *ARKodeSetStopTime()* before the call to ARKode to specify a fixed stop time to end the time step and return to the user. Once the integrator returns at a *tstop* time, any future testing for *tstop* is disabled (and can be reenabled only though a new call to *ARKodeSetStopTime()*).

>> The *ARK_ONE_STEP* option tells the solver to take just one internal step and then return the solution at the point reached by that step.

> **Return value:**

>> - *ARK_SUCCESS* if successful

>> - *ARK_ROOT_RETURN* if ARKode succeeded, and found one or more roots. If *nrtfn* is greater than 1, call *ARKodeGetRootInfo()* to see which g_i were found to have a root at (**tret*).

>> - *ARK_TSTOP_RETURN* if ARKode 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 ARKode 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. The tolerances have not been set.

>>> 2. A component of the error weight vector became zero during internal time-stepping.

>>> 3. The linear solver initialization function (called by the user after calling *ARKodeCreate()*) failed to set the linear solver-specific *lsolve* field in *arkode_mem*.

>>> 4. 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_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.

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

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 ARKode failures.

On any error return in which one or more internal steps were taken by ARKode, 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.

4.5.7 Optional input functions

There are numerous optional input parameters that control the behavior of the ARKode 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 ARKode 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 solver options (*Optional inputs for ARKode*),

- 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*),

- Direct linear solver interface options (*Direct linear solver interface optional input functions*),

- Iterative linear solver interface options (*Iterative linear solvers optional input functions*).

For the most casual use of ARKode, 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. We also note that all error return values are negative, so a test on the return arguments for negative values will catch all errors.

Optional inputs for ARKode

Optional input	Function name	Default
Return all solver parameters to their defaults	`ARKodeSetDefaults()`	internal
Set dense output order	`ARKodeSetDenseOrder()`	3
Supply a pointer to a diagnostics output file	`ARKodeSetDiagnostics()`	`NULL`
Supply a pointer to an error output file	`ARKodeSetErrFile()`	`stderr`
Supply a custom error handler function	`ARKodeSetErrHandlerFn()`	internal fn
Supply an initial step size to attempt	`ARKodeSetInitStep()`	estimated
Disable time step adaptivity (fixed-step mode)	`ARKodeSetFixedStep()`	disabled
Maximum no. of warnings for $t_n + h = t_n$	`ARKodeSetMaxHnilWarns()`	10
Maximum no. of internal steps before *tout*	`ARKodeSetMaxNumSteps()`	500
Maximum no. of error test failures	`ARKodeSetMaxErrTestFails()`	7
Maximum absolute step size	`ARKodeSetMaxStep()`	∞
Minimum absolute step size	`ARKodeSetMinStep()`	0.0
Set 'optimal' adaptivity params for a method	`ARKodeSetOptimalParams()`	internal
Set a value for t_{stop}	`ARKodeSetStopTime()`	∞
Supply a pointer for user data	`ARKodeSetUserData()`	`NULL`

int **ARKodeSetDefaults** (void* *arkode_mem*)
　　Resets all optional input parameters to ARKode's original default values.

　　Arguments:

　　　　• *arkode_mem* – pointer to the ARKode memory block.

　　Return value:

　　　　• *ARK_SUCCESS* if successful

　　　　• *ARK_MEM_NULL* if the ARKode memory is `NULL`

　　　　• *ARK_ILL_INPUT* if an argument has an illegal value

　　Notes: Does not change problem-defining function pointers *fe* and *fi* or the *user_data* pointer.

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

int **ARKodeSetDenseOrder** (void* *arkode_mem*, int *dord*)
　　Specifies the order of accuracy for the polynomial interpolant used for dense output (i.e. interpolation of solution output values and implicit method predictors).

　　Arguments:

　　　　• *arkode_mem* – pointer to the ARKode memory block.

　　　　• *dord* – requested polynomial order of accuracy

　　Return value:

　　　　• *ARK_SUCCESS* if successful

　　　　• *ARK_MEM_NULL* if the ARKode memory is `NULL`

　　　　• *ARK_ILL_INPUT* if an argument has an illegal value

　　Notes: Allowed values are between 0 and `min(q, 3)`, where `q` is the order of the overall integration method.

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

Arguments:

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

- *diagfp* – pointer to the diagnostics output file

Return value:

- *ARK_SUCCESS* if successful

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

Arguments:

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

- *errfp* – pointer to the output file.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 ARKode 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 **ARKodeSetErrHandlerFn** (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 ARKode 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 ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

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

int **ARKodeSetInitStep** (void* *arkode_mem*, realtype *hin*)
    Specifies the initial time step size ARKode should use after initialization or reinitialization.

    Arguments:

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

- *hin* – value of the initial step to be attempted (≥ 0)

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is NULL

- *ARK_ILL_INPUT* if an argument has an illegal value

    Notes: Pass 0.0 to use the default value.

    By default, ARKode 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*.

int **ARKodeSetFixedStep** (void* *arkode_mem*, realtype *hfixed*)
    Disabled time step adaptivity within ARKode, and specifies the fixed time step size to use for all internal steps.

    Arguments:

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

- *hfixed* – value of the fixed step size to use

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is NULL

- *ARK_ILL_INPUT* if an argument has an illegal value

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

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

    When using *ARKodeSetFixedStep()*, any values provided to the functions *ARKodeSetInitStep()*, *ARKodeSetAdaptivityFn()*, *ARKodeSetMaxErrTestFails()*, *ARKodeSetAdaptivityMethod()*, *ARKodeSetCFLFraction()*, *ARKodeSetErrorBias()*, *ARKodeSetFixedStepBounds()*, *ARKodeSetMaxCFailGrowth()*, *ARKodeSetMaxEFailGrowth()*, *ARKodeSetMaxFirstGrowth()*, *ARKodeSetMaxGrowth()*, *ARKodeSetSafetyFactor()*, *ARKodeSetSmallNumEFails()* and *ARKodeSetStabilityFn()* will be ignored, since temporal adaptivity is disabled.

    If both *ARKodeSetFixedStep()* and *ARKodeSetStopTime()* 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 *ARKodeSetFixedStep()* must be made prior to calling *ARKode()* to resume integration.

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

int **ARKodeSetMaxHnilWarns** (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 ARKode will instead return with an error.

    Arguments:

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

- *mxhnil* – maximum allowed number of warning messages (>0).

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 **ARKodeSetMaxNumSteps** (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 ARKode will return with an error.

Arguments:

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

- *mxsteps* – maximum allowed number of internal steps.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

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

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

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

Arguments:

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

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

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 **ARKodeSetMaxStep** (void* *arkode_mem*, realtype *hmax*)
Specifies the upper bound on the magnitude of the time step size.

Arguments:

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

- *hmax* – maximum absolute value of the time step size (≥ 0)

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass $hmax \leq 0.0$ to set the default value of ∞.

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

Arguments:

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

- *hmin* – minimum absolute value of the time step size (≥ 0)

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Pass $hmin \leq 0.0$ to set the default value of 0.

int **ARKodeSetOptimalParams** (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 ARKode memory block.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 ARKode's solvers on a variety of training problems. However, all problems are different, so these values may not be optimal for all users.

int **ARKodeSetStopTime** (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 ARKode memory block.

- *tstop* – stopping time for the integrator.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

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

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

Arguments:

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

- *user_data* – pointer to the user data.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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.

Optional inputs for IVP method selection

Optional input	Function name	Default
Set integrator method order	*ARKodeSetOrder()*	4
Specify implicit/explicit problem	*ARKodeSetImEx()*	SUNTRUE
Specify explicit problem	*ARKodeSetExplicit()*	SUNFALSE
Specify implicit problem	*ARKodeSetImplicit()*	SUNFALSE
Set additive RK tables	*ARKodeSetARKTables()*	internal
Set explicit RK table	*ARKodeSetERKTable()*	internal
Set implicit RK table	*ARKodeSetIRKTable()*	internal
Specify additive RK table numbers	*ARKodeSetARKTableNum()*	internal
Specify explicit RK table number	*ARKodeSetERKTableNum()*	internal
Specify implicit RK table number	*ARKodeSetIRKTableNum()*	internal

int **ARKodeSetOrder** (void* *arkode_mem*, int *ord*)
　　Specifies the order of accuracy for the integration method.

　　Arguments:

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

- *ord* – requested order of accuracy.

　　Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

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

Since *ord* affects the memory requirements for the internal ARKode memory block, it cannot be increased between calls to *ARKode()* unless *ARKodeReInit()* is called.

int **ARKodeSetImEx** (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 ARKode memory block.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 `ARKodeInit()` are `NULL`, but may be set directly by the user if desired.

int **ARKodeSetExplicit** (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 ARKode memory block.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 `ARKodeInit()` is `NULL`, but may be set directly by the user if desired.

int **ARKodeSetImplicit** (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 ARKode memory block.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 `ARKodeInit()` is `NULL`, but may be set directly by the user if desired.

int **ARKodeSetARKTables** (void* *arkode_mem*, int *s*, int *q*, int *p*, realtype* *ci*, realtype* *ce*, realtype* *Ai*, realtype* *Ae*, realtype* *bi*, realtype* *be*, realtype* *b2i*, realtype* *b2e*)

Specifies a customized Butcher table pair for the additive RK method.

Arguments:

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

- *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.

- *ci* – array (of length *s*) of stage times for the implicit RK method.

- *ce* – array (of length *s*) of stage times for the explicit RK method.

- *Ai* – array of coefficients defining the implicit RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.

- *Ae* – array of coefficients defining the explicit RK stages. This should be stored as a 1D array of size *s*s*, in row-major order.

- *bi* – array of implicit coefficients (of length *s*) defining the time step solution.

- *be* – array of explicit coefficients (of length *s*) defining the time step solution.

- *b2i* – array of implicit coefficients (of length *s*) defining the embedded solution.

- *b2e* – array of explicit coefficients (of length *s*) defining the embedded solution.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This automatically calls `ARKodeSetImEx()`.

No error checking is performed to ensure that either *p* or *q* correctly describe the coefficients that were input.

Error checking is performed on both *Ai* and *Ae* to ensure that they specify DIRK and ERK methods, respectively.

If either the inputs *b2i* or *b2e* are set to `NULL`, ARKode will run in fixed-step mode (see `ARKodeSetFixedStep()`); if called in this manner the user *must* call either `ARKodeSetFixedStep()` or `ARKodeSetInitStep()` to set the desired time step size.

int **ARKodeSetERKTable** (void* *arkode_mem*, int *s*, int *q*, int *p*, realtype* *c*, realtype* *A*, realtype* *b*, real-type* *bembed*)
Specifies a customized Butcher table for the explicit portion of the system.

Arguments:

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

- *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.

- *bembed* – array of coefficients (of length *s*) defining the embedded solution.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This automatically calls `ARKodeSetExplicit()`.

No error checking is performed to ensure that either *p* or *q* correctly describe the coefficients that were input.

Error checking is performed to ensure that *A* is strictly lower-triangular (i.e. that it specifies an ERK method).

An input *bembed* of `NULL` will signal that ARKode will run in fixed-step mode (see `ARKodeSetFixedStep()`); if called in this manner the user *must* call either `ARKodeSetFixedStep()` or `ARKodeSetInitStep()` to set the desired time step size.

int **ARKodeSetIRKTable** (void* *arkode_mem*, int *s*, int *q*, int *p*, realtype* *c*, realtype* *A*, realtype* *b*, realtype* *bembed*)

Specifies a customized Butcher table for the implicit portion of the system.

Arguments:

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

- *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.

- *bembed* – array of coefficients (of length *s*) defining the embedded solution.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: This automatically calls *ARKodeSetImplicit()*.

No error checking is performed to ensure that either *p* or *q* correctly describe the coefficients that were input.

Error checking is performed to ensure that *A* is lower-triangular with a nonzero value on at least one of the diagonal entries (i.e. that it specifies a DIRK method).

An input *bembed* of `NULL` will signal that ARKode will run in fixed-step mode (see *ARKodeSetFixedStep()*); if called in this manner the user *must* call either *ARKodeSetFixedStep()* or *ARKodeSetInitStep()* to set the desired time step size.

int **ARKodeSetARKTableNum** (void* *arkode_mem*, int *itable*, int *etable*)

Indicates to use specific built-in Butcher tables for the ImEx system.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: Both *itable* and *etable* should match an existing implicit/explicit pair, listed in the section *Additive Butcher tables*. Error-checking is performed to ensure that the tables exist. Subsequent error-checking is automatically performed to ensure that the tables' stage times and solution coefficients match.

This automatically calls *ARKodeSetImEx()*.

int **ARKodeSetERKTableNum** (void* *arkode_mem*, int *etable*)

Indicates to use a specific built-in Butcher table for explicit integration of the problem.

Arguments:

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

- *etable* – index of the Butcher table.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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.

This automatically calls *ARKodeSetExplicit()*.

int **ARKodeSetIRKTableNum** (void* *arkode_mem*, int *itable*)
Indicates to use a specific built-in Butcher table for implicit integration of the problem.

Arguments:

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

- *itable* – index of the Butcher table.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is NULL

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: *itable* should match an existing implicit method from the section *Implicit Butcher tables*. Error-checking is performed to ensure that the table exists, and is not explicit.

This automatically calls *ARKodeSetImplicit()*.

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	*ARKodeSetAdaptivityFn()*	internal
Choose an existing time step adaptivity method	*ARKodeSetAdaptivityMethod()*	0
Explicit stability safety factor	*ARKodeSetCFLFraction()*	0.5
Time step error bias factor	*ARKodeSetErrorBias()*	1.5
Bounds determining no change in step size	*ARKodeSetFixedStepBounds()*	1.0 1.5
Maximum step growth factor on convergence fail	*ARKodeSetMaxCFailGrowth()*	0.25
Maximum step growth factor on error test fail	*ARKodeSetMaxEFailGrowth()*	0.3
Maximum first step growth factor	*ARKodeSetMaxFirstGrowth()*	10000.0
Maximum general step growth factor	*ARKodeSetMaxGrowth()*	20.0
Time step safety factor	*ARKodeSetSafetyFactor()*	0.96
Error fails before MaxEFailGrowth takes effect	*ARKodeSetSmallNumEFails()*	2
Explicit stability function	*ARKodeSetStabilityFn()*	internal

int **ARKodeSetAdaptivityFn** (void* *arkode_mem*, *ARKAdaptFn* *hfun*, void* *h_data*)
Sets a user-supplied time-step adaptivity function.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode 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 *ARKodeSetStabilityFn()* should be used instead.

int **ARKodeSetAdaptivityMethod** (void* *arkode_mem*, int *imethod*, int *idefault*, int *pq*, real-type* *adapt_params*)
Specifies the method (and associated parameters) used for time step adaptivity.

Arguments:

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

- *imethod* – accuracy-based adaptivity method choice ($0 \leq imethod \leq 5$): 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 ARKode 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 ARKode 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 *ARKodeSetAdaptivityFn()*.

int **ARKodeSetCFLFraction** (void* *arkode_mem*, realtype *cfl_frac*)
Specifies the fraction of the estimated explicitly stable step to use.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode 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 **ARKodeSetErrorBias** (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 ARKode 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 ARKode 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 **ARKodeSetFixedStepBounds** (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 ARKode 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 ARKode 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 **ARKodeSetMaxCFailGrowth** (void* *arkode_mem*, realtype *etacf*)
    Specifies the maximum step size growth factor upon a convergence failure on a stage solve within a step.

    Arguments:

    - *arkode_mem* – pointer to the ARKode 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 ARKode 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 **ARKodeSetMaxEFailGrowth** (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 ARKode 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 ARKode 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 **ARKodeSetMaxFirstGrowth** (void* *arkode_mem*, realtype *etamx1*)
Specifies the maximum allowed step size change following the very first integration step.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode 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 **ARKodeSetMaxGrowth** (void* *arkode_mem*, realtype *mx_growth*)
Specifies the maximum growth of the step size between consecutive steps in the integration process.

Arguments:

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

- *growth* – maximum allowed growth factor between consecutive time steps (default is 20.0).

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 **ARKodeSetSafetyFactor** (void* *arkode_mem*, realtype *safety*)
Specifies the safety factor to be applied to the accuracy-based estimated step.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode 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 **ARKodeSetSmallNumEFails** (void* *arkode_mem*, int *small_nef*)

> Specifies the threshold for "multiple" successive error failures before the *etamxf* parameter from *ARKodeSetMaxEFailGrowth()* is applied.
>
> **Arguments:**
>
> > - *arkode_mem* – pointer to the ARKode memory block.
> >
> > - *small_nef* – bound to determine 'multiple' for *etamxf* (default is 2).
>
> **Return value:**
>
> > - *ARK_SUCCESS* if successful
> >
> > - *ARK_MEM_NULL* if the ARKode 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 **ARKodeSetStabilityFn** (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 ARKode 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 ARKode 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.

Optional inputs for implicit stage solves

The mathematical explanation for ARKode's nonlinear solver strategies, 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 use of the fixed-point stage solver	*ARKodeSetFixedPoint()*	SUNFALSE
Specify use of the Newton stage solver	*ARKodeSetNewton()*	SUNTRUE
Specify linearly implicit f_I	*ARKodeSetLinear()*	SUNFALSE
Specify nonlinearly implicit f_I	*ARKodeSetNonlinear()*	SUNTRUE
Implicit predictor method	*ARKodeSetPredictorMethod()*	0
Maximum number of nonlinear iterations	*ARKodeSetMaxNonlinIters()*	3
Coefficient in the nonlinear convergence test	*ARKodeSetNonlinConvCoef()*	0.1
Nonlinear convergence rate constant	*ARKodeSetNonlinCRDown()*	0.3
Nonlinear residual divergence ratio	*ARKodeSetNonlinRDiv()*	2.3
Max change in step signaling new J	*ARKodeSetDeltaGammaMax()*	0.2
Max steps between calls to new J	*ARKodeSetMaxStepsBetweenLSet()*	20
Maximum number of convergence failures	*ARKodeSetMaxConvFails()*	10

int **ARKodeSetFixedPoint** (void* *arkode_mem*, long int *fp_m*)
   Specifies that the implicit portion of the problem should be solved using the accelerated fixed-point solver instead of the modified Newton iteration, and provides the maximum dimension of the acceleration subspace.

   Arguments:

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

   - *fp_m* – number of vectors to store within the Anderson acceleration subspace.

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode memory is NULL

   - *ARK_ILL_INPUT* if an argument has an illegal value

   Notes: Since the accelerated fixed-point solver has a slower rate of convergence than the Newton iteration (but each iteration is typically much more efficient), it is recommended that the maximum nonlinear correction iterations be increased through a call to *ARKodeSetMaxNonlinIters()*.

int **ARKodeSetNewton** (void* *arkode_mem*)
   Specifies that the implicit portion of the problem should be solved using the modified Newton solver.

   Arguments:

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

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode memory is NULL

   - *ARK_ILL_INPUT* if an argument has an illegal value

   Notes: This is the default behavior of ARKode, so the function is primarily useful to undo a previous call to *ARKodeSetFixedPoint()*.

int **ARKodeSetLinear** (void* *arkode_mem*, int *timedepend*)
   Specifies that the implicit portion of the problem is linear.

   Arguments:

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

   - *timedepend* – flag denoting whether the Jacobian of $f_I(t, y)$ is time-dependent (1) or not (0). Alternately, when using an iterative linear solver this flag denotes time dependence of the preconditioner.

   Return value:

4.5. User-callable functions 65

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 *ARKodeSetDeltaGammaMax()* 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 Newton iteration (not the fixed-point solver).

int **ARKodeSetNonlinear** (void* *arkode_mem*)
    Specifies that the implicit portion of the problem is nonlinear.

    Arguments:

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

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

    Notes: This is the default behavior of ARKode, so the function is primarily useful to undo a previous call to *ARKodeSetLinear()*. Calls *ARKodeSetDeltaGammaMax()* to reset the step size ratio threshold to the default value.

int **ARKodeSetPredictorMethod** (void* *arkode_mem*, int *method*)
    Specifies the method to use for predicting implicit solutions.

    Arguments:

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

- *method* – method choice ($0 \leq method \leq 4$):

  - 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 ARKode 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 **ARKodeSetMaxNonlinIters** (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 ARKode memory block.

- *maxcor* – maximum allowed solver iterations per stage (> 0).

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory is `NULL`

- *ARK_ILL_INPUT* if an argument has an illegal value

Notes: The default value is 3; set *maxcor* ≤ 0 to specify this default.

int **ARKodeSetNonlinConvCoef** (void* *arkode_mem*, realtype *nlscoef*)
Specifies the safety factor used within the nonlinear solver convergence test.

Arguments:

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

- *nlscoef* – coefficient in nonlinear solver convergence test (> 0.0).

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 **ARKodeSetNonlinCRDown** (void* *arkode_mem*, realtype *crdown*)
Specifies the constant used in estimating the nonlinear solver convergence rate.

Arguments:

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

- *crdown* – nonlinear convergence rate estimation constant (default is 0.3).

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 **ARKodeSetNonlinRDiv** (void* *arkode_mem*, realtype *rdiv*)
Specifies the nonlinear correction threshold beyond which the iteration will be declared divergent.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode 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 **ARKodeSetDeltaGammaMax** (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 ARKode 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 ARKode 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 **ARKodeSetMaxStepsBetweenLSet** (void* *arkode_mem*, int *msbp*)
   Specifies the frequency of calls to the linear solver setup routine. Positive values specify the number of time steps between setup calls; negative values force recomputation at each Newton step; zero values reset to the default.

   Arguments:

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

   - *msbp* – maximum number of time steps between linear solver setup calls, or flag to force recomputation at each Newton iteration (default is 20).

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode memory is NULL

int **ARKodeSetMaxConvFails** (void* *arkode_mem*, int *maxncf*)
   Specifies the maximum number of nonlinear solver convergence failures permitted during one step, before ARKode will return with an error.

   Arguments:

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

   - *maxncf* – maximum allowed nonlinear solver convergence failures per step (> 0).

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode 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, ARKode 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 *ARKodeSetMaxCFailGrowth()*).

Direct linear solver interface optional input functions

The mathematical explanation of ARKode's direct linear solver methods is provided in the section *Linear solver methods*.

Table: Optional inputs for ARKDLS

Optional input	Function name	Default
Jacobian function	*ARKDlsSetJacFn()*	DQ
Mass matrix function	*ARKDlsSetMassFn()*	none

The ARKDLS solver interface needs a function to compute an approximation to the Jacobian matrix $J(t, y)$. This function must be of type *ARKDlsJacFn()*. 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 ARKDLS interface. To specify a user-supplied Jacobian function *jac*, ARKDLS provides the function *ARKDlsSetJacFn()*. The ARKDLS interface passes the user data pointer to the Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The user data pointer may be specified through *ARKodeSetUserData()*.

Similarly, if the ODE system involves a non-identity mass matrix, $M \neq I$, the ARKDLS interface needs a function to compute an approximation to the mass matrix $M(t)$. There is no default difference quotient approximation, so this routine must be supplied by the user. This function must be of type *ARKDlsMassFn()*, and should be set using the function *ARKDlsSetMassFn()*. We note that the ARKDLS solver passes the user data pointer to the mass matrix function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied mass matrix function, without using global data in the program. The pointer user data may be specified through *ARKodeSetUserData()*.

int **ARKDlsSetJacFn** (void* *arkode_mem*, *ARKDlsJacFn jac*)

Specifies the Jacobian approximation routine to be used for the ARKDLS interface.

Arguments:

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

- *jac* – name of user-supplied Jacobian approximation function.

Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

Notes: This routine must be called after the ARKDLS linear solver interface has been initialized through a call to *ARKDlsSetLinearSolver()*.

By default, ARKDLS 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 a sparse matrix.

The function type *ARKDlsJacFn()* is described in the section *User-supplied functions*.

int **ARKDlsSetMassFn** (void* *arkode_mem*, *ARKDlsMassFn mass*)

Specifies the mass matrix approximation routine to be used for the ARKDLS interface.

Arguments:

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

- *mass* – name of user-supplied mass matrix approximation function.

Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_MASSMEM_NULL* if the mass matrix solver memory was NULL

> **Notes:** This routine must be called after the ARKDLS mass matrix solver interface has been initialized through a call to `ARKDlsSetMassLinearSolver()`.
>
> Since there is no default difference quotient function for mass matrices, *mass* must be non-`NULL`.
>
> The function type `ARKDlsMassFn()` is described in the section *User-supplied functions*.

Iterative linear solvers optional input functions

As described in the section *Linear solver methods*, when using the ARKSPILS iterative linear solver interface, a 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 ARKode using either the function `ARKSpilsSetPreconditioner()` (for preconditioning the Newton system), or the function `ARKSpilsSetMassPreconditioner()` (for preconditioning the mass matrix system). The *psetup* function 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 `ARKodeSetUserData()` (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.

Additionally, when solving the Newton linear systems, the ARKSPILS 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 ARKSPILS interface. A user-defined Jacobian-vector function must be of type `ARKSpilsJacTimesVecFn` and can be specified through a call to `ARKSpilsSetJacTimes()` (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 `ARKSpilsJacTimesSetupFn` (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 `ARKodeSetUserData()` (or a `NULL` pointer otherwise) is passed to the Jacobian-times-vector setup and product functions each time they are called.

Similarly, if a problem involves a non-identity mass matrix, $M \neq I$, then the ARKSPILS 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 `ARKSpilsMassTimesVecFn()`, and can be specified through a call to the `ARKSpilsSetMassTimes()` routine. As with the user-supplied preconditioner functions, the evaluation and processing of any Jacobian-related data needed by the user's mass-matrix-times-vector function is done in the optional user-supplied function of type `ARKSpilsMassTimesSetupFn` (see the section *User-supplied functions* for specification details).

Finally, as described in the section *Linear iteration error control*, the ARKSPILS 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 `ARKSpilsSetEpsLin()` function.

Table: Optional inputs for ARKSPILS

Optional input	Function name	Default
Jv functions (*jtimes* and *jtsetup*)	*ARKSpilsSetJacTimes()*	DQ, none
Newton linear and nonlinear tolerance ratio	*ARKSpilsSetEpsLin()*	0.05
Newton preconditioning functions	*ARKSpilsSetPreconditioner()*	NULL, NULL
Mv functions (*mtimes* and *mtsetup*)	*ARKSpilsSetMassTimes()*	none, none
Mass matrix linear and nonlinear tolerance ratio	*ARKSpilsSetMassEpsLin()*	0.05
Mass matrix preconditioning functions	*ARKSpilsSetMassPreconditioner()*	NULL, NULL

int **ARKSpilsSetJacTimes** (void* *arkode_mem*, *ARKSpilsJacTimesSetupFn jtsetup*, *ARKSpilsJac-TimesVecFn jtimes*)

Specifies the Jacobian-times-vector setup and product functions.

Arguments:

- *arkode_mem* – pointer to the ARKode 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:

- *ARKSPILS_SUCCESS* if successful.

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL.

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL.

- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

- *ARKSPILS_SUNLS_FAIL* if an error occurred when setting up the Jacobian-vector product in the SUNLinearSolver object used by the ARKSPILS 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*.

This function must be called *after* the ARKSPILS system solver interface has been initialized through a call to *ARKSpilsSetLinearSolver()*.

The function types *ARKSpilsJacTimesSetupFn* and *ARKSpilsJacTimesVecFn* is described in the section *User-supplied functions*.

int **ARKSpilsSetEpsLin** (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 ARKode memory block.

- *eplifac* – linear convergence safety factor (≥ 0.0).

Return value:

- *ARKSPILS_SUCCESS* if successful.

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL.

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL.

- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: Passing a value *eplifac* of 0.0 indicates to use the default value of 0.05.

This function must be called *after* the ARKSPILS system solver interface has been initialized through a call to *ARKSpilsSetLinearSolver()*.

int **ARKSpilsSetPreconditioner** (void* *arkode_mem*, *ARKSpilsPrecSetupFn psetup*, *ARKSpilsPrec-SolveFn psolve*)

Specifies the user-supplied preconditioner setup and solve functions.

Arguments:

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

- *psetup* – user defined preconditioner setup function. Pass `NULL` if no setup is needed.

- *psolve* – user-defined preconditioner solve function.

Return value:

- *ARKSPILS_SUCCESS* if successful.

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`.

- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

- *ARKSPILS_SUNLS_FAIL* if an error occurred when setting up preconditioning in the `SUNLinearSolver` object used by the ARKSPILS interface.

Notes: The default is `NULL` for both arguments (i.e. no preconditioning).

This function must be called *after* the ARKSPILS system solver interface has been initialized through a call to *ARKSpilsSetLinearSolver()*.

Both of the function types *ARKSpilsPrecSetupFn()* and *ARKSpilsPrecSolveFn()* are described in the section *User-supplied functions*.

int **ARKSpilsSetMassTimes** (void* *arkode_mem*, *ARKSpilsMassTimesSetupFn mtsetup*, *ARKSpils-MassTimesVecFn mtimes*, void* *mtimes_data*)

Specifies the mass matrix-times-vector setup and product functions.

Arguments:

- *arkode_mem* – pointer to the ARKode 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:

- *ARKSPILS_SUCCESS* if successful.

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`.

- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was `NULL`.

- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

- *ARKSPILS_SUNLS_FAIL* if an error occurred when setting up the mass-matrix-vector product in the `SUNLinearSolver` object used by the ARKSPILS interface.

Notes: There is no default finite difference quotient for *mtimes*, so if using the ARKSPILS mass matrix solver interface and this routine is not called with non-`NULL` *mtimes*, and error will occur. A user may specify `NULL` for *mtsetup*.

This function must be called *after* the ARKSPILS mass matrix solver interface has been initialized through a call to *ARKSpilsSetMassLinearSolver()*.

The function types *ARKSpilsMassTimesSetupFn* and *ARKSpilsMassTimesVecFn* are described in the section *User-supplied functions*.

int **ARKSpilsSetMassEpsLin** (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 ARKode memory block.

- *eplifac* – linear convergence safety factor (≥ 0.0).

Return value:

- *ARKSPILS_SUCCESS* if successful.

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL.

- *ARKSPILS_MASSMEM_NULL* if the mass matrix solver memory was NULL.

- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

Notes: This function must be called *after* the ARKSPILS mass matrix solver interface has been initialized through a call to *ARKSpilsSetMassLinearSolver()*.

Passing a value *eplifac* of 0.0 indicates to use the default value of 0.05.

int **ARKSpilsSetMassPreconditioner** (void* *arkode_mem*, *ARKSpilsMassPrecSetupFn psetup*, *ARKSpilsMassPrecSolveFn psolve*)
Specifies the mass matrix preconditioner setup and solve functions.

Arguments:

- *arkode_mem* – pointer to the ARKode 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:

- *ARKSPILS_SUCCESS* if successful.

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL.

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL.

- *ARKSPILS_ILL_INPUT* if an input has an illegal value.

- *ARKSPILS_SUNLS_FAIL* if an error occurred when setting up preconditioning in the SUNLinearSolver object used by the ARKSPILS interface.

Notes: This function must be called *after* the ARKSPILS mass matrix solver interface has been initialized through a call to *ARKSpilsSetMassLinearSolver()*.

The default is NULL for both arguments (i.e. no preconditioning).

Both of the function types *ARKSpilsMassPrecSetupFn()* and *ARKSpilsMassPrecSolveFn()* are described in the section *User-supplied functions*.

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	*ARKodeSetRootDirection()*	both
Disable inactive root warnings	*ARKodeSetNoInactiveRootWarn()*	enabled

int **ARKodeSetRootDirection** (void* *arkode_mem*, int* *rootdir*)
> Specifies the direction of zero-crossings to be located and returned.

> **Arguments:**

>> • *arkode_mem* – pointer to the ARKode memory block.

>> • *rootdir* – state array of length *nrtfn*, the number of root functions g_i, as specified in the call to the function *ARKodeRootInit()*. 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 ARKode 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 **ARKodeSetNoInactiveRootWarn** (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 ARKode memory block.

> **Return value:**

>> • *ARK_SUCCESS* if successful

>> • *ARK_MEM_NULL* if the ARKode memory is `NULL`

> **Notes:** ARKode 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), ARKode will issue a warning which can be disabled with this optional input function.

4.5.8 Interpolated output function

An optional function *ARKodeGetDky()* is available to obtain additional output values. This function should only be called after a successful return from *ARKode()*, 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 *ARKode()*. 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 **ARKodeGetDky** (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 $t_n - h_n \leq t \leq t_n$, with t_n as current internal time reached, and h_n is the last internal step size successfully used by the solver. The user may request *k* in the range {0,1,2,3}. This routine uses an interpolating polynomial of degree *max(dord, k)*, where *dord* is the argument provided to *ARKodeSetDenseOrder()*.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 {0,1,2,3}.

- *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 ARKode memory is `NULL`

Notes: It is only legal to call this function after a successful return from `ARKode()`.

A user may access the values t_n and h_n via the functions `ARKodeGetCurrentTime()` and `ARKodeGetLastStep()`, respectively.

4.5.9 Optional output functions

ARKode provides an extensive set of functions that can be used to obtain solver performance information. We organize these into groups:

1. General ARKode output routines are in the subsection *Main solver optional output functions*,

2. ARKode implicit solver output routines are in the subsection *Implicit solver optional output functions*,

3. Output routines regarding root-finding results are in the subsection *Rootfinding optional output functions*,

4. Dense linear solver output routines are in the subsection *Direct linear solver interface optional output functions* and

5. Iterative linear solver output routines are in the subsection *Iterative linear solver interface optional output functions*.

6. General usability routines (e.g. to print the current ARKode 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 the `ARKode()` solver. 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.

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 The function 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*.

Main solver optional output functions

Optional output	Function name
Size of ARKode real and integer workspaces	`ARKodeGetWorkSpace()`
Cumulative number of internal steps	`ARKodeGetNumSteps()`
No. of explicit stability-limited steps	`ARKodeGetNumExpSteps()`
No. of accuracy-limited steps	`ARKodeGetNumAccSteps()`
No. of attempted steps	`ARKodeGetNumStepAttempts()`
No. of calls to *fe* and *fi* functions	`ARKodeGetNumRhsEvals()`
No. of local error test failures that have occurred	`ARKodeGetNumErrTestFails()`
Actual initial time step size used	`ARKodeGetActualInitStep()`
Step size used for the last successful step	`ARKodeGetLastStep()`
Step size to be attempted on the next step	`ARKodeGetCurrentStep()`
Current internal time reached by the solver	`ARKodeGetCurrentTime()`
Current ERK and DIRK Butcher tables	`ARKodeGetCurrentButcherTables()`
Suggested factor for tolerance scaling	`ARKodeGetTolScaleFactor()`
Error weight vector for state variables	`ARKodeGetErrWeights()`
Estimated local truncation error vector	`ARKodeGetEstLocalErrors()`
Single accessor to many statistics at once	`ARKodeGetIntegratorStats()`
Name of constant associated with a return flag	`ARKodeGetReturnFlagName()`

int **ARKodeGetWorkSpace** (void* *arkode_mem*, long int* *lenrw*, long int* *leniw*)
    Returns the ARKode real and integer workspace sizes.

    Arguments:

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

- *lenrw* – the number of `realtype` values in the ARKode workspace.

- *leniw* – the number of integer values in the ARKode workspace.

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

int **ARKodeGetNumSteps** (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 ARKode memory block.

- *nsteps* – number of steps taken in the solver.

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

int **ARKodeGetNumExpSteps** (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 ARKode memory block.

- *expsteps* – number of stability-limited steps taken in the solver.

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetNumAccSteps** (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 ARKode memory block.

- *accsteps* – number of accuracy-limited steps taken in the solver.

　　Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetNumStepAttempts** (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 ARKode memory block.

- *step_attempts* – number of steps attempted by solver.

　　Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetNumRhsEvals** (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 ARKode 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 ARKode 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 **ARKodeGetNumErrTestFails** (void* *arkode_mem*, long int* *netfails*)
　　Returns the number of local error test failures that have occured (so far).

　　Arguments:

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

- *netfails* – number of error test failures.

　　Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetActualInitStep** (void* *arkode_mem*, realtype* *hinused*)
　　Returns the value of the integration step size used on the first step.

Arguments:

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

- *hinused* – actual value of initial step size.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

Notes: Even if the value of the initial integration step was specified by the user through a call to *ARKodeSetInitStep()*, this value may have been changed by ARKode 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 **ARKodeGetLastStep** (void* *arkode_mem*, realtype* *hlast*)
Returns the integration step size taken on the last successful internal step.

Arguments:

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

- *hlast* – step size taken on the last internal step.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetCurrentStep** (void* *arkode_mem*, realtype* *hcur*)
Returns the integration step size to be attempted on the next internal step.

Arguments:

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

- *hcur* – step size to be attempted on the next internal step.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetCurrentTime** (void* *arkode_mem*, realtype* *tcur*)
Returns the current internal time reached by the solver.

Arguments:

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

- *tcur* – current internal time reached.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetCurrentButcherTables** (void* *arkode_mem*, int* *s*, int* *q*, int* *p*, realtype* *Ai*, realtype* *Ae*, realtype* *ci*, realtype* *ce*, realtype* *bi*, realtype* *be*, realtype* *b2i*, realtype* *b2e*)
Returns the explicit and implicit Butcher tables currently in use by the solver.

Arguments:

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

- *s* – number of stages in the method.

- *q* – global order of accuracy of the method.

- *p* – global order of accuracy of the embedding.

- *Ai* – coefficients of DIRK method.

- *Ae* – coefficients of ERK method.

- *ci* – array of implicit stage times.

- *ce* – array of explicit stage times.

- *bi* – array of implicit solution coefficients.

- *be* – array of explicit solution coefficients.

- *b2i* – array of implicit embedding coefficients.

- *b2e* – array of explicit embedding coefficients.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

Notes: The user must allocate space for *Ae* and *Ai* of size `ARK_S_MAX*ARK_S_MAX`, and for *ci*, *ce*, *bi*, *be*, *b2i*, and *b2e* of size `ARK_S_MAX` prior to calling this function.

int **ARKodeGetTolScaleFactor** (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 ARKode memory block.

- *tolsfac* – suggested scaling factor for user-supplied tolerances.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

int **ARKodeGetErrWeights** (void* *arkode_mem*, N_Vector *eweight*)
Returns the current error weight vector.

Arguments:

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

- *eweight* – solution error weights at the current time.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

Notes: The user must allocate space for *eweight*, that will be filled in by this function.

int **ARKodeGetEstLocalErrors** (void* *arkode_mem*, N_Vector *ele*)
Returns the vector of estimated local truncation errors for the current step.

Arguments:

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

- *ele* – vector of estimated local truncation errors.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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 if `ARKode()` returned a non-negative value.

The *ele* vector, together with the *eweight* vector from `ARKodeGetErrWeights()`, 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 RMS 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 **ARKodeGetIntegratorStats** (void* *arkode_mem*, long int* *nsteps*, long int* *expsteps*, long int* *accsteps*, long int* *step_attempts*, long int* *nfe_evals*, long int* *nfi_evals*, long int* *nlinsetups*, long int* *netfails*, realtype* *hinused*, realtype* *hlast*, realtype* *hcur*, realtype* *tcur*)
Returns many of the most useful integrator statistics in a single call.

Arguments:

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

- *nsteps* – number of steps taken in the solver.

- *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.

- *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 ARKode memory was NULL

char ***ARKodeGetReturnFlagName** (long int *flag*)
Returns the name of the ARKode constant corresponding to *flag*.

Arguments:

- *flag* – a return flag from an ARKode function.

Return value: The return value is a string containing the name of the corresponding constant.

Implicit solver optional output functions

Optional output	Function name
No. of calls to linear solver setup function	*ARKodeGetNumLinSolvSetups()*
No. of nonlinear solver iterations	*ARKodeGetNumNonlinSolvIters()*
No. of nonlinear solver convergence failures	*ARKodeGetNumNonlinSolvConvFails()*
Single accessor to all nonlinear solver statistics	*ARKodeGetNonlinSolvStats()*

int **ARKodeGetNumLinSolvSetups** (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 ARKode memory block.

   - *nlinsetups* – number of linear solver setup calls made.

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetNumNonlinSolvIters** (void* *arkode_mem*, long int* *nniters*)
   Returns the number of nonlinear solver iterations performed (so far).

   Arguments:

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

   - *nniters* – number of nonlinear iterations performed.

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetNumNonlinSolvConvFails** (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 ARKode memory block.

   - *nncfails* – number of nonlinear convergence failures.

   Return value:

   - *ARK_SUCCESS* if successful

   - *ARK_MEM_NULL* if the ARKode memory was NULL

int **ARKodeGetNonlinSolvStats** (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 ARKode 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 ARKode memory was `NULL`

Rootfinding optional output functions

Optional output	Function name
Array showing roots found	*ARKodeGetRootInfo()*
No. of calls to user root function	*ARKodeGetNumGEvals()*

int **ARKodeGetRootInfo** (void* *arkode_mem*, int* *rootsfound*)

    Returns an array showing which functions were found to have a root.

    Arguments:

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

- *rootsfound* – array of length *nrtfn* with the indices of the user functions g_i found to have a root. 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 ARKode 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 **ARKodeGetNumGEvals** (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 ARKode memory block.

- *ngevals* – number of calls made to g so far.

    Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode memory was `NULL`

Direct linear solver interface optional output functions

The following optional outputs are available from the ARKDLS 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, and last return value from an ARKDLS 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	*ARKDlsGetWorkSpace()*
Size of mass real and integer workspaces	*ARKDlsGetMassWorkSpace()*
No. of Jacobian evaluations	*ARKDlsGetNumJacEvals()*
No. of mass matrix setups	*ARKDlsGetNumMassSetups()*
No. of mass matrix solves	*ARKDlsGetNumMassSolves()*
No. of mass matrix multiplies	*ARKDlsGetNumMassMult()*
No. of *fi* calls for finite diff. Jacobian evals	*ARKDlsGetNumRhsEvals()*
Last return flag from a linear solver function	*ARKDlsGetLastFlag()*
Last return flag from a mass matrix solver function	*ARKDlsGetLastMassFlag()*
Name of constant associated with a return flag	*ARKDlsGetReturnFlagName()*

int **ARKDlsGetWorkSpace** (void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)
Returns the real and integer workspace used by the ARKDLS linear solver interface.

> **Arguments:**

>> - *arkode_mem* – pointer to the ARKode memory block.

>> - *lenrwLS* – the number of `realtype` values in the ARKDLS workspace.

>> - *leniwLS* – the number of integer values in the ARKDLS workspace.

> **Return value:**

>> - *ARKDLS_SUCCESS* if successful

>> - *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`

>> - *ARKDLS_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 ARKDLS is not included in this report.

int **ARKDlsGetMassWorkSpace** (void* *arkode_mem*, long int* *lenrwMLS*, long int* *leniwMLS*)
Returns the real and integer workspace used by the ARKDLS mass matrix linear solver interface.

> **Arguments:**

>> - *arkode_mem* – pointer to the ARKode memory block.

>> - *lenrwMLS* – the number of `realtype` values in the ARKDLS workspace.

>> - *leniwMLS* – the number of integer values in the ARKDLS workspace.

> **Return value:**

>> - *ARKDLS_SUCCESS* if successful

>> - *ARKDLS_MEM_NULL* if the ARKode memory was `NULL`

>> - *ARKDLS_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 ARKDLS is not included in this report.

int **ARKDlsGetNumJacEvals** (void* *arkode_mem*, long int* *njevals*)
Returns the number of calls made to the ARKDLS Jacobian approximation routine.

> **Arguments:**

>> - *arkode_mem* – pointer to the ARKode memory block.

>> - *njevals* – number of calls to the Jacobian function.

Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

int **ARKDlsGetNumMassSetups** (void* *arkode_mem*, long int* *nmsetups*)
　　Returns the number of calls made to the ARKDLS mass matrix solver 'setup' routine.

　　Arguments:

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

- *nmsetups* – number of calls to the mass matrix solver setup routine.

　　Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

int **ARKDlsGetNumMassSolves** (void* *arkode_mem*, long int* *nmsolves*)
　　Returns the number of calls made to the ARKDLS mass matrix solver 'solve' routine.

　　Arguments:

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

- *nmsolves* – number of calls to the mass matrix solver solve routine.

　　Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

int **ARKDlsGetNumMassMult** (void* *arkode_mem*, long int* *nmmults*)
　　Returns the number of calls made to the ARKDLS mass matrix 'matvec' routine.

　　Arguments:

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

- *nmmults* – number of calls to the mass matrix solver matrix-times-vector routine

　　Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

int **ARKDlsGetNumRhsEvals** (void* *arkode_mem*, long int* *nfevalsLS*)
　　Returns the number of calls made to the user-supplied f_I routine due to the finite difference Jacobian approximation.

　　Arguments:

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

- *nfevalsLS* – the number of calls made to the user-supplied f_I function.

　　Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

Notes: The value of *nfevalsLS* is incremented only if one of the default internal difference quotient functions (dense or banded) is used.

int **ARKDlsGetLastFlag** (void* *arkode_mem*, long int* *lsflag*)
Returns the last return value from an ARKDLS routine.

Arguments:

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

- *lsflag* – the value of the last return flag from an ARKDLS function.

Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

Notes: If the SUNLINSOL_DENSE or SUNLINSOL_BAND setup function failed (ARKode returned ARK_LSETUP_FAIL), 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.

int **ARKDlsGetLastMassFlag** (void* *arkode_mem*, long int* *mlsflag*)
Returns the last return value from an ARKDLS mass matrix solve routine.

Arguments:

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

- *mlsflag* – the value of the last return flag from an ARKDLS mass matrix solver function.

Return value:

- *ARKDLS_SUCCESS* if successful

- *ARKDLS_MEM_NULL* if the ARKode memory was NULL

- *ARKDLS_LMEM_NULL* if the linear solver memory was NULL

Notes: If the SUNLINSOL_DENSE or SUNLINSOL_BAND setup function failed (ARKode returned ARK_LSETUP_FAIL), 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) mass matrix. For all other failures, *lsflag* is negative.

char ***ARKDlsGetReturnFlagName** (long int *lsflag*)
Returns the name of the ARKDLS constant corresponding to *lsflag*.

Arguments:

- *lsflag* – a return flag from an ARKDLS function.

Return value: The return value is a string containing the name of the corresponding constant. If $1 \leq lsflag \leq n$ (LU factorization failed), this routine returns "NONE".

Iterative linear solver interface optional output functions

The following optional outputs are available from the ARKSPILS modules: workspace requirements, 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, number of calls to the implicit right-hand side routine for finite-difference Jacobian-vector product approximation, and last return value from a linear solver 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	*ARKSpilsGetWorkSpace()*
No. of preconditioner evaluations	*ARKSpilsGetNumPrecEvals()*
No. of preconditioner solves	*ARKSpilsGetNumPrecSolves()*
No. of linear iterations	*ARKSpilsGetNumLinIters()*
No. of linear convergence failures	*ARKSpilsGetNumConvFails()*
No. of Jacobian-vector setup evaluations	*ARKSpilsGetNumJTSetupEvals()*
No. of Jacobian-vector product evaluations	*ARKSpilsGetNumJtimesEvals()*
No. of f_i calls for finite diff. Jacobian-vector evals.	*ARKSpilsGetNumRhsEvals()*
Last return from a linear solver function	*ARKSpilsGetLastFlag()*
Size of real and integer mass matrix solver workspaces	*ARKSpilsGetMassWorkSpace()*
No. of mass matrix preconditioner evaluations	*ARKSpilsGetNumMassPrecEvals()*
No. of mass matrix preconditioner solves	*ARKSpilsGetNumMassPrecSolves()*
No. of mass matrix linear iterations	*ARKSpilsGetNumMassIters()*
No. of mass matrix solver convergence failures	*ARKSpilsGetNumMassConvFails()*
No. of mass-matrix-vector setup evaluations	*ARKSpilsGetNumMTSetupEvals()*
No. of mass-matrix-vector product evaluations	*ARKSpilsGetNumMtimesEvals()*
Last return from a mass matrix solver function	*ARKSpilsGetLastMassFlag()*
Name of constant associated with a return flag	*ARKSpilsGetReturnFlagName()*

int **ARKSpilsGetWorkSpace** (void* *arkode_mem*, long int* *lenrwLS*, long int* *leniwLS*)
    Returns the global sizes of the ARKSPILS real and integer workspaces.

    Arguments:

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

- *lenrwLS* – the number of `realtype` values in the ARKSPILS workspace.

- *leniwLS* – the number of integer values in the ARKSPILS workspace.

    Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_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.

    In a parallel setting, the above values are global (i.e. summed over all processors).

int **ARKSpilsGetNumPrecEvals** (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`.

    Arguments:

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

- *npevals* – the current number of calls to *psetup*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

int **ARKSpilsGetNumPrecSolves** (void* *arkode_mem*, long int* *npsolves*)
  Returns the number of calls made to the preconditioner solve function, *psolve*.

  Arguments:

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

  - *npsolves* – the number of calls to *psolve*.

  Return value:

  - *ARKSPILS_SUCCESS* if successful

  - *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

  - *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

int **ARKSpilsGetNumLinIters** (void* *arkode_mem*, long int* *nliters*)
  Returns the cumulative number of linear iterations.

  Arguments:

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

  - *nliters* – the current number of linear iterations.

  Return value:

  - *ARKSPILS_SUCCESS* if successful

  - *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

  - *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

int **ARKSpilsGetNumConvFails** (void* *arkode_mem*, long int* *nlcfails*)
  Returns the cumulative number of linear convergence failures.

  Arguments:

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

  - *nlcfails* – the current number of linear convergence failures.

  Return value:

  - *ARKSPILS_SUCCESS* if successful

  - *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

  - *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

int **ARKSpilsGetNumJTSetupEvals** (void* *arkode_mem*, long int* *njtsetup*)
  Returns the cumulative number of calls made to the Jacobian-vector setup function, *jtsetup*.

  Arguments:

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

  - *njtsetup* – the current number of calls to *jtsetup*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

int **ARKSpilsGetNumJtimesEvals** (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 ARKode memory block.

- *njvevals* – the current number of calls to *jtimes*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

int **ARKSpilsGetNumRhsEvals** (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-vector product approximation.

Arguments:

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

- *nfevalsLS* – the number of calls to the user implicit right-hand side function.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

- *ARKSPILS_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.

int **ARKSpilsGetLastFlag** (void* *arkode_mem*, long int* *lsflag*)
Returns the last return value from an ARKSPILS routine.

Arguments:

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

- *lsflag* – the value of the last return flag from an ARKSPILS function.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

- *ARKSPILS_LMEM_NULL* if the linear solver memory was NULL

Notes: If the ARKSPILS setup function failed (*ARKode()* returned *ARK_LSETUP_FAIL*), then *lsflag* will be *SUNLS_PSET_FAIL_UNREC*, *SUNLS_ASET_FAIL_UNREC* or *SUNLS_PACKAGE_FAIL_UNREC*.

If the ARKSPILS solve function failed (*ARKode()* 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_FAIL_UNREC*, indicating an unrecoverable failure in the Jv function; *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 ***ARKSpilsGetReturnFlagName** (long int *lsflag*)
Returns the name of the ARKSPILS constant corresponding to *lsflag*.

Arguments:

- *lsflag* – a return flag from an ARKSPILS function.

Return value: The return value is a string containing the name of the corresponding constant.

int **ARKSpilsGetMassWorkSpace** (void* *arkode_mem*, long int* *lenrwMLS*, long int* *leniwMLS*)
Returns the global sizes of the ARKSPILS real and integer workspaces.

Arguments:

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

- *lenrwMLS* – the number of `realtype` values in the ARKSPILS workspace.

- *leniwMLS* – the number of integer values in the ARKSPILS workspace.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_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.

In a parallel setting, the above values are global (i.e. summed over all processors).

int **ARKSpilsGetNumMassPrecEvals** (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 ARKode memory block.

- *nmpevals* – the current number of calls to *psetup*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKSpilsGetNumMassPrecSolves** (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 ARKode memory block.

- *nmpsolves* – the number of calls to *psolve*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKSpilsGetNumMassIters** (void* *arkode_mem*, long int* *nmiters*)
Returns the cumulative number of mass matrix solver iterations.

Arguments:

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

- *nmiters* – the current number of mass matrix solver linear iterations.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKSpilsGetNumMassConvFails** (void* *arkode_mem*, long int* *nmcfails*)
Returns the cumulative number of mass matrix solver convergence failures.

Arguments:

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

- *nmcfails* – the current number of mass matrix solver convergence failures.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKSpilsGetNumMTSetupEvals** (void* *arkode_mem*, long int* *nmtsetup*)
Returns the cumulative number of calls made to the mass-matrix-vector setup function, *mtsetup*.

Arguments:

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

- *nmtsetup* – the current number of calls to *mtsetup*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKSpilsGetNumMtimesEvals** (void* *arkode_mem*, long int* *nmvevals*)
Returns the cumulative number of calls made to the mass-matrix-vector product function, *mtimes*.

Arguments:

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

- *nmvevals* – the current number of calls to *mtimes*.

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory was `NULL`

int **ARKSpilsGetLastMassFlag** (void* *arkode_mem*, long int* *msflag*)
    Returns the last return value from an ARKSPILS mass matrix solver routine.

    Arguments:

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

- *msflag* – the value of the last return flag from an ARKSPILS mass matrix solver function.

    Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was `NULL`

- *ARKSPILS_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 *ARKSpilsGetLastFlag()*.

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(s), 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 ARKode and/or specific Runge-Kutta methods.

Optional routine	Function name
Output all ARKode solver parameters	*ARKodeWriteParameters()*
Retrieve a given Butcher table by its unique name	*ARKodeLoadButcherTable()*
Output the current Butcher table(s)	*ARKodeWriteButcher()*
Test the analytical order of accuracy for a Butcher table	*ARKodeTestButcherTable()*
Test the analytical order for a pair of Butcher tables	*ARKodeTestButcherTables()*

int **ARKodeWriteParameters** (void* *arkode_mem*, FILE **fp*)
    Outputs all solver parameters to the provided file pointer.

    Arguments:

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

- *fp* – pointer to use for printing the solver parameters

    Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode 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 **ARKodeLoadButcherTable** (int *imethod*, int **s*, int **q*, int **p*, realtype **A*, realtype **b*, realtype **c*,
                        realtype **b2*)
    Retrieves a specified Butcher table. The array *A* must be declared of size `ARK_S_MAX*ARK_S_MAX`, and the
    arrays *c*, *b* and *b2* should all have length at least `ARK_S_MAX`.

    Arguments:

- *imethod* – integer input specifying the given Butcher table – valid values match those for the functions *ARKodeSetERKTableNum()* and *ARKodeSetIRKTableNum()*

- *s* – integer number of stages for the method (output)

- *q* – integer order of the method (output)

- *p* – integer order of the embedding (output)

- *A* – realtype Butcher table coefficients (output)

- *b* – realtype root node coefficients (output)

- *c* – realtype canopy node coefficients (output)

- *b2* – realtype embedding coefficients (output)

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_ILL_INPUT* if *imethod* was invalid

int **ARKodeWriteButcher** (void* *arkode_mem*, FILE **fp*)
Outputs the current Butcher table(s) to the provided file pointer.

Arguments:

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

- *fp* – pointer to use for printing the Butcher table(s)

Return value:

- *ARKSPILS_SUCCESS* if successful

- *ARKSPILS_MEM_NULL* if the ARKode memory was NULL

Notes: The *fp* argument can be stdout or stderr, or it may point to a specific file created using fopen.

If ARKode 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.

int **ARKodeTestButcherTable** (realtype **A*, realtype **b*, realtype **c*, int *s*, booleantype *printstats*)
Checks the order conditions for a single Butcher table (ERK or DIRK or even IRK), and returns the integer order of accuracy for the method (up to 6th order). It will optionally output detailed information on the tests that pass/fail, depending on the *printstats* argument.

Arguments:

- *A* – realtype Butcher table of allocated size ARK_S_MAX*ARK_S_MAX, although not all of this space need be used.

- *b* – realtype root node coefficients for the Butcher table

- *c* – realtype canopy node coefficients for the Butcher table

- *s* – integer number of stages in this Butcher table

- *printstats* – booleantype flag denoting whether to output detailed test statistics to stdout

Return value:

- the integer order of accuracy for the method (0 through 6); a return value of −1 indicates that the row sum condition is not satisfied (see below).

Notes: The only assumption is that the row-sum condition must be satisfied, i.e. that

$$c_i = \sum_{j=1}^{s} A_{i,j}, \quad i = 1, \ldots, s,$$

no other "simplifying assumptions" are made.

int **ARKodeTestButcherTables** (realtype *A1*, realtype *b1*, realtype *c1*, realtype *A2*, realtype *b2*,
    realtype *c2*, int *s*, booleantype *printstats*)
Checks the order conditions for a pair of Butcher tables that comprise a 2-additive Runge-Kutta method, and
returns the integer order of accuracy for the coupled method (up to 6th order). No assumptions are made about
whether each method is explicit or implicit. It will optionally output detailed information on the tests that
pass/fail, depending on the *printstats* argument.

Arguments:

- *A1* – realtype Butcher table of allocated size `ARK_S_MAX*ARK_S_MAX`, although not all of this
  space need be used.

- *b1* – realtype root node coefficients for the Butcher table

- *c1* – realtype root node coefficients for the Butcher table

- *A2* – realtype Butcher table of allocated size `ARK_S_MAX*ARK_S_MAX`, although not all of this
  space need be used.

- *b2* – realtype root node coefficients for the Butcher table

- *c2* – realtype canopy coefficients for the Butcher table

- *s* – integer number of stages in this Butcher table

- *printstats* – booleantype flag denoting whether to output detailed test statistics to `stdout`

Return value:

- the integer order of accuracy for the coupled method (0 through 6); a return value of -1 indicates that
  one or both of the row sum conditions is not satisfied (see below).

Notes: The only assumptions are that the row-sum condition for each component method must be satisfied, i.e.
that

$$c_i^1 = \sum_{j=1}^{s} A_{i,j}^1, \quad i = 1, \ldots, s$$

and

$$c_i^2 = \sum_{j=1}^{s} A_{i,j}^2, \quad i = 1, \ldots, s,$$

where c^i and A^i, $i = \{1,2\}$ denote the coefficients for the respective table; no other "simplifying assumptions"
are made.

4.5.10 ARKode reinitialization function

The function `ARKodeReInit()` reinitializes the main ARKode solver for the solution of a new problem, where
a prior call to `ARKodeInit()` has been made. The new problem must have the same size as the previous one.
`ARKodeReInit()` performs the same input checking and initializations that `ARKodeInit()` does, but does no
memory allocation as it assumes that the existing internal memory is sufficient for the new problem. A call to
`ARKodeReInit()` deletes the solution history that was stored internally during the previous integration. Following
a successful call to `ARKodeReInit()`, call `ARKode()` again for the solution of the new problem.

The use of *ARKodeReInit()* 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. 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 ARKDLS or ARKSPILS 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 *ARKodeReInit()* 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 *ARKodeReInit()*. 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 extention 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 **ARKodeReInit** (void* *arkode_mem*, *ARKRhsFn fe*, *ARKRhsFn fi*, realtype *t0*, N_Vector *y0*)
> Provides required problem specifications and reinitializes ARKode.

> **Arguments:**
>> - *arkode_mem* – pointer to the ARKode memory block.
>> - *fe* – the name of the C function (of type *ARKRhsFn()*) defining the explicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$.
>> - *fi* – the name of the C function (of type *ARKRhsFn()*) defining the implicit portion of the right-hand side function in $\dot{y} = f_E(t, y) + f_I(t, y)$.
>> - *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 ARKode memory was `NULL`
>> - *ARK_MEM_FAIL* if a memory allocation failed
>> - *ARK_ILL_INPUT* if an argument has an illegal value.

> **Notes:** If an error occurred, *ARKodeReInit()* also sends an error message to the error handler function.

4.5.11 ARKode 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 ARKode integrator may be "resized" between integration steps, through calls to the *ARKodeResize()* function. This function modifies ARKode'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 *ARKodeResize()* remain valid after the call. If instead the dynamics should be recomputed from scratch, the ARKode memory structure should be deleted with a call to *ARKodeFree()*, and recreated with calls to *ARKodeCreate()* and *ARKodeInit()*.

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 ARKode 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 **ARKodeResize** (void* *arkode_mem*, N_Vector *ynew*, realtype *hscale*, realtype *t0*, *ARKVecResizeFn resize*, void* *resize_data*)

Re-initializes ARKode with a different state vector but with comparable dynamical time scale.

Arguments:

- *arkode_mem* – pointer to the ARKode 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 ARKode vectors.

Return value:

- *ARK_SUCCESS* if successful

- *ARK_MEM_NULL* if the ARKode 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, *ARKodeResize()* also sends an error message to the error handler function.

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 *ARKodeResize()*. Moreover, the existing ARKDLS or ARK-SPILS interface should then be deleted and recreated by attaching the updated `SUNLinearSolver` (and possibly `SUNMatrix`) object(s) through calls to *ARKDlsSetLinearSolver()*, *ARKSpilsSetLinearSolver()*, *ARKDlsSetMassLinearSolver()* and *ARKSpilsSetMassLinearSolver()*.

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.

Resizing the absolute tolerance array

If using array-valued absolute tolerances, the absolute tolerance vector will be invalid after the call to *ARKodeResize()*, so the new absolute tolerance vector should be re-set **following** each call to *ARKodeResize()* through a new call to *ARKodeSVtolerances()* (and similarly to *ARKodeResVtolerance()* if that was used for the original problem).

If scalar-valued tolerances or a tolerance function was specified through either *ARKodeSStolerances()* or *ARKodeWFtolerances()*, then these will remain valid. and no further action is necessary.

4.6 User-supplied functions

The user-supplied functions for ARKode 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 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 *ARKodeResize()* (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 teh dependent variable vector, $y(t)$.

- *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 *ARKodeSetUserData()*.

Return value: An *ARKRhsFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode 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 ARKode. 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 nonnegative value is physically meaningful). If such a return is made, ARKode 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 ARKode 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 ARKode 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 *ARKodeSetErrFile()*), 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 ARKode and is sub-modules.

Arguments:

- *error_code* – the error code.

- *module* – the name of the ARKode 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 *ARKodeSetErrHandlerFn()*.

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 $\|v\|_{WRMS} = \left(\frac{1}{n} \sum_{i=1}^{n} (ewt_i \ v_i)^2 \right)^{1/2}$. These weights will be used in place of those defined in the section *Choice of norm*.

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 *ARKodeSetUserData()*.

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 ARKode.

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 $\|v\|_{WRMS} = \left(\frac{1}{n} \sum_{i=1}^{n} (rwt_i \; v_i)^2 \right)^{1/2}$. These weights will be used in place of those defined in the section *Choice of norm*.

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 `ARKodeSetUserData()`.

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 ARKode.

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, $y(t)$.

- *t* – the current value of the independent variable.

- *h1* – the current step size, $t_m - t_{m-1}$.

- *h2* – the previous step size, $t_{m-1} - t_{m-2}$.

- *h3* – the step size $t_{m-2} - t_{m-3}$.

- *e1* – the error estimate from the current step, m.

- *e2* – the error estimate from the previous step, $m-1$.

- *e3* – the error estimate from the step $m-2$.

- *q* – the global order of accuracy for the integration method.

- *p* – the global order of accuracy for the embedding.

- *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 `ARKodeSetAdaptivityFn()`.

Return value: An *ARKAdaptFn* function should return 0 if it successfuly 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 ARKode. 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, $y(t)$.

- *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 `ARKodeSetStabilityFn()`.

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 $hstab \leq 0.0$, then ARKode will assume that there is no explicit stability restriction on the time step size.

4.6.7 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, $y(t)$.

- *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 `ARKodeSetUserData()`.

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 ARKode returns *ARK_RTFUNC_FAIL*).

Notes: Allocation of memory for *gout* is handled within ARKode.

4.6.8 Jacobian information (direct method Jacobian)

If the direct linear solver interface is used (i.e., `ARKDlsSetLinearSolver()` is called in the section *A skeleton of the user's main program*), the user may provide a function of type `ARKDlsJacFn` to provide the Jacobian approximation.

typedef int **(*ARKDlsJacFn)** (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 $J = \frac{\partial f_I}{\partial y}$ (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 $f_I(t, y)$.
>
> - *Jac* – the output Jacobian matrix.
>
> - *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to `ARKodeSetUserData()`.
>
> - *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type `N_Vector` which can be used by an ARKDlsacFn as temporary storage or work space.

> **Return value:** An *ARKDlsJacFn* function should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode will attempt to correct, while ARKDLS sets *last_flag* to *ARKDLS_JACFUNC_RECVR*), or a negative value if it failed unrecoverably (in which case the integration is halted, `ARKode()` returns *ARK_LSETUP_FAIL* and ARKDLS sets *last_flag* to *ARKDLS_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).

> 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*.

> If the user's `ARKDlsJacFn` 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 ARKodeGet* 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*.

4.6.9 Jacobian information (matrix-vector product)

If the ARKSPILS solver interface is selected (i.e. *ARKSpilsSetLinearSolver()* is called in the section *A skeleton of the user's main program*), the user may provide a function of type *ARKSpilsJacTimesVecFn* 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 **(*ARKSpilsJacTimesVecFn)** (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 $Jv = \left(\frac{\partial f_I}{\partial y} \right) v$ (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 $f_I(t, y)$.

- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to *ARKodeSetUserData()*.

- *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 *ARKSpilsJacTimesVecFn* 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 ARKodeGet* 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.

4.6.10 Jacobian information (matrix-vector setup)

If the user's Jacobian-times-vector requires that any Jacobian-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type *ARKSpilsJacTimesSetupFn*, defined as follows:

typedef int **(*ARKSpilsJacTimesSetupFn)** (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 *ARKodeSetUserData()*.

   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 *ARKSpilsJacTimesSetupFn* 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 ARKodeGet* 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 Preconditioning (linear system solution)

If preconditioning is used with the ARKSPILS solver interface, then the user must provide a function of type *ARKSpilsPrecSolveFn* 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 Newton matrix $A = M - \gamma J$, where M is the mass matrix (typically $M = I$ unless working in a finite-element setting) and $J = \frac{\partial f_I}{\partial y}$ If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate A.

typedef int **(*ARKSpilsPrecSolveFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, N_Vector *r*, N_Vector *z*, re-
altype *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 $A = M - \gamma J$.

- *delta* – an input tolerance to be used if an iterative method is employed in the solution. In that case,
the resdual 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
ARKodeGetErrWeights().

- *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
ARKodeSetUserData().

    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).

4.6.12 Preconditioning (Jacobian data)

If the user's preconditioner requires that any data be preprocessed or evaluated, then these actions need to occur within
a user-supplied function of type *ARKSpilsPrecSetupFn*.

typedef int **(*ARKSpilsPrecSetupFn)** (realtype *t*, N_Vector *y*, N_Vector *fy*, booleantype *jok*, boolean-
type* *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 $A = M - \gamma J$.

- *user_data* – a pointer to user data, the same as the *user_data* parameter that was passed to
ARKodeSetUserData().

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 $A = M - \gamma J$.

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 ODE right-hand side.

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 *ARKSpilsPrecSetupFn* 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 ARKodeGet* 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`.

4.6.13 Mass matrix information (direct method mass matrix)

If the direct mass matrix linear solver interface is used (i.e., *ARKDlsSetMassLinearSolver()* is called in the section *A skeleton of the user's main program*), the user must provide a function of type *ARKDlsMassFn* to provide the mass matrix approximation.

typedef int **(*ARKDlsMassFn)** (realtype *t*, SUNMatrix *M*, void* *user_data*, N_Vector *tmp1*, N_Vector *tmp2*,
N_Vector *tmp3*)
This function computes the mass matrix M (or an approximation to it).

Arguments:

- *N* – the size of the ODE system.

- *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 *ARKodeSetUserData()*.

- *tmp1*, *tmp2*, *tmp3* – pointers to memory allocated to variables of type N_Vector which can be used by an ARKDlsDenseMassFn as temporary storage or work space.

Return value: An *ARKDlsMassFn* function should return 0 if successful, or a negative value if it failed unrecoverably (in which case the integration is halted, *ARKode()* returns *ARK_MASSSETUP_FAIL* and ARKDLS sets *last_flag* to *ARKDLS_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 information (direct method Jacobian)*, 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 information (direct method Jacobian)*, 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*.

4.6.14 Mass matrix information (matrix-vector product)

If the ARKSPILS solver interface is selected (i.e. *ARKSpilsSetMassLinearSolver()* is called in the section *A skeleton of the user's main program*), the user must provide a function of type *ARKSpilsMassTimesVecFn* in the following form, to compute matrix-vector products Mv.

typedef int **(*ARKSpilsMassTimesVecFn)** (N_Vector *v*, N_Vector *Mv*, realtype *t*, void* *mtimes_data*)
> This function computes the product $M * v$ (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 *ARKSpilsSetMassTimes()*.

> **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.

4.6.15 Mass matrix information (matrix-vector setup)

If the user's mass-matrix-times-vector requires that any mass matrix-related data be preprocessed or evaluated, then this needs to be done in a user-supplied function of type *ARKSpilsMassTimesSetupFn*, defined as follows:

typedef int **(*ARKSpilsMassTimesSetupFn)** (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 `ARKSpilsSetMassTimes()`.

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 ARKSPILS mass matrix solver interface, in which case the integration is halted.

4.6.16 Mass matrix preconditioning (linear system solution)

If preconditioning is used with the ARKSPILS mass matrix solver interface, then the user must provide a function of type `ARKSpilsMassPrecSolveFn` 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. If preconditioning is done on both sides, the product of the two preconditioner matrices should approximate M.

typedef int **(*ARKSpilsMassPrecSolveFn)** (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 resdual 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 `ARKodeGetErrWeights()`.

- *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 `ARKodeSetUserData()`.

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).

4.6.17 Mass matrix preconditioning (mass matrix data)

If the user's mass matrix preconditioner requires that any problem data be preprocessed or evaluated, then these actions need to occur within a user-supplied function of type `ARKSpilsMassPrecSetupFn`.

typedef int **(*ARKSpilsMassPrecSetupFn)** (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 `ARKodeSetUserData()`.

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.

4.6.18 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 ARKode integrator may be "resized" between integration steps, through calls to the `ARKodeResize()` 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 `ARKodeResize()` so that all internal ARKode 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 `ARKodeResize()`.

>   **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 ARKode 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: a banded preconditioner for serial 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 the ARKSPILS iterative linear solver interface, in a serial 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 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.

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 remainder 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 ARKode object*

5. *Initialize ARKode solver*

6. *Specify integration tolerances*

7. *Set optional inputs*

8. Create iterative linear solver object

   When creating the iterative linear solver object, specify the type of preconditioning (PREC_LEFT or PREC_RIGHT) to use.

9. *Set linear solver optional inputs*

10. *Attach linear solver module*

11. 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.

12. *Set linear solver interface optional inputs*

    Note that the user should not overwrite the preconditioner setup function or solve function through calls to the ARKSpilsSet* optional input functions.

13. *Specify rootfinding problem*

14. *Advance solution in time*

15. Get optional outputs

    Additional optional outputs associated with ARKBANDPRE are available by way of the two routines described below, *ARKBandPrecGetWorkSpace()* and *ARKBandPrecGetNumRhsEvals()*.

16. *Free solver memory*

17. *Deallocate memory for solution vector*

We note that the ARKBANDPRE preconditioner may not be used for problems involving a non-identity mass matrix, $M \neq I$.

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 ARKode 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:

- *ARKSPILS_SUCCESS* if no errors occurred

- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`

- *ARKSPILS_ILL_INPUT* if an input has an illegal value

- *ARKSPILS_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 ARKode memory block.

- *lenrwLS* – the number of `realtype` values in the ARKBANDPRE workspace.

- *leniwLS* – the number of integer values in the ARKBANDPRE workspace.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred

- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`

- *ARKSPILS_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 `ARKSpilsGetWorkspace()`.

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 ARKode memory block.

- *nfevalsBP* – number of calls to f_I

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred

- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`

- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`

- *ARKSPILS_PMEM_NULL* if the preconditioner memory is `NULL`

Notes: The counter *nfevalsBP* is distinct from the counter *nfevalsLS* returned by the corresponding function *ARKSpilsGetNumRhsEvals()* and also from *nfi_evals* returned by *ARKodeGetNumRhsEvals()*. 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 *ARKodeGetNumRhsEvals()*.

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]* and is included in a software module within the ARKode package. This module works with the parallel vector module NVECTOR_PARALLEL and is usable with any of the Krylov iterative linear solvers through the ARKSPILS 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. As with the rest of ARKode, we assume here that the ODE system is written as

$$M\dot{y} = f_E(t, y) + f_I(t, y),$$

where f_I corresponds to the ODE components to be treated implicitly. 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

$$g(t, y) = [g_1(t, \bar{y}_1), g_2(t, \bar{y}_2), \dots, g_Q(t, \bar{y}_Q)]^T,$$

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

$$P = \text{diag}[P_1, P_2, \dots, P_Q]$$

where

$$P_q \approx M - \gamma J_q$$

and where J_q is a difference quotient approximation to $\frac{\partial g_q}{\partial \bar{y}_q}$. 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.

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 interprocess 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 `ARKodeSetUserData()` 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 `ARKodeSetUserData()`.
>
> **Return value:** An *ARKLocalFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and `ARKode()` will return *ARK_LSETUP_FAIL*).
>
> **Notes:** This function should assume that all interprocess 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 interprocess 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 *ARKodeSetUserData()*.

Return value: An *ARKCommFn* should return 0 if successful, a positive value if a recoverable error occurred (in which case ARKode will attempt to correct), or a negative value if it failed unrecoverably (in which case the integration is halted and *ARKode()* 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).

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 ARKode object*

5. *Initialize ARKode solver*

6. *Specify integration tolerances*

7. *Set optional inputs*

8. Create iterative linear solver object

   When creating the iterative linear solver object, specify the type of preconditioning (PREC_LEFT or PREC_RIGHT) to use.

9. Set linear solver optional inputs

10. *Attach linear solver module*

11. 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, respectivelyl.

12. *Set the linear solver interface optional inputs*

Note that the user should not overwrite the preconditioner setup function or solve function through calls to ARKSPILS optional input functions.

11. *Specify rootfinding problem*

12. *Advance solution in time*

13. *Get optional outputs*

    Additional optional outputs associated with ARKBBDPRE are available through the routines *ARKBBDPrecGetWorkSpace()* and *ARKBBDPrecGetNumGfnEvals()*.

14. *Free solver memory*

15. *Deallocate memory for solution vector*

16. *Finalize MPI*

We note that the ARKBBDPRE preconditioner may not be used for problems involving a non-identity mass matrix, $M \neq I$.

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*, *ARK-CommFn cfn*)
Initializes and allocates (internal) memory for the ARKBBDPRE preconditioner.

> **Arguments:**
>
> - *arkode_mem* – pointer to the ARKode 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 interprocess communication required for the computation of $g(t, y)$.
>
> **Return value:**
>
> - *ARKSPILS_SUCCESS* if no errors occurred
>
> - *ARKSPILS_MEM_NULL* if the integrator memory is NULL
>
> - *ARKSPILS_LMEM_NULL* if the linear solver memory is NULL
>
> - *ARKSPILS_ILL_INPUT* if an input has an illegal value
>
> - *ARKSPILS_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 reinitialization 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 `ARKodeReInit()` to re-initialize ARKode 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 ``SUNLinearSolver module, and/or one or more of the corresponding `ARKSpilsSet***` 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 ARKode 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:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`
- *ARKSPILS_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 ARKode memory block.
- *lenrwBBDP* – the number of `realtype` values in the ARKBBDPRE workspace.
- *leniwBBDP* – the number of integer values in the ARKBBDPRE workspace.

Return value:

- *ARKSPILS_SUCCESS* if no errors occurred
- *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`
- *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`

> • *ARKSPILS_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 *ARKSpilsGetWorkSpace()*.

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 ARKode memory block.

   > • *ngevalsBBDP* – the number of calls made to the user-supplied *gloc* function.

   Return value:

   > • *ARKSPILS_SUCCESS* if no errors occurred

   > • *ARKSPILS_MEM_NULL* if the integrator memory is `NULL`

   > • *ARKSPILS_LMEM_NULL* if the linear solver memory is `NULL`

   > • *ARKSPILS_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 ARKode output and *npsolves* and *nfevalsLS* are linear solver optional outputs (see the table *Iterative linear solver interface optional output functions*).

FARKODE, AN INTERFACE MODULE FOR FORTRAN APPLICATIONS

The FARKODE interface module is a package of C functions which support the use of the ARKODE solver 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. This package provides the necessary interfaces to ARKODE for all supplied serial and parallel NVECTOR implementations.

5.1 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*).

5.2 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

Details on the Fortran interface to ARKode are provided in the following sub-sections:

5.2.1 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 ARKode 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 *SUNBandLinearSolver()*.

- `FSUNDENSELINSOLINIT()` (defined by SUNLINSOL_DENSE) interfaces to *SUNDenseLinearSolver()*.

- `FSUNKLUINIT()` (defined by SUNLINSOL_KLU) interfaces to *SUNKLU()*.

- `FSUNKLUREINIT()` (defined by SUNLINSOL_KLU) interfaces to `SUNKLUReinit()`.

- `FSUNLAPACKBANDINIT()` (defined by SUNLINSOL_LAPACKBAND) interfaces to *SUNLapackBand()*.

- `FSUNLAPACKDENSEINIT()` (defined by SUNLINSOL_LAPACKDENSE) interfaces to *SUNLapackDense()*.

- `FSUNPCGINIT()` (defined by SUNLINSOL_PCG) interfaces to *SUNPCG()*.

- `FSUNSPBCGSINIT()` (defined by SUNLINSOL_SPBCGS) interfaces to *SUNSPBCGS()*.

- `FSUNSPFGMRINIT()` (defined by SUNLINSOL_SPFGMR) interfaces to *SUNSPFGMR()*.

- `FSUNSPGMRINIT()` (defined by SUNLINSOL_SPGMR) interfaces to *SUNSPGMR()*.

- `FSUNSPTFQMRINIT()` (defined by SUNLINSOL_SPTFQMR) interfaces to *SUNSPTFQMR()*.

- `FSUNSUPERLUMTINIT()` (defined by SUNLINSOL_SUPERLUMT) interfaces to *SUNSuperLUMT()*.

Interface to the main ARKODE module

- *FARKMALLOC()* interfaces to *ARKodeCreate()*, *ARKodeSetUserData()*, and *ARKodeInit()*, as well as one of *ARKodeSStolerances()* or *ARKodeSVtolerances()*.

- *FARKREINIT()* interfaces to *ARKodeReInit()*.

- *FARKRESIZE()* interfaces to *ARKodeResize()*.

- *FARKSETIIN()* and *FARKSETRIN()* interface to the ARKodeSet* functions (see *Optional input functions*).

- *FARKEWTSET()* interfaces to *ARKodeWFtolerances()*.

- *FARKADAPTSET()* interfaces to *ARKodeSetAdaptivityFn()*.

- *FARKEXPSTABSET()* interfaces to *ARKodeSetStabilityFn()*.

- *FARKSETERKTABLE()* interfaces to *ARKodeSetERKTable()*.

- *FARKSETIRKTABLE()* interfaces to *ARKodeSetIRKTable()*.

- *FARKSETARKTABLES()* interfaces to *ARKodeSetARKTables()*.

- *FARKSETRESTOLERANCE()* interfaces to either *ARKodeResStolerance()* and *ARKodeResVtolerance()*

- *FARKODE()* interfaces to *ARKode()*, the ARKodeGet* 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 *ARKodeGetDky()*.

- *FARKGETERRWEIGHTS()* interfaces to *ARKodeGetErrWeights()*.

- *FARKGETESTLOCALERR()* interfaces to *ARKodeGetEstLocalErrors()*.

- *FARKFREE()* interfaces to *ARKodeFree()*.

Interface to the system linear solver interfaces

- *FARKDLSINIT()* interfaces to *ARKDlsSetLinearSolver()*.

- *FARKDENSESETJAC()* interfaces to *ARKDlsSetJacFn()*.

- *FARKBANDSETJAC()* interfaces to *ARKDlsSetJacFn()*.

- *FARKSPARSESETJAC()* interfaces to *ARKDlsSetJacFn()*.

- *FARKSPILSINIT()* interfaces to *ARKSpilsSetLinearSolver()*

- *FARKSPILSSETEPSLIN()* interfaces to *ARKSpilsSetEpsLin()*.

- *FARKSPILSSETJAC()* interfaces to *ARKSpilsSetJacTimes()*.

- *FARKSPILSSETPREC()* interfaces to *ARKSpilsSetPreconditioner()*.

Interface to the mass matrix linear solver interfaces

- *FARKDLSMASSINIT()* interfaces to *ARKDlsSetMassLinearSolver()*.

- *FARKDENSESETMASS()* interfaces to *ARKDlsSetMassFn()*.

- *FARKBANDSETMASS()* interfaces to *ARKDlsSetMassFn()*.

- *FARKSPARSESETMASS()* interfaces to *ARKDlsSetMassFn()*.

- *FARKSPILSMASSINIT()* interfaces to *ARKSpilsSetMassLinearSolver()*.

- *FARKSPILSSETMASSEPSLIN()* interfaces to *ARKSpilsSetMassEpsLin()*.

- *FARKSPILSSETMASS()* interfaces to *ARKSpilsSetMassTimes()*.

- *FARKSPILSSETMASSPREC()* interfaces to *ARKSpilsSetMassPreconditioner()*.

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 ARKode, 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)	ARKode 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)	ARKode interface function type	FARKODE "activation" routine
FARKDJAC()	*ARKDlsJacFn()*	*FARKDENSESETJAC()*
FARKBJAC()	*ARKDlsJacFn()*	*FARKBANDSETJAC()*
FARKSPJAC()	*ARKDlsJacFn()*	*FARKSPARSESETJAC()*
FARKDMASS()	*ARKDlsMassFn()*	*FARKDENSESETMASS()*
FARKBMASS()	*ARKDlsMassFn()*	*FARKBANDSETMASS()*
FARKSPMASS()	*ARKDlsMassFn()*	*FARKSPARSESETMASS()*
FARKPSET()	*ARKSpilsPrecSetupFn()*	*FARKSPILSSETPREC()*
FARKPSOL()	*ARKSpilsPrecSolveFn()*	*FARKSPILSSETPREC()*
FARKJTSETUP()	*ARKSpilsJacTimesSetupFn()*	*FARKSPILSSETJAC()*
FARKJTIMES()	*ARKSpilsJacTimesVecFn()*	*FARKSPILSSETJAC()*
FARKMASSPSET()	*ARKSpilsMassPrecSetupFn()*	*FARKSPILSSETMASSPREC()*
FARKMASSPSOL()	*ARKSpilsMassPrecSolveFn()*	*FARKSPILSSETMASSPREC()*
FARKMTSETUP()	*ARKSpilsMassTimesSetupFn()*	*FARKSPILSSETMASS()*
FARKMTIMES()	*ARKSpilsMassTimesVecFn()*	*FARKSPILSSETMASS()*
FARKEWT()	*ARKEwtFn()*	*FARKEWTSET()*
FARKADAPT()	*ARKAdaptFn()*	*FARKADAPTSET()*
FARKEXPSTAB()	*ARKExpStabFn()*	*FARKEXPSTABSET()*

5.2.2 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 ARKode 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*, 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 $M(t)$ is the possibly time-dependent 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 *Linear Solver Data Structures*. 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 *Linear Solver Data Structures*.

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 *Linear Solver Data Structures*.

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 *Linear Solver Data Structures*.

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 ARKode solver, providing interfaces to the C routines *ARKodeCreate()*, *ARKodeSetUserData()*, and *ARKodeInit()*, as well as one of *ARKodeSStolerances()* or *ARKodeSVtolerances()*.

    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 ARKode 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 ARKode's C interface, that provides separate functions for setting each optional input, FARKODE uses only two functions, that accept keywords to specify which optional input should be set to the provided value. These routines are *FARKSETIIN()* and *FARKSETRIN()*, 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	ARKode routine
ORDER	*ARKodeSetOrder()*
DENSE_ORDER	*ARKodeSetDenseOrder()*
LINEAR	*ARKodeSetLinear()*
NONLINEAR	*ARKodeSetNonlinear()*
FIXEDPOINT	*ARKodeSetFixedPoint()*
NEWTON	*ARKodeSetNewton()*
EXPLICIT	*ARKodeSetExplicit()*
IMPLICIT	*ARKodeSetImplicit()*
IMEX	*ARKodeSetImEx()*
IRK_TABLE_NUM	*ARKodeSetIRKTableNum()*
ERK_TABLE_NUM	*ARKodeSetERKTableNum()*
ARK_TABLE_NUM *(a)*	*ARKodeSetARKTableNum()*
MAX_NSTEPS	*ARKodeSetMaxNumSteps()*
HNIL_WARNS	*ARKodeSetMaxHnilWarns()*
PREDICT_METHOD	*ARKodeSetPredictorMethod()*
MAX_ERRFAIL	*ARKodeSetMaxErrTestFails()*
MAX_CONVFAIL	*ARKodeSetMaxConvFails()*
MAX_NITERS	*ARKodeSetMaxNonlinIters()*
ADAPT_SMALL_NEF	*ARKodeSetSmallNumEFails()*
LSETUP_MSBP	*ARKodeSetMaxStepsBetweenLSet()*

(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 file `arkode.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	ARKode routine
INIT_STEP	*ARKodeSetInitStep()*
MAX_STEP	*ARKodeSetMaxStep()*
MIN_STEP	*ARKodeSetMinStep()*
STOP_TIME	*ARKodeSetStopTime()*
NLCONV_COEF	*ARKodeSetNonlinConvCoef()*
ADAPT_CFL	*ARKodeSetCFLFraction()*
ADAPT_SAFETY	*ARKodeSetSafetyFactor()*
ADAPT_BIAS	*ARKodeSetErrorBias()*
ADAPT_GROWTH	*ARKodeSetMaxGrowth()*
ADAPT_ETAMX1	*ARKodeSetMaxFirstGrowth()*
ADAPT_BOUNDS	*ARKodeSetFixedStepBounds()*
ADAPT_ETAMXF	*ARKodeSetMaxEFailGrowth()*
ADAPT_ETACF	*ARKodeSetMaxCFailGrowth()*
NONLIN_CRDOWN	*ARKodeSetNonlinCRDown()*
NONLIN_RDIV	*ARKodeSetNonlinRDiv()*
LSETUP_DGMAX	*ARKodeSetDeltaGammaMax()*
FIXED_STEP	*ARKodeSetFixedStep()*

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 *ARKode()* 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 *ARKodeSetERKTable()*.

> **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 *ARKodeSetIRKTable()*.

   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 *ARKodeSetARKTables()*.

   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 *ARKodeResStolerance()* and *ARKodeResVtolerance()*.

  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 *ARKodeSetAdaptivityMethod()*.

  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 ARKode 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 ARKode. In ARKode'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 embedding.

- *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, ≠ 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, ≠ 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 ARKode'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, ≠ 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, ≠ 0 failure).

> Note: The call to *FARKEXPSTABSET()* must occur *after* the call to *FARKMALLOC()*.

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 ARKDLS or ARKSPILS linear solver interface.

ARKDLS direct linear solver interface

To attach a direct SUNLINSOL object and corresponding SUNMATRIX object to the ARKDLS interface, then following calls to initialize the SUNLINSOL and SUNMATRIX objects in steps *SUNMATRIX module initialization* and *SUNLINSOL module initialization* above, the user must call the *FARKDLSINIT()* routine:

subroutine **FARKDLSINIT** (*IER*)

> Interfaces with the *ARKDlsSetLinearSolver()* function to specify use of the direct linear solver interface.

> **Arguments:**

> > • *IER* (int, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

As an option when using the ARKDLS interface with 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 *ARKDlsJacFn()*), 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 *FARKDLSINIT()*, the user must call the routine *FARKDENSESETJAC()*:

subroutine **FARKDENSESETJAC** (*FLAG, IER*)

> Interface to the *ARKDlsSetJacFn()* 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 the ARKDLS interface with 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 *ARKDlsJacFn()*), 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 *FARKDLSINIT()*, the user must call the routine *FARKBANDSETJAC()*.

subroutine FARKBANDSETJAC (*FLAG*, *IER*)

> Interface to the *ARKDlsSetJacFn()* 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 the ARKDLS interface with 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 `ARKDlsJacFn()`), 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 `FARKDLSINIT()`, the user must call the routine `FARKSPARSESETJAC()`.

subroutine **FARKSPARSESETJAC** (*IER*)

Interface to the `ARKDlsSetJacFn()` 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).

ARKSPILS iterative linear solver interface

To attach an iterative SUNLINSOL object to the ARKSPILS interface, then following the call to initialize the SUNLINSOL object in step *SUNLINSOL module initialization* above, the user must call the `FARKSPILSINIT()` routine:

subroutine FARKSPILSINIT (*IER*)

> Interfaces with the *ARKSpilsSetLinearSolver()* function to specify use of the iterative linear solver interface.

> **Arguments:**

>> • *IER* (int, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

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 *FARKSPILSSETEPSLIN()*:

subroutine FARKSPILSSETEPSLIN (*EPLIFAC*, *IER*)

> Interface to the function *ARKSpilsSetEpsLin()* to specify the linear solver tolerance scale factor ϵ_L for the Newton system linear solver.

> This routine must be called *after FARKSPILSINIT()*.

> **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 *FARKSPILSINIT()*, the user must call the *FARKSPILSSETJAC()* routine with *FLAG* $\neq 0$:

subroutine FARKSPILSSETJAC (*FLAG*, *IER*)

> Interface to the function *ARKSpilsSetJacTimes()* to specify use of the user-supplied Jacobian-times-vector setup and product functions, *FARKJTSETUP()* and *FARKJTIMES()*, respectively.

> This routine must be called *after FARKSPILSINIT()*.

> **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 *FARKSPILSINIT()* the user must call the routine *FARKSPILSSETPREC()* with *FLAG* $\neq 0$:

subroutine FARKSPILSSETPREC (*FLAG*, *IER*)

> Interface to the function *ARKSpilsSetPreconditioner()* to specify use of the user-supplied preconditioner setup and solve functions, *FARKPSET()* and *FARKPSOL()*, respectively.

> This routine must be called *after FARKSPILSINIT()*.

> **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 the ARKSPILS linear solver interface, 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 *ARKSpilsJacTimesVecFn()*), 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 *ARKSpilsJacTimesSetupFn()*).

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 *ARKSpilsPrecSolveFn()*).

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 $M(T) - \gamma J$, 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 `ARKSpilsPrecSetupFn()`).

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 ARKMASSDLS or ARKMASSSPILS linear solver interface.

ARKDLS direct mass matrix linear solver interface

To attach a direct SUNLINSOL object and corresponding SUNMATRIX object to the ARKDLS mass matrix solver interface, then following the calls to initialize the SUNLINSOL and SUNMATRIX objects for the mass-matrix system in steps *SUNMATRIX module initialization* and *SUNLINSOL module initialization* above, the user must call the *FARKDLSMASSINIT()* routine:

subroutine **FARKDLSMASSINIT** (*TIME_DEP*, *IER*)
> Interfaces with the *ARKDlsSetMassLinearSolver()* function to specify use of the direct mass matrix linear solver interface.

> **Arguments:**

>> • *TIME_DEP* (int, input) – flag indicating whether the mass matrix is time-dependent (1) or not (0).

>> • *IER* (int, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

When using the ARKDLS 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 $M(t)$. 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 *ARKDlsMassFn()*), 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 *`FARKDLSMASSINIT()`*, the user must call the routine *`FARKDENSESETMASS()`*:

subroutine FARKDENSESETMASS (*IER*)
  Interface to the *`ARKDlsSetMassFn()`* 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 ARKDLS 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 $M(t)$. 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 *`ARKDlsMassFn()`*), 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 *`FARKDLSMASSINIT()`*, the user must call the routine *`FARKBANDSETMASS()`*:

subroutine FARKBANDSETMASS (*IER*)
  Interface to the *`ARKDlsSetMassFn()`* 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 ARKDLS 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 $M(t)$. 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 $M(t)$ 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 *ARKDlsMassFn()*), 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 *FARKDLSMASSINIT()*, the user must call the routine *FARKSPARSESETMASS()*:

subroutine **FARKSPARSESETMASS** (*IER*)

> Interface to the *ARKDlsSetMassFn()* 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).

ARKSPILS iterative mass matrix linear solver interface

To attach an iterative SUNLINSOL object to the ARKSPILS mass matrix solver interface, then following the call to initialize the SUNLINSOL object in step *SUNLINSOL module initialization* above, the user must call the *FARKSPILSMASSINIT()* routine:

subroutine **FARKSPILSMASSINIT** (*TIME_DEP*, *IER*)

> Interfaces with the *ARKSpilsSetMassLinearSolver()* function to specify use of the iterative mass matrix solver interface.

> **Arguments:**

- *TIME_DEP* (`int`, input) – flag indicating whether the mass matrix is time-dependent (1) or not (0).

- *IER* (`int`, output) – return flag (0 if success, -1 if a memory allocation error occurred, -2 for an illegal input).

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 *FARKSPILSSETMASSEPSLIN()*:

subroutine FARKSPILSSETMASSEPSLIN (*EPLIFAC*, *IER*)

Interface to the function *ARKSpilsSetEpsLin()* to specify the linear solver tolerance scale factor ϵ_L for the mass matrix linear solver.

This routine must be called *after* *FARKSPILSMASSINIT()*.

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 possibly time-dependent system mass matrix $M(t)$ 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 *ARKSpilsMassTimesVecFn()*), 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 *ARKSpilsMassTimesSetupFn()*).

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 *FARKSPILSMASSINIT()*, the user must call the routine *FARKSPILSSETMASS()*:

subroutine FARKSPILSSETMASS (*IER*)

    Interface to the function *ARKSpilsSetMassTimes()* to specify use of the user-supplied mass-matrix-times-vector setup and product functions *FARKMTSETUP()* and *FARKMTIMES()*.

    This routine must be called *after FARKSPILSMASSINIT()*.

    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 *FARKSPILSSETMASSPREC()* with *FLAG* $\neq 0$:

subroutine FARKSPILSSETMASSPREC (*FLAG*, *IER*)

    Interface to the function *ARKSpilsSetMassPreconditioner()* to specify use of the user-supplied preconditioner setup and solve functions, *FARKMASSPSET()* and *FARKMASSPSOL()*, respectively.

    This routine must be called *after FARKSPILSMASSINIT()*.

    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 *ARKSpilsMassPrecSetupFn()*).

    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 matrix $M(t)$, where M is the system mass matrix.

subroutine FARKMASSPSOL (*T*, *R*, *Z*, *DELTA*, *LR*, *IPAR*, *RPAR*, *IER*)

    User-supplied preconditioner solve routine (of type *ARKSpilsMassPrecSolveFn()*).

    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 $M(t)$.

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.

$$\left(\sum_i \left(\rho_i\, EWT_i\right)^2\right)^{1/2} < \delta.$$

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 *ARKode()* 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 *ARKode()* 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 *ARKode()*.

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 ARKode 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 ARKode 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 the ARKDLS or ARKSPILS interface, as shown above. If only linear solver parameters are being modified, then these calls may be made without re-attaching to the ARKDLS or ARKSPILS interface.

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 *ARKodeResize()*. This function modifies ARKode'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 ARKode 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 the ARKDLS or ARKSPILS interfaces.

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()*, *FARKDLSINIT()*/*FARKSPILSINIT()*, *FARKDLSMASSINIT()*/*FARKSPILSMASSINIT()*, and the SUNMATRIX and SUNLINSOL objects, the user may call *FARKFREE()*, as follows:

subroutine FARKFREE ()

> Frees the internal memory created by *FARKMALLOC()*.

> **Arguments:** None.

5.2.3 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 29, 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 ARKode solver, and list the relevant ARKode C/C++ function that is actually called to extract the output value. Similarly, optional integer output values that are specific to the ARKDLS linear solver interface are listed in *Table: Optional ARKDLS interface outputs*, while integer optional output values specific to the ARKSPILS iterative linear solver interface are listed in *Table: Optional ARKSPILS interface outputs*.

For more details on the optional inputs and outputs to ARKode, see the sections *Optional input functions* and *Optional output functions*.

Table: Optional FARKODE integer outputs

IOUT Index	Optional output	ARKode function
1	LENRW	*ARKodeGetWorkSpace()*
2	LENIW	*ARKodeGetWorkSpace()*
3	NST	*ARKodeGetNumSteps()*
4	NST_STB	*ARKodeGetNumExpSteps()*
5	NST_ACC	*ARKodeGetNumAccSteps()*
6	NST_ATT	*ARKodeGetNumStepAttempts()*
7	NFE	*ARKodeGetNumRhsEvals()* (num f_E calls)
8	NFI	*ARKodeGetNumRhsEvals()* (num f_I calls)
9	NSETUPS	*ARKodeGetNumLinSolvSetups()*
10	NETF	*ARKodeGetNumErrTestFails()*
11	NNI	*ARKodeGetNumNonlinSolvIters()*
12	NCFN	*ARKodeGetNumNonlinSolvConvFails()*
13	NGE	*ARKodeGetNumGEvals()*

Table: Optional FARKODE real outputs

ROUT Index	Optional output	ARKode function
1	H0U	*ARKodeGetActualInitStep()*
2	HU	*ARKodeGetLastStep()*
3	HCUR	*ARKodeGetCurrentStep()*
4	TCUR	*ARKodeGetCurrentTime()*
5	TOLSF	*ARKodeGetTolScaleFactor()*
6	UROUND	UNIT_ROUNDOFF (see the section *Data Types*)

Table: Optional ARKDLS interface outputs

IOUT Index	Optional output	ARKode function
14	LENRWLS	*ARKDlsGetWorkSpace()*
15	LENIWLS	*ARKDlsGetWorkSpace()*
16	LSTF	*ARKDlsGetLastFlag()*
17	NFELS	*ARKDlsGetNumRhsEvals()*
18	NJE	*ARKDlsGetNumJacEvals()*

Table: Optional ARKDLS mass interface outputs

IOUT Index	Optional output	ARKode function
23	LENRWMS	*ARKDlsGetMassWorkSpace()*
24	LENIWMS	*ARKDlsGetMassWorkSpace()*
25	LSTMF	*ARKDlsGetLastMassFlag()*
26	NMSET	*ARKDlsGetNumMassSetups()*
27	NMSOL	*ARKDlsGetNumMassSolves()*
28	NMMUL	*ARKDlsGetNumMassMult()*

Table: Optional ARKSPILS interface outputs

IOUT Index	Optional output	ARKode function
14	LENRWLS	*ARKSpilsGetWorkSpace()*
15	LENIWLS	*ARKSpilsGetWorkSpace()*
16	LSTF	*ARKSpilsGetLastFlag()*
17	NFELS	*ARKSpilsGetNumRhsEvals()*
18	NJTV	*ARKSpilsGetNumJtimesEvals()*
19	NPE	*ARKSpilsGetNumPrecEvals()*
20	NPS	*ARKSpilsGetNumPrecSolves()*
21	NLI	*ARKSpilsGetNumLinIters()*
22	NCFL	*ARKSpilsGetNumConvFails()*

Table: Optional ARKSPILS mass interface outputs

IOUT Index	Optional output	ARKode function
23	LENRWMS	*ARKSpilsGetMassWorkSpace()*
24	LENIWMS	*ARKSpilsGetMassWorkSpace()*
25	LSTMF	*ARKSpilsGetLastMassFlag()*
26	NMPE	*ARKSpilsGetNumMassPrecEvals()*
27	NMPS	*ARKSpilsGetNumMassPrecSolves()*
28	NMLI	*ARKSpilsGetNumMassIters()*
29	NMCFL	*ARKSpilsGetNumMassConvFails()*

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 *ARKodeGetErrWeights()*).

>   **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 *ARKodeGetEstLocalErrors()*).

>   **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*.

5.2.4 Usage of the FARKROOT interface to rootfinding

The FARKROOT interface package allows programs written in Fortran to use the rootfinding feature of the ARKode solver module. The user-callable functions in FARKROOT, with the corresponding ARKODE functions, are as follows:

  • *FARKROOTINIT()* interfaces to *ARKodeRootInit()*,

  • *FARKROOTINFO()* interfaces to *ARKodeGetRootInfo()*, and

  • *FARKROOTFREE()* interfaces to *ARKodeRootInit()*, 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 ARKode, 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 ARKode 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/ARKode 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.

5.2.5 Usage of the FARKODE interface to built-in preconditioners

The FARKODE interface enables usage of the two built-in preconditioning modules ARKBANDPRE and ARKBBD-PRE. 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 ARKode 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 ARKSPILS interface and any of the Krylov iterative linear solvers.

The two user-callable functions in this package, with the corresponding ARKode 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 ARKSPILS iterative linear solver interface by calling *FARKSPILSINIT()*.

   Optionally, to specify that ARKSPILS should use the supplied *FARKJTIMES()* and *FARKJTSETUP()* routines, the user should call *FARKSPILSSETJAC()* with FLAG $\neq$ 0, as described in the section *ARKSPILS iterative linear solver interface*.

   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 specific to the ARKSPILS interface are listed in *Table: Optional ARKSPILS 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 reinitialization*

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 ARKode 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 ARKode 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 ARKode).

Table: FARKBBD function mapping

FARKBBD routine (FORTRAN, user-supplied)	ARKode routine (C, interface)	ARKode interface function type
FARKGLOCFN()	FARKgloc	*ARKLocalFn()*
FARKCOMMFN()	FARKcfn	*ARKCommFn()*
FARKJTIMES()	FARKJtimes	*ARKSpilsJacTimesVecFn()*
FARKJTSETUP()	FARKJTSetup	*ARKSpilsJacTimesSetupFn()*

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 the ARKSPILS iterative linear solver interface by calling *FARKSPILSINIT()*.

   Optionally, to specify that ARKSPILS should use the supplied *FARKJTIMES()* and *FARKJTSETUP()* routines, the user should call *FARKSPILSSETJAC()* with FLAG $\neq$ 0, as described in the section *ARKSPILS iterative linear solver interface*.

   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 specific to the ARKSPILS interface are listed in *Table: Optional ARKSPILS 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 reinitialization

   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 ARKode 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 *FARKSPILSINIT()* 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 dyanamical time scales) is being solved using the same linear solver (SPGMR, SPBCG, SPTFQMR, SPFGMR or PCG) in combination with the ARKBBDPRE preconditioner, then the ARKode 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 *FARKSPILSINIT()* 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 interprocess 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*.

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.

6.1 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);
   void        (*nvsetarraypointer)(realtype *, 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);
};
```

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 vector operations defined by the generic NVECTOR module. Finally, we note that the generic NVECTOR module defines the functions N_VCloneVectorArray and N_VCloneVectorArrayEmpty. Both 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.

Similarly, 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.

In particular, any implementation of the NVECTOR module **must**:

- Specify the *content* field of the N_Vector.

- Define and implement the necessary 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. We further note that not all of the defined operations are required for each solver in SUNDIALS. The list of required operations for use with ARKode is given in the section *NVECTOR functions required by ARKode*.

- Define and implement user-callable constructor and destructor routines to create and free a `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`.

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.

6.1.1 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_CUSTOM	User-provided custom vector	6

6.2 Description of the NVECTOR operations

For each of the `N_Vector` 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. 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 = NVGetArrayPointer(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:

```
NVSetArrayPointer(vdata,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:

$$z_i = ax_i + by_i, \quad i = 1, \ldots, n.$$

> 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*:

$$z_i = c, \quad i = 1, \ldots, n.$$

> 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*:

$$z_i = x_i y_i, \quad i = 1, \ldots, n.$$

> 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*:

$$z_i = \frac{x_i}{y_i}, \quad i = 1, \ldots, n.$$

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*:

$$z_i = cx_i, \quad i = 1, \ldots, n.$$

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*:

$$y_i = |x_i|, \quad i = 1, \ldots, n.$$

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*:

$$z_i = 1.0/x_i, \quad i = 1, \ldots, n.$$

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*:

$$z_i = x_i + b, \quad i = 1, \ldots, n.$$

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*:

$$d = \sum_{i=1}^{n} x_i y_i.$$

Usage:

```
     d = N_VDotProd(x, y);
```

realtype **N_VMaxNorm** (N_Vector x)

    Returns the value of the l_∞ norm of the N_Vector x:

$$m = \max_{1 \le i \le n} |x_i|.$$

    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:

$$m = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i w_i)^2 \right)^{1/2}.$$

    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 (positive) realtype weight vector w built using only the elements of x corresponding to nonzero elements of the N_Vector id:

$$m = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i w_i \text{sign}(id_i))^2 \right)^{1/2}.$$

```
     m = N_VWrmsNormMask(x, w, id);
```

realtype **N_VMin** (N_Vector x)

    Returns the smallest element of the N_Vector x:

$$m = \min_{1 \le i \le n} x_i.$$

    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:

$$m = \left(\sum_{i=1}^{n} (x_i w_i)^2 \right)^{1/2}.$$

    Usage:

```
     m = N_VWL2Norm(x, w);
```

realtype **N_VL1Norm** (N_Vector x)

    Returns the l_1 norm of the N_Vector x:

$$m = \sum_{i=1}^{n} |x_i|.$$

    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 $1 \le i \le n$,

$$z_i = \begin{cases} 1.0 & \text{if } |x_i| \ge c, \\ 0.0 & \text{otherwise} \end{cases}.$$

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:

$$z_i = 1.0/x_i, \quad i = 1, \dots, n.$$

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:

$$x_i > 0 \text{ if } c_i = 2,$$
$$x_i \ge 0 \text{ if } c_i = 1,$$
$$x_i < 0 \text{ if } c_i = -2,$$
$$x_i \le 0 \text{ if } c_i = -1.$$

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:

$$\min_{i=1,\dots,n} \frac{\text{num}_i}{\text{denom}_i}.$$

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);
```

6.3 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 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] )
```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in the section *Description of the NVECTOR operations*. Their names are obtained from those in that section by appending the suffix `_Serial` (e.g. `N_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()*.

sunindextype **N_VGetLength_Serial** (N_Vector *v*)
　　This function returns the number of vector elements.

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`.

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.

For solvers that include a Fortran interface module, the NVECTOR_SERIAL module also includes a Fortran-callable function `FNVINITS(code, NEQ, IER)`, to initialize this NVECTOR_SERIAL 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.

6.4 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 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] )
```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in the section *Description of the NVECTOR operations*. Their names are obtained from those that section 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_VGetLength_Parallel**(N_Vector *v*)

This function returns the number of vector elements (global vector length).

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.

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.

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 NVEC-TOR_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.

6.5 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 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] )
```

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in the section *Description of the NVECTOR operations*. Their names are obtained from those in that section by appending the suffix _OpenMP (e.g. N_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()*.

sunindextype **N_VGetLength_OpenMP** (N_Vector *v*)
> This function returns the number of vector elements.

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.

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.

For solvers that include a Fortran interface module, the NVECTOR_OPENMP module also includes a Fortran-callable function FNVINITOMP(code, NEQ, NUMTHREADS, IER), to initialize this NVECTOR_OPENMP 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.

6.6 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), the number of threads used is based on the supplied argument in the vector constructor.

```c
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 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:

```c
#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:

```c
#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:

```c
#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:

```c
#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] )
```

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in the section *Description of the NVECTOR operations*. Their names are obtained from those in that section by appending the suffix _Pthreads (e.g. N_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()*.

sunindextype **N_VGetLength_Pthreads** (N_Vector *v*)

This function returns the number of vector elements.

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`.

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.

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 NVECTOR_PTHREADS 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.

6.7 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. Unlike native SUNDIALS vector types, NVECTOR_PARHYP does not provide macros to access its member variables.

The NVECTOR_PARHYP module defines implementations of all vector operations listed in the section *Description of the NVECTOR 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 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 section *Description of the NVECTOR operations* by appending the suffix _ParHyp (e.g. N_VDestroy_ParHyp). The module {nvecph} 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.

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.

6.8 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. 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.

The NVECTOR_PETSC module defines implementations of all vector operations listed in the section *Description of the NVECTOR 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 section *Description of the NVECTOR 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`.

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.

6.9 The NVECTOR_CUDA Module

The NVECTOR_CUDA module is an experimental implementation of `N_Vector` in CUDA language. It allows for SUNDIALS vector kernels to run on GPU devices. It is intended for users who are already familiar with CUDA and GPU programming. Building this vector module requires CUDA compiler and, by extension, C++ compiler. Class `Vector` in namespace `suncudavec` manages vector data layout.

```
template <class T, class I>
class Vector {
  I size_;
  I mem_size_;
  T* h_vec_;
  T* d_vec_;
  StreamPartitioning<T, I>* partStream_;
  ReducePartitioning<T, I>* partReduce_;
  bool ownPartitioning_;

  ...
};
```

The class members are vector size (length), size of the vector data memory block, pointers to vector data on the host and the device, pointers to classes `StreamPartitioning` and `ReducePartitioning`, which handle thread partitioning for streaming and reduction vector kernels, respectively, and the boolean flag that signals if the vector owns thread partitioning. The class `Vector` inherits from empty structure

```
struct _N_VectorContent_Cuda {
};
```

to interface the C++ class with `N_Vector` C code. When instantiated, the class `Vector` will allocate memory on both, host and device. Due to rapid progress in of CUDA development, we expect that `suncudavec::Vector` class will change frequently in the future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `suncudavec::Vector` class without requiring changes to user API.

The header file to be included when using this module is `nvector/nvector_cuda.h`. Unlike other native SUN-DIALS vector types, NVECTOR_CUDA does not provide macros to access its member variables. Note that NVEC-TOR_CUDA requires SUNDIALS to be built with MPI support.

The NVECTOR_CUDA module defines implementations of all vector operations listed in the section *Description of the NVECTOR operations*, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. As such, this vector cannot be used with SUNDIALS Fortran interfaces, nor with SUNDIALS direct solvers and preconditioners. This support will be added in subsequent SUNDIALS releases. The NVECTOR_CUDA 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_CUDA are provided in example programs for CVODE *[HSR2017]*.

The names of vector operations are obtained from those in the section *Description of the NVECTOR 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 *vec_length*)
> This function creates and allocates memory for a CUDA `N_Vector`. The memory is allocated on both, host and device. Its only argument is the vector length.

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** (N_VectorContent_Cuda *c*)
> This function creates and allocates memory for an NVECTOR_CUDA wrapper around a user-provided

suncudavec::Vector class. Its only argument is of type N_VectorContent_Cuda, which is the pointer to the class.

N_Vector* **N_VCloneVectorArray_Cuda** (int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* NVECTOR_CUDA vectors.

N_Vector* **N_VCloneVectorArrayEmpty_Cuda** (int *count*, N_Vector *w*)
This function creates (by cloning) an array of *count* NVECTOR_CUDA vectors, each with pointers to CUDA vectors set to NULL.

void **N_VDestroyVectorArray_Cuda** (N_Vector* *vs*, int *count*)
This function frees memory allocated for the array of *count* variables of type N_Vector created with *N_VCloneVectorArray_Cuda()* or with *N_VCloneVectorArrayEmpty_Cuda()*.

sunindextype **N_VGetLength_Cuda** (N_Vector *v*)
This function returns the length of the vector.

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.

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.

Notes

- When there is a need to access components of an N_Vector_Cuda, v, it is recommended to use functions *N_VGetDeviceArrayPointer_Cuda()* or *N_VGetHostArrayPointer_Cuda()*.

- 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.

6.10 The NVECTOR_RAJA Module

The NVECTOR_RAJA module is an experimental implementation of N_Vector using the RAJA hardware abstraction layer https://software.llnl.gov/RAJA/. In this implementation, RAJA allows for SUNDIALS vector kernels to run on 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 a CUDA software development toolkit. Besides the CUDA backend, RAJA has other backends such as serial, OpenMP and OpenAC. These backends are not used in this SUNDIALS release. Class Vector in namespace sunrajavec manages the vector data layout:

```
template <class T, class I>
class Vector {
  I size_;
  I mem_size_;
  T* h_vec_;
```

```
    T* d_vec_;

    ...
};
```

The class members are: vector size (length), size of the vector data memory block, and pointers to vector data on the host and on the device. The class `Vector` inherits from an empty structure

```
struct _N_VectorContent_Raja {
};
```

to interface the C++ class with the `N_Vector` C code. When instantiated, the class `Vector` will allocate memory on both the host and the device. Due to the rapid progress of RAJA development, we expect that the `sunrajavec::Vector` class will change frequently in the future SUNDIALS releases. The code is structured so that it can tolerate significant changes in the `sunrajavec::Vector` class without requiring changes to the user API.

The header file to be included when using this module is `nvector/nvector_raja.h`. Unlike other native SUN-DIALS vector types, NVECTOR_RAJA does not provide macros to access its member variables. Note that NVEC-TOR_RAJA requires SUNDIALS to be built with MPI support.

The NVECTOR_RAJA module defines the implementations of all vector operations listed in the section *Description of the NVECTOR operations*, except for `N_VGetArrayPointer` and `N_VSetArrayPointer`. 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 section *Description of the NVECTOR 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_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** (N_VectorContent_Raja *c*)
> This function creates and allocates memory for an NVECTOR_RAJA wrapper around a user-provided `sunrajavec::Vector` class. Its only argument is of type `N_VectorContent_Raja`, which is the pointer to the class.

N_Vector* **N_VCloneVectorArray_Raja** (int *count*, N_Vector *w*)
> This function creates (by cloning) an array of *count* NVECTOR_RAJA vectors.

N_Vector* **N_VCloneVectorArrayEmpty_Raja** (int *count*, N_Vector *w*)
> This function creates (by cloning) an array of *count* NVECTOR_RAJA vectors, each with pointers to RAJA vectors set to `NULL`.

void **N_VDestroyVectorArray_Raja** (N_Vector* *vs*, int *count*)
> This function frees memory allocated for the array of *count* variables of type `N_Vector` created with *N_VCloneVectorArray_Raja()* or with *N_VCloneVectorArrayEmpty_Raja()*.

sunindextype **N_VGetLength_Raja** (N_Vector *v*)
> This function returns the length of the vector.

realtype* **N_VGetHostArrayPointer_Raja** (N_Vector *v*)
> This function returns a pointer to the vector data on the host.

realtype* **N_VGetDeviceArrayPointer_Raja** (N_Vector *v*)
> This function returns a pointer to the vector data on the device.

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`.

Notes

- When there is a need to access components of an `N_Vector_Raja`, `v`, it is recommeded to use functions *N_VGetDeviceArrayPointer_Raja()* or *N_VGetHostArrayPointer_Raja()*.

- 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.

6.11 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_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 max norm.

- `Test_N_VWrmsNorm`: Create vector of known values, find and validate weighted root mean square.

- `Test_N_VWrmsNormMask`: Case 1: Create vector of known values, find and validate weighted root mean square using all elements.

- `Test_N_VWrmsNormMask`: Case 2: Create vector of known values, find and validate weighted root mean square using no elements.

- `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.

6.12 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 ARKode column shows function usage within the main integrator module, while the remaining columns show function usage within the ARKode linear solvers, the ARKBANDPRE and ARKBBDPRE preconditioner modules, and the FARKODE module. Here

ARKDLS stands for the direct linear solver interface in ARKode, and ARKSPILS stands for the scaled, preconditioned, iterative linear solver interface in ARKode.

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.

Routine	ARKode	ARKDLS	ARKSPILS	ARKBAND-PRE	ARKBBD-PRE	FARKODE
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
N_VDiv	X		X			X
N_VDotProd	X^1		X			X^1
N_VGetArrayPointer		X		X	X	X
N_VGetVectorID						
N_VInv	X					X
N_VLinearSum	X	X	X			X
N_VMaxNorm	X					X
N_VMin	X					X
N_VProd			X			
N_VScale	X	X	X	X	X	X
N_VSetArrayPointer		X				X
N_VSpace	X^2					X^2
N_VWrmsNorm	X	X	X	X	X	X

1. The `N_VDotProd()` function is only used by the main ARKode integrator module when the fixed-point non-linear solver is specified; when solving an explicit problem or when using a Newton solver with direct linear solver, it need not be supplied by the N_Vector implementation.

2. The `N_VSpace()` function is only informational, and need not be supplied by the N_Vector implementation.

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.

7.1 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          (*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`.

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.

7.1.1 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_CUSTOM	User-provided custom matrix	3

7.2 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);
```

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);
```

7.3 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 *Linear Solver Data Structures*. 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.

7.3.1 SUNDIALS matrix interfaces and vector implementations that can be used for each

Linear Solver	Se-rial	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
User supplied	X	X	X	X	X	X	X	X	X

7.4 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 $0 \le j < N$). 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 $0 \le i < M$ and $0 \le j < N$).

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).

7.5 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, $0 \leq \text{mu} < N$

- `ml` - lower half-bandwidth, $0 \leq \text{ml} < N$

- `smu` - storage upper bandwidth, $\text{mu} \leq \text{smu} < N$. The LU decomposition routines in the associated SUN-LINSOL_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 ($\text{ldim} \geq smu$)

- `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 ($= \text{ldim} \cdot (\text{smu} + \text{ml} + 1)$)

- `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 $j - \text{mu} \leq i \leq j + \text{ml}$.

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:

```
#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 )
```
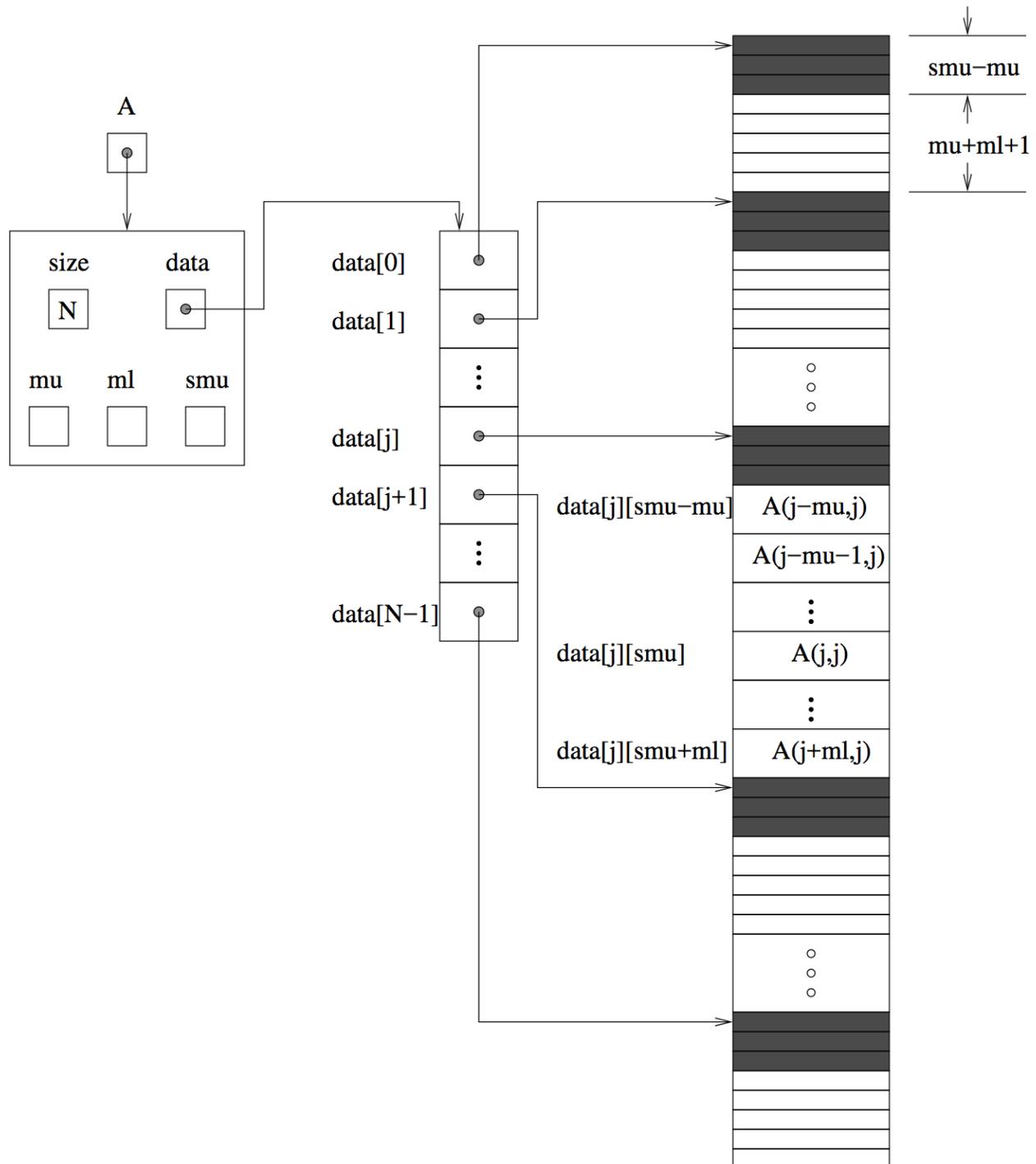
Fig. 7.1: Diagram of the storage for the SUNMATRIX_BAND module. Here A is an $N \times N$ band matrix with upper and lower half-bandwidths mu and ml, respectively. The rows and columns of A are numbered from 0 to N−1 and the (i, j)-th element of A is denoted A(i, j). The greyed out areas of the underlying component storage are used by the associated SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND linear solver.

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 $j - \mathrm{mu} \le i \le j + \mathrm{ml}$.

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 $j - \text{mu} \leq i \leq j + \text{ml}$.

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*, 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`, if the matrix will be used by the SUNLINSOL_BAND or SUNLINSOL_LAPACKBAND module then `smu` should be at least `min(N-1,mu+ml)`; otherwise `smu` should be at least `mu`.

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*, *SMU*, *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.

- *SMU* (`long int`, input) – storage upper 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*, *SMU*, *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.

- *SMU* (`long int`, input) – storage upper half-bandwidth.

- *IER* (`int`, output) – return flag (0 success, -1 for failure).

7.6 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 = N;
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:
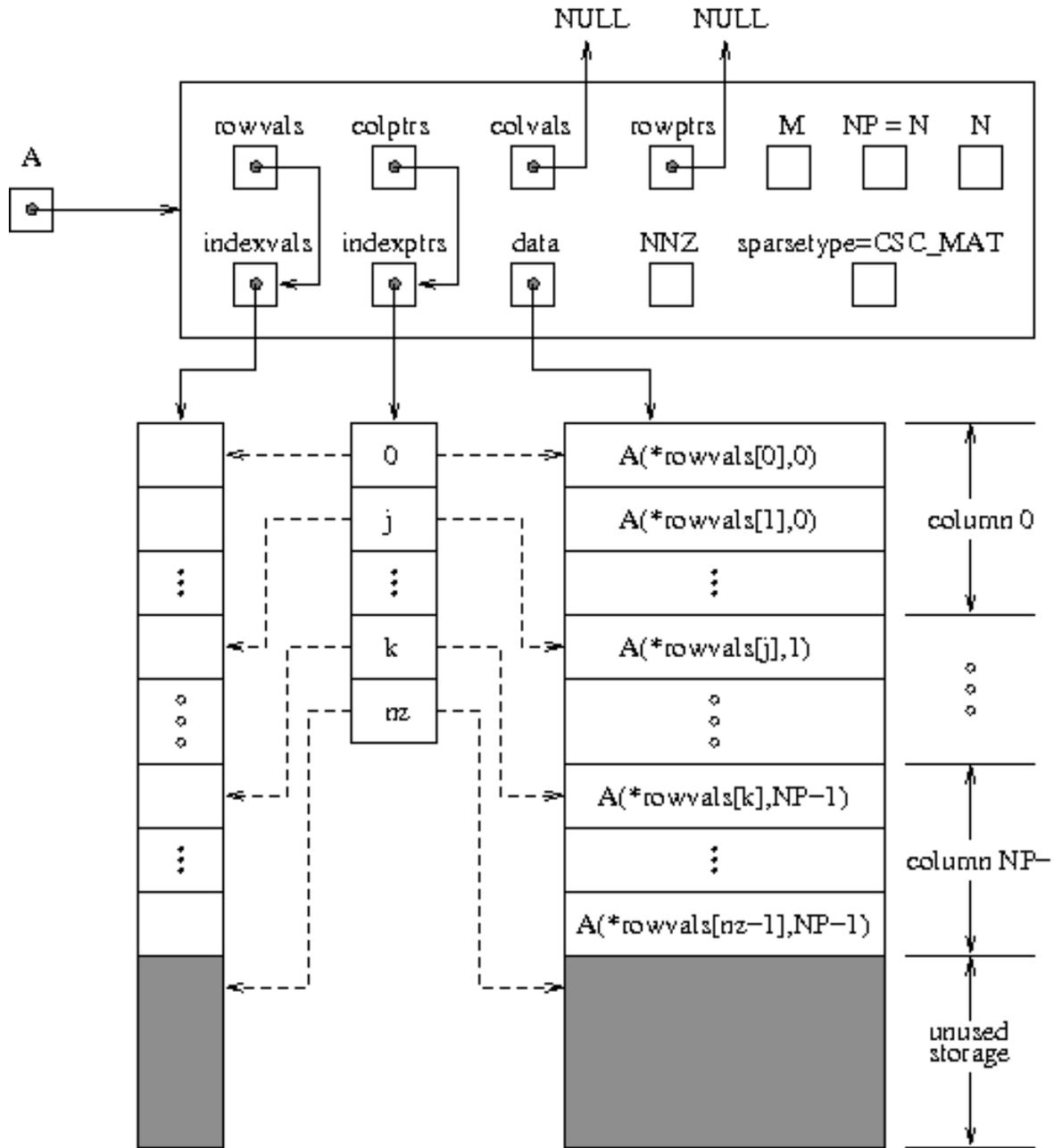
Fig. 7.2: Diagram of the storage for a compressed-sparse-column matrix of type SUNMATRIX_SPARSE: Here A is an $M \times N$ sparse CSC matrix with storage for up to NNZ nonzero entries (the allocated length of both `data` and `indexvals`). The entries in `indexvals` may assume values from 0 to M-1, corresponding 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.

```
#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.

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, 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_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).

7.7 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`.

7.8 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. Neither the main ARKode integrator or the ARKSPILS interface call `SUNMatrix` functions directly, so the table columns are specific to the ARKDLS direct solver interface and the ARKBANDPRE and ARKBBDPRE preconditioner modules.

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.

7.8.1 List of matrix functions usage by ARKode code modules

Routine	ARKDLS	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		
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`).

LINEAR SOLVER DATA STRUCTURES

The SUNDIALS library comes packaged with a variety of `SUNLinearSolver` implementations, designed for simulations requiring either direct or matrix-free iterative linear solvers for problems in serial or shared-memory parallel environments. SUNDIALS additionally provides a simple interface for generic linear solvers (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 linear solver operations, making them immediately extensible to new user-defined linear solver objects.

8.1 Description of the SUNLinearSolver Module

For problems that involve the solution of linear systems of equations, the SUNDIALS solvers operate using generic linear solver modules (of type `SUNLinearSolver`), through a set of operations defined by the particular `SUNLinearSolver` implementation. These 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 methods. Moreover, users can provide their own specific `SUNLinearSolver` implementation to each SUNDIALS solver, particularly in cases where they provide their own `N_Vector` and/or `SUNMatrix` modules, and the customized linear solver leverages these additional data structures to create highly efficient and/or scalable solvers for their particular problem. Additionally, SUNDIALS provides native implementations `SUNLinearSolver` modules, as well as `SUNLinearSolver` modules that interface between SUNDIALS and external linear solver libraries.

The various SUNDIALS solvers have been designed to specifically leverage the use of either *direct linear solvers* or matrix-free, *scaled, preconditioned, iterative linear solvers*, through their "Dls" and "Spils" interfaces, respectively. Additionally, SUNDIALS solvers can make use of user-supplied custom linear solvers, whether these are problem-specific or come from external solver libraries.

For iterative (and possibly custom) linear solvers, the SUNDIALS solvers 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, we apply the underlying iterative algorithm to the transformed system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.1}$$

where

$$\begin{aligned}
\tilde{A} &= S_1 P_1^{-1} A P_2^{-1} S_2^{-1}, \\
\tilde{b} &= S_1 P_1^{-1} b, \\
\tilde{x} &= S_2 P_2 x,
\end{aligned} \tag{8.2}$$

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_2x.

The SUNDIALS solvers request that iterative linear solvers stop based on the 2-norm of the scaled preconditioned residual meeting a prescribed tolerance

$$\left\| \tilde{b} - \tilde{A}\tilde{x} \right\|_2 < \text{tol.}$$

We note that not all of the iterative linear solvers implemented in SUNDIALS support the full range of the above options. Similarly, some of the SUNDIALS integrators only utilize a subset of these options. Exceptions to the operators shown above are described in the documentation for each `SUNLinearSolver` implementation, or for each SUNDIALS solver "Spils" interface.

The generic `SUNLinearSolver` type has been modeled after the object-oriented style of the generic `N_Vector` type. Specifically, a generic `SUNLinearSolver` is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the linear solver, and an *ops* field pointing to a structure with generic linear solver operations. The type `SUNLinearSolver` is defined as

```
typedef struct _generic_SUNLinearSolver *SUNLinearSolver;

struct _generic_SUNLinearSolver {
  void *content;
  struct _generic_SUNLinearSolver_Ops *ops;
};
```

The `_generic_SUNLinearSolver_Ops` structure is essentially a list of pointers to the various actual linear solver operations, and is defined as

```
struct _generic_SUNLinearSolver_Ops {
  SUNLinearSolver_Type (*gettype)(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);
  long int             (*lastflag)(SUNLinearSolver);
  int                  (*space)(SUNLinearSolver, long int*, long int*);
  N_Vector             (*resid)(SUNLinearSolver);
  int                  (*free)(SUNLinearSolver);
};
```

The generic `SUNLinearSolver` module defines and implements the linear solver operations acting on `SUNLinearSolver` objects. These routines are in fact only wrappers for the linear solver operations defined by a particular `SUNLinearSolver` implementation, which are accessed through the {em 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/failed operation:

```
int SUNLinSolInitialize(SUNLinearSolver S)
{
  return ((int) S->ops->initialize(S));
}
```

The subsection *Description of the SUNLinearSolver operations* contains a complete list of all linear solver operations defined by the generic `SUNLinearSolver` module. In order to support both direct and iterative linear solver types, the generic `SUNLinearSolver` module defines linear solver routines (or arguments) that may be specific to individual use cases. As such, for each routine we specify its intended use. If a custom `SUNLinearSolver` module is provided, the function pointers for non-required routines may be set to `NULL` to indicate that they are not provided.

A particular implementation of the `SUNLinearSolver` module must:

- Specify the *content* field of the `SUNLinearSolver` object.

- Define and implement a minimal subset of the linear solver operations. See the documentation for each SUN-DIALS linear solver interface to determine which `SUNLinearSolver` operations they require.

  Note that the names of these routines should be unique to that implementation in order to permit using more than one `SUNLinearSolver` 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.

- Optionally, define and implement additional user-callable routines acting on the newly defined `SUNLinearSolver` (e.g., routines to set various configuration options for tuning the linear solver to a particular problem).

- Optionally, provide functions as needed for that particular implementation to access different parts in the *content* field of the newly defined `SUNLinearSolver` object (e.g., routines to return various statistics from the solver).

Each `SUNLinearSolver` implementation included in SUNDIALS has a "type" identifier specified in enumeration and shown in Table *Identifiers associated with linear solver kernels supplied with SUNDIALS*. It is recommended that a user-supplied `SUNLinearSolver` implementation set this identifier based on the SUNDIALS solver interface they intend to use: "Dls" interfaces require the `SUNLINEARSOLVER_DIRECT` `SUNLinearSolver` objects, "Spils" interfaces require the `SUNLINEARSOLVER_ITERATIVE` objects.

8.1.1 Identifiers associated with linear solver kernels supplied with SUNDIALS

Linear Solver ID	Solver type	ID Value
SUNLINEARSOLVER_DIRECT	Direct solvers	0
SUNLINEARSOLVER_ITERATIVE	Iterative solvers	1
SUNLINEARSOLVER_CUSTOM	Custom solvers	2

8.2 Description of the SUNLinearSolver operations

For each of the `SUNLinearSolver` operations, we give the name, usage of the function, and a description of its mathematical operations below.

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 custom) from the abstract `SUNLinearSolver` interface. This is used to assess compatibility with SUNDIALS-provided linear solver interfaces. Returned values are given in the Table *Identifiers associated with linear solver kernels supplied with SUNDIALS*.

Usage:

```
type = SUNLinSolGetType(LS);
```

int **SUNLinSolInitialize** (SUNLinearSolver *LS*)

Performs linear solver initialization (assumes 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 *Error Codes returned from SUNLinearSolver implementations*.

Usage:

```
ier = 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 *Error Codes returned from SUNLinearSolver implementations*.

Usage:

```
ier = SUNLinSolSetup(LS, A);
```

int **SUNLinSolSolve** (SUNLinearSolver *LS*, SUNMatrix *A*, N_Vector *x*, N_Vector *b*, realtype *tol*)

Solves a linear system $Ax = b$. 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 *Error Codes returned from SUNLinearSolver implementations*.

Direct solvers: can ignore the *tol* argument.

Iterative solvers: can ignore the `SUNMatrix` input *A* since a `NULL` argument will be passed (these should instead rely on the matrix-vector product function supplied through the routine *SUNLinSolSetATimes()*). These should attempt to solve to the specified `realtype` tolerance *tol* in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

Custom solvers: all arguments will be supplied, and if the solver is approximate then it should attempt to solve to the specified `realtype` tolerance *tol* in a weighted 2-norm. If the solver does not support scaling then it should just use a 2-norm.

Usage:

```
ier = 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:

```
ier = SUNLinSolFree(LS);
```

int **SUNLinSolSetATimes** (SUNLinearSolver *LS*, void* *A_data*, *ATimesFn ATimes*)

Provides *ATimesFn* function pointer, as well as a `void *` pointer to a data structure used by this routine, to a linear solver object. SUNDIALS solvers 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 *Error Codes returned from SUNLinearSolver implementations*.

Usage:

```
    ier = SUNLinSolSetATimes(LS, A\_data, ATimes);
```

int **SUNLinSolSetPreconditioner** (SUNLinearSolver   *LS*,   void*   *P_data*,   *PSetupFn*   *Pset*,
PSolveFn Psol)
    (Optional; Iterative/Custom linear solvers only)

    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 solver, which will provide translation between the generic *Pset* and
    Psol calls and the integrator-specific and integrator- 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 *Error Codes returned from SUNLinearSolver implementations*.

    Usage:

```
    ier = SUNLinSolSetPreconditioner(LS, Pdata, Pset, Psol);
```

int **SUNLinSolSetScalingVectors** (SUNLinearSolver *LS*, N_Vector *s1*, N_Vector *s2*)
    (Optional; Iterative/Custom linear solvers only)

    Sets pointers to left/right scaling vectors for the linear system solve. Here, *s1* is an N_Vector of positive scale
    factors containing the diagonal of the S_1 scaling matrix. Similarly, *s2* is an N_Vector containing the diagonal
    of the S_2 scaling matrix. Neither of these vectors are 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 *Error
    Codes returned from SUNLinearSolver implementations*.

    Usage:

```
    ier = SUNLinSolSetScalingVectors(LS, s1, s2);
```

int **SUNLinSolNumIters** (SUNLinearSolver *LS*)
    (Optional; Iterative/Custom linear solvers only)

    Should return the number of linear iterations performed in the last "solve" call.

    Usage:

```
    its = SUNLinSolNumIters(LS);
```

realtype **SUNLinSolResNorm** (SUNLinearSolver *LS*)
    (Optional; Iterative/Custom linear solvers only)

    Should return the final residual norm from the last "solve" call.

    Usage:

```
    rnorm = SUNLinSolResNorm(LS);
```

N_Vector **SUNLinSolResid** (SUNLinearSolver *LS*)
    (Optional; Iterative/Custom linear solvers only)

    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
    function may be called by the SUNDIALS solver. This routine should return the N_Vector containing the
    preconditioned initial residual vector.

    Usage:

```
    rvec = SUNLinSolResid(LS);
```

long int **SUNLinSolLastFlag** (SUNLinearSolver *LS*)
    (Optional)

8.2. Description of the SUNLinearSolver operations   **205**

Should return the last error flag encountered within the linear solver. This is not called by the SUNDIALS solvers 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*)
(Optional)

Returns 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 in determining a user's total space requirements.

Usage:

```
ier = SUNLinSolSpace(LS, &lrw, &liw);
```

8.3 Description of the client-supplied SUNLinearSolver routines

The SUNDIALS packages provide the *ATimes*, *Pset* and *Psol* routines utilized by the `SUNLinearSolver` modules. These function types are defined in the header file `sundials/sundials_iterative.h`, and are described here in case a user wishes to interact directly with an iterative `SUNLinearSolver` object.

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 should already be alloced 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 should already be alloced 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 preconditioner or the 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 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.

8.4 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 direct 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.

8.4.1 Compatible SUNLinearSolver and SUNMatrix implementations

Linear Solver	Dense	Banded	Sparse	User Supplied
Dense	X			X
Band		X		X
LapackDense	X			X
LapackBand		X		X
KLU			X	X
SuperLU_MT			X	X
User supplied	X	X	X	X

8.5 Error Codes returned from SUNLinearSolver implementations

The functions within the SUNDIALS-provided `SUNLinearSolver` implementations return a common set of error codes, listed here:

- `SUNLS_SUCCESS` (0) – successful call or converged solve

- `SUNLS_MEM_NULL` (-1) – the memory argument to the function is `NULL`

- `SUNLS_ILL_INPUT` (-2) – an illegal input has been provided to the function

- `SUNLS_MEM_FAIL` (-3) – failed memory access or allocation

- `SUNLS_ATIMES_FAIL_UNREC` (-4) – an unrecoverable failure occurred in the `ATimes` routine

- `SUNLS_PSET_FAIL_UNREC` (-5) – an unrecoverable failure occurred in the `Pset` routine

- `SUNLS_PSOLVE_FAIL_UNREC` (-6) – an unrecoverable failure occurred in the `Psolve` routine

- `SUNLS_PACKAGE_FAIL_UNREC` (-7) – an unrecoverable failure occurred in an external linear solver package

- `SUNLS_GS_FAIL` (-8) – a failure occurred during Gram-Schmidt orthogonalization (SPGMR/SPFGMR)

- `SUNLS_QRSOL_FAIL` (-9) – a singular R matrix was encountered in a QR factorization (SPGMR/SPFGMR)

- `SUNLS_RES_REDUCED` (1) – an iterative solver reduced the residual, but did not converge to the desired tolerance

- `SUNLS_CONV_FAIL` (2) – an iterative solver did not converge (and the residual was not reduced)

- `SUNLS_ATIMES_FAIL_REC` (3) – a recoverable failure occurred in the `ATimes` routine

- `SUNLS_PSET_FAIL_REC` (4) – a recoverable failure occurred in the `Pset` routine

- `SUNLS_PSOLVE_FAIL_REC` (5) – a recoverable failure occurred in the `Psolve` routine

- `SUNLS_PACKAGE_FAIL_REC` (6) – a recoverable failure occurred in an external linear solver package

- `SUNLS_QRFACT_FAIL` (7) – a singular matrix was encountered during a QR factorization (SPGMR/SPFGMR)

- `SUNLS_LUFACT_FAIL` (8) – a singular matrix was encountered during a LU factorization

8.6 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). The SUNLINSOL_DENSE module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
  long int 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 header file to be included when using this module is `sunlinsol/sunlinsol_dense.h`.

The SUNLINSOL_DENSE module defines dense implementations of all "direct" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_DENSE provides the following additional user-callable constructor routine:

SUNLinearSolver **SUNDenseLinearSolver** (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 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 SUNDI-ALS 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).

8.7 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). The SUNLINSOL_BAND module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_Band {
  sunindextype N;
  sunindextype *pivots;
  long int 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 header file to be included when using this module is `sunlinsol/sunlinsol_band.h`.

The SUNLINSOL_BAND module defines band implementations of all "direct" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_BAND provides the following additional user-callable constructor routine:

SUNLinearSolver **SUNBandLinearSolver**(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 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).

8.8 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). The SUNLINSOL_LAPACKDENSE module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Dense {
  sunindextype N;
  sunindextype *pivots;
  long int 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 header file to be included when using this module is `sunlinsol/sunlinsol_lapackdense.h`.

The SUNLINSOL_LAPACKDENSE module defines dense implementations of all "direct" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_LAPACKDENSE provides the following additional user-callable constructor routine:

SUNLinearSolver **SUNLapackDense** (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 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).

8.9 The SUNLINSOL_LAPACKBAND Module

The LAPACK band implementation of the `SUNLinearSolver` module provided with SUNDIALS, SUNLIN-SOL_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 NVEC-TOR_PTHREADS). The SUNLINSOL_LAPACKBAND module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_Band {
  sunindextype N;
  sunindextype *pivots;
  long int 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 header file to be included when using this module is `sunlinsol/sunlinsol_lapackband.h`.

The SUNLINSOL_LAPACKBAND module defines band implementations of all "direct" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_LAPACKBAND provides the following additional user-callable routine:

SUNLinearSolver **SUNLapackBand**(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 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).

8.10 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). The SUNLINSOL_KLU module defines the *content* field of a `SUNLinearSolver` to be the following structure:

```
struct _SUNLinearSolverContent_KLU {
  long 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,

- `Numeric` - KLU storage structure for numeric factorization components,

- `Common` - storage structure for common KLU solver components,

- `klu_solver` – pointer to the appropriate KLU solver function (depending on whether it is using a CSR or CSC sparse matrix).

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 header file to be included when using this module is `sunlinsol/sunlinsol_klu.h`.

The SUNLINSOL_KLU module defines implementations of all "direct" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_KLU provides the following additional user-callable routines:

SUNLinearSolver **SUNKLU** (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 **SUNKLUReInit** (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 **SUNKLUSetOrdering** (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.

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 *SUNKLUReInit()* and *SUNKLUSetOrdering()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine **FSUNKLUReInit** (*CODE*, *NNZ*, *REINIT_TYPE*, *IER*)

Fortran interface to *SUNKLUReInit()* 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 *SUNKLUReInit()* 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 *SUNKLUSetOrdering()* 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 *SUNKLUSetOrdering()* 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.

8.11 The SUNLINSOL_SUPERLUMT Module

The SuperLU_MT implementation of the SUNLinearSolver module provided with SUNDIALS, SUNLIN-SOL_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. The SUNLINSOL_SUPERLUMT module defines the *content* field of a SUNLinearSolver to be the following structure:

```
struct _SUNLinearSolverContent_SuperLUMT {
  long 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 header file to be included when using this module is `sunlinsol/sunlinsol_superlumt.h`.

The SUNLINSOL_SUPERLUMT module defines implementations of all "direct" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_SUPERLUMT provides the following additional user-callable routines:

SUNLinearSolver **SUNSuperLUMT** (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 **SUNSuperLUMTSetOrdering** (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 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 *SUNSuperLUMTSetOrdering()* routine also supports Fortran interfaces for the system and mass matrix solvers:

subroutine **FSUNSuperLUMTSetOrdering** (*CODE*, *ORDERING*, *IER*)

> Fortran interface to *SUNSuperLUMTSetOrdering()* 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 *SUNSuperLUMTSetOrdering()* 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

8.12 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()*).

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;
  long 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;
};
```

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 $(\text{maxl}+1) \times \text{maxl}$ Hessenberg matrix. It is stored row-wise so that the (i,j)th element is given by `Hes[i][j]`,

- `givens` - a length 2 maxl array which represents the Givens rotation matrices that arise in the GMRES 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]` $= s_1, \ldots$, `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.

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 header file to be included when using this module is `sunlinsol/sunlinsol_spgmr.h`.

The SUNLINSOL_SPGMR module defines implementations of all "iterative" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_SPGMR provides the following additional user-callable routines:

SUNLinearSolver **SUNSPGMR** (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 **SUNSPGMRSetPrecType** (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 **SUNSPGMRSetGSType** (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 **SUNSPGMRSetMaxRestarts** (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`.

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 *SUNSPGMRSetGSType()*, *SUNSPGMRSetPrecType()* and *SUNSPGMRSetMaxRestarts()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPGMRSetGSType (*CODE*, *GSTYPE*, *IER*)

> Fortran interface to *SUNSPGMRSetGSType()* 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 *SUNSPGMRSetGSType()* 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 *SUNSPGMRSetPrecType()* 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 *SUNSPGMRSetPrecType()* 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 SUNSPGMRSetMaxRS() 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 SUNSPGMRSetMaxRS() 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.

8.13 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).

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;
  long 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;
};
```

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.

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 header file to be included when using this module is `sunlinsol/sunlinsol_spfgmr.h`.

The SUNLINSOL_SPFGMR module defines implementations of all "iterative" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_SPFGMR provides the following additional user-callable routines:

SUNLinearSolver **SUNSPFGMR** (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 **SUNSPFGMRSetPrecType** (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 **SUNSPFGMRSetGSType** (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 **SUNSPFGMRSetMaxRestarts** (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`.

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 *SUNSPFGMRSetGSType()*, *SUNSPFGMRSetPrecType()* and *SUNSPFGMRSetMaxRestarts()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPFGMRSetGSType (*CODE*, *GSTYPE*, *IER*)

    Fortran interface to *SUNSPFGMRSetGSType()* 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 *SUNSPFGMRSetGSType()* 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 *SUNSPFGMRSetPrecType()* 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 *SUNSPFGMRSetPrecType()* 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 `SUNSPFGMRSetMaxRS()` 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 SUNSPFGMRSetMaxRS() 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.

8.14 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.

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;
  long 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;
};
```

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.

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 SUNDI-ALS 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 header file to be included when using this module is `sunlinsol/sunlinsol_spbcgs.h`.

The SUNLINSOL_SPBCGS module defines implementations of all "iterative" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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`

The module SUNLINSOL_SPBCGS provides the following additional user-callable routines:

SUNLinearSolver **SUNSPBCGS** (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 **SUNSPBCGSSetPrecType** (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 **SUNSPBCGSSetMaxl** (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`.

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 *SUNSPBCGSSetPrecType()* and *SUNSPBCGSSetMaxl()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPBCGSSetPrecType(*CODE*, *PRETYPE*, *IER*)
> Fortran interface to *SUNSPBCGSSetPrecType()* 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 *SUNSPBCGSSetPrecType()* 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 *SUNSPBCGSSetMaxl()* 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 *SUNSPBCGSSetMaxl()* 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.

8.15 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.

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;
  long 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;
};
```

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.

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 SUNDI-ALS 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 header file to be included when using this module is `sunlinsol/sunlinsol_sptfqmr.h`.

The SUNLINSOL_SPTFQMR module defines implementations of all "iterative" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- 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`

The module SUNLINSOL_SPTFQMR provides the following additional user-callable routines:

SUNLinearSolver **SUNSPTFQMR** (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 **SUNSPTFQMRSetPrecType** (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 **SUNSPTFQMRSetMaxl** (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`.

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 *SUNSPTFQMRSetPrecType()* and *SUNSPTFQMRSetMaxl()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine FSUNSPTFQMRSetPrecType(*CODE*, *PRETYPE*, *IER*)

Fortran interface to *SUNSPTFQMRSetPrecType()* 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 *SUNSPTFQMRSetPrecType()* 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 *SUNSPTFQMRSetMaxl()* 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 *SUNSPTFQMRSetMaxl()* 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.

8.16 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

$$\tilde{A}\tilde{x} = \tilde{b} \tag{8.3}$$

where

$$\begin{aligned}
\tilde{A} &= SP^{-1}AP^{-1}S, \\
\tilde{b} &= SP^{-1}b, \\
\tilde{x} &= S^{-1}Px.
\end{aligned} \tag{8.4}$$

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:

$$\|\tilde{b} - \tilde{A}\tilde{x}\|_2 < \delta$$
$$\Leftrightarrow$$
$$\|SP^{-1}b - SP^{-1}Ax\|_2 < \delta$$
$$\Leftrightarrow$$
$$\|P^{-1}b - P^{-1}Ax\|_S < \delta$$

where $\|v\|_S = \sqrt{v^T S^T S v}$, with an input tolerance δ.

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;
  long 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;
};
```

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.

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 header file to be included when using this module is `sunlinsol/sunlinsol_pcg.h`.

The SUNLINSOL_PCG module defines implementations of all "iterative" linear solver operations listed in the section *Description of the SUNLinearSolver operations*:

- `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

The module SUNLINSOL_PCG provides the following additional user-callable routines:

SUNLinearSolver **SUNPCG** (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 **SUNPCGSetPrecType** (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 **SUNPCGSetMaxl** (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.

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 *SUNPCGSetPrecType()* and *SUNPCGSetMaxl()* routines also support Fortran interfaces for the system and mass matrix solvers:

subroutine **FSUNPCGSetPrecType**(*CODE*, *PRETYPE*, *IER*)

> Fortran interface to *SUNPCGSetPrecType()* 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 *SUNPCGSetPrecType()* 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 *SUNPCGSetMaxl()* 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 *SUNPCGSetMaxl()* 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.

8.17 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 texttt{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.

8.18 SUNLinearSolver functions required by ARKode

In the table below, we list the linear solver functions in the `SUNLinearSolver` module used within the ARKode package. The table also shows, for each function, which of the code modules uses the function. In general, the main ARKode integrator considers three categories of linear solvers, *direct*, *iterative* and *custom*, with interfaces accessible in the ARKode header files `arkode/arkode_direct.h` (ARKDLS), `arkode/arkode_spils.h` (ARK-SPILS) and `arkode/arkode_customls.h` (ARKCLS), respectively. Hence, the the table columns reference the use of `SUNLinearSolver` functions by each of these solver interfaces.

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. vector functions in the `N_Vector`

Routine	ARKDLS	ARKSPILS	ARKCLS
SUNLinSolGetType	X	X	O
SUNLinSolSetATimes		X	O
SUNLinSolSetPreconditioner		X	O
SUNLinSolSetScalingVectors		X	O
SUNLinSolInitialize	X	X	X
SUNLinSolSetup	X	X	X
SUNLinSolSolve	X	X	X
SUNLinSolNumIters		X	O
SUNLinSolResNorm		X	O
SUNLinSolResid			
SUNLinSolLastFlag			
SUNLinSolFree	X	X	X
SUNLinSolSpace	O	O	O

The linear solver functions listed above with a "O" are optionally used, in that these are only called if they are implemented in the `SUNLinearSolver` module that is being used (i.e. their function pointers are non-`NULL`). Also, although ARKode does not call *SUNLinSolLastFlag()* directly, this routine is available for users to query linear solver issues directly.

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:

  SRCDIR 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 `SRCDIR` 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 `SRCDIR`.

- 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*

9.1 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 2.8.1 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.

9.1.1 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
```

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 SRCDIR:

```
$ ccmake (...)/SRCDIR
```

Similarly, to build the default configuration using the wxWidgets GUI, from the BUILDDIR enter the `cmake-gui` command and point to the SRCDIR:

```
$ cmake-gui (...)/SRCDIR
```

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:

```
$ make install
```

Fig. 9.1: 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. 9.2: Changing the INSTDIR for SUNDIALS and corresponding EXAMPLES.

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
```

9.1.2 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.

BLAS_ENABLE Enable BLAS support

> Default: `OFF`

> **Note:** Setting this option to ON will trigger additional CMake options. See additional information on building with BLAS enabled in *Working with external Libraries*.

BLAS_LIBRARIES BLAS library

> Default: `/usr/lib/libblas.so`

> **Note:** CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

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 BLAS/LAPACK support is enabled (`BLAS_ENABLE` or `LAPACK_ENABLE` 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.
> ---

CXX_ENABLE Flag to enable C++ ARKode examples (if examples are enabled)

> Default: `OFF`

CUDA_ENABLE Build the SUNDIALS CUDA vector module.

> Default: `OFF`

EXAMPLES_ENABLE_C Build the SUNDIALS C examples

> Default: `ON`

EXAMPLES_ENABLE_CUDA Build the SUNDIALS CUDA examples

> Default: `OFF`

> ---
> **Note:** You need to enable CUDA support to build these examples.
> ---

EXAMPLES_ENABLE_CXX Build the SUNDIALS C++ examples

> Default: `OFF`

EXAMPLES_ENABLE_RAJA Build the SUNDIALS RAJA examples

> Default: `OFF`

> ---
> **Note:** You need to enable CUDA and RAJA 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: `OFF`

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`.

FCMIX_ENABLE Enable Fortran-C support

Default: `OFF`

F90_ENABLE Flag to enable Fortran 90 ARKode examples (if examples are enabled)

Default: `OFF`

HYPRE_ENABLE 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

KLU_ENABLE 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

LAPACK_ENABLE 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.

MPI_ENABLE Enable MPI support (build the parallel nvector).

> 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 (`MPI_ENABLE` 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 `MPI_ENABLE`.
> ---

MPI_Fortran_COMPILER `mpifort` program

> Default:

> ---
> **Note:** This option is triggered only if MPI is enabled (`MPI_ENABLE` is `ON`) and Fortran-C support is enabled (`FCMIX_ENABLE` is `ON`).
> ---

MPIEXEC_EXECUTABLE Specify run command for MPI

> Default: `mpirun`

> ---
> **Note:** This option is triggered only if MPI is enabled (`MPI_ENABLE` is `ON`).
> ---

OPENMP_ENABLE Enable OpenMP support (build the OpenMP NVector)

> Default: `OFF`

PETSC_ENABLE Enable PETSc support

> Default: `OFF`

> ---
> **Note:** See additional information on building with PETSc enabled in *Working with external Libraries*.
> ---

PETSC_INCLUDE_DIR Path to PETSc header files

> Default: none

PETSC_LIBRARY_DIR Path to PETSc installed library files

> Default: none

PTHREAD_ENABLE Enable Pthreads support (build the Pthreads NVector)

> Default: `OFF`

RAJA_ENABLE Enable RAJA support (build the RAJA NVector).

> Default: OFF

> ---
> **Note:** You need to enable CUDA in order to build the RAJA vector module.
> ---

SUNDIALS_INDEX_TYPE Integer type used for SUNDIALS indices, options are: `int32_t` or `int64_t`

> Default: `int64_t`

9.1. CMake-based installation   **251**

SUNDIALS_PRECISION Precision used in SUNDIALS, options are: `double`, `single` or `extended`

> Default: `double`

SUPERLUMT_ENABLE 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

USE_GENERIC_MATH Use generic (`stdc`) math libraries

> Default: `ON`

xSDK Configuration Options

SUNDIALS supports CMake configuration options defined by the Extreme-scale Scientific Software Development Kit (xSDK) community policies (see https://xsdk.info for more information). xSDK CMake options are unused by default but may be activated by setting `USE_XSDK_DEFAULTS` to `ON`.

Note: When xSDK options are active, they will overwrite the corresponding SUNDIALS option and may have different default values (see details below). As such the equivalent SUNDIALS options should not be used when configuring with xSDK options. In the GUI front end to CMake (`ccmake` or `cmake-gui`), setting `USE_XSDK_DEFAULTS` to ON will hide the corresponding SUNDIALS options as advanced CMake variables. During configuration, messages are output detailing which xSDK flags are active and the equivalent SUNDIALS options that are replaced. Below is a complete list xSDK options and the corresponding SUNDIALS options if applicable.

TPL_BLAS_LIBRARIES BLAS library

> Default: `/usr/lib/libblas.so`

> SUNDIALS equivalent: `BLAS_LIBRARIES`

> **Note:** CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

TPL_ENABLE_BLAS Enable BLAS support

> Default: `OFF`

> SUNDIALS equivalent: `BLAS_ENABLE`

TPL_ENABLE_HYPRE Enable *hypre* support

> Default: `OFF`

> SUNDIALS equivalent: `HYPRE_ENABLE`

TPL_ENABLE_KLU Enable KLU support

> Default: `OFF`

> SUNDIALS equivalent: `KLU_ENABLE`

TPL_ENABLE_PETSC Enable PETSc support

> Default: `OFF`

> SUNDIALS equivalent: `PETSC_ENABLE`

TPL_ENABLE_LAPACK Enable LAPACK support

> Default: `OFF`

> SUNDIALS equivalent: `LAPACK_ENABLE`

TPL_ENABLE_SUPERLUMT Enable SuperLU_MT support

> Default: `OFF`

> SUNDIALS equivalent: `SUPERLUMT_ENABLE`

TPL_HYPRE_INCLUDE_DIRS Path to *hypre* header files

> SUNDIALS equivalent: `HYPRE_INCLUDE_DIR`

TPL_HYPRE_LIBRARIES *hypre* library

> SUNDIALS equivalent: N/A

TPL_KLU_INCLUDE_DIRS Path to KLU header files

> SUNDIALS equivalent: `KLU_INCLUDE_DIR`

TPL_KLU_LIBRARIES KLU library

> SUNDIALS equivalent: N/A

TPL_LAPACK_LIBRARIES LAPACK (and BLAS) libraries

> Default: `/usr/lib/liblapack.so;/usr/lib/libblas.so`

> SUNDIALS equivalent: `LAPACK_LIBRARIES`

> **Note:** CMake will search for libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

TPL_PETSC_INCLUDE_DIRS Path to PETSc header files

> SUNDIALS equivalent: `PETSC_INCLUDE_DIR`

TPL_PETSC_LIBRARIES PETSc library

> SUNDIALS equivalent: N/A

TPL_SUPERLUMT_INCLUDE_DIRS Path to SuperLU_MT header files

> SUNDIALS equivalent: `SUPERLUMT_INCLUDE_DIR`

TPL_SUPERLUMT_LIBRARIES SuperLU_MT library

> SUNDIALS equivalent: N/A

TPL_SUPERLUMT_THREAD_TYPE SuperLU_MT library thread type

> SUNDIALS equivalent: `SUPERLUMT_THREAD_TYPE`

USE_XSDK_DEFAULTS Enable xSDK default configuration settings

> Default: `OFF`
>
> SUNDIALS equivalent: N/A
>
> ---
>
> **Note:** Enabling xSDK defaults also sets `CMAKE_BUILD_TYPE` to `Debug`

XSDK_ENABLE_FORTRAN Enable SUNDIALS Fortran interface

> Default: `OFF`
>
> SUNDIALS equivalent: `FCMIX_ENABLE`

XSDK_INDEX_SIZE Integer size (bits) used for indices in SUNDIALS, options are: `32` or `64`

> Default: `32`
>
> SUNDIALS equivalent: `SUNDIALS_INDEX_TYPE`

XSDK_PRECISION Precision used in SUNDIALS, options are: `double`, `single`, or `quad`

> Default: `double`
>
> SUNDIALS equivalent: `SUNDIALS_PRECISION`

9.1.3 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 \
> -DMPI_ENABLE=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 \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir

% make install
```

9.1.4 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.

Building with BLAS

SUNDIALS does not utilize BLAS directly but it may be needed by other external libraries that SUNDIALS can be build with (e.g. LAPACK, PETSc, SuperLU_MT, etc.). To enable BLAS, set the `BLAS_ENABLE` option to `ON`. If the directory containing the BLAS library is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `BLAS_LIBRARIES` variable accordingly, otherwise CMake will attempt to find the BLAS library in standard system locations. To explicitly tell CMake what libraries to use, the `BLAS_LIBRARIES` variable can be set to the desired library. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/myblaspath/lib/libblas.so \
> -DSUPERLUMT_ENABLE=ON \
> -DSUPERLUMT_INCLUDE_DIR=/mysuperlumtpath/SRC
> -DSUPERLUMT_LIBRARY_DIR=/mysuperlumtpath/lib
> /home/myname/sundials/srcdir

% make install
```

Note: If enabling LAPACK and allowing CMake to automatically locate the LAPACK library, it is not necessary to also enable BLAS as CMake will find the corresponding BLAS library and include it when searching for LAPACK.

Building with LAPACK

To enable LAPACK, set the `LAPACK_ENABLE` 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.

Note: When setting the LAPACK location explicitly the location of the corresponding BLAS library will also need to be set. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DBLAS_ENABLE=ON \
> -DBLAS_LIBRARIES=/mylapackpath/lib/libblas.so \
> -DLAPACK_ENABLE=ON \
> -DLAPACK_LIBRARIES=/mylapackpath/lib/liblapack.so \
> /home/myname/sundials/srcdir

% make install
```

Note: If enabling LAPACK and allowing CMake to automatically locate the LAPACK library, it is not necessary to also enable BLAS as CMake will find the corresponding BLAS library and include it when searching for LAPACK.

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 4.5.3. To enable KLU, set `KLU_ENABLE` 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`.

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: http://crd-legacy.lbl.gov/simxiaoye/SuperLU/#superlu_mt .

SUNDIALS has been tested with SuperLU_MT version 3.1. To enable SuperLU_MT, set `SUPERLUMT_ENABLE` 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_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 `OPENMP_ENABLE` or `PTHREAD_ENABLE` set to `ON` then SuperLU_MT should be set to use the same threading type.

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 3.7.2. To enable PETSc, set `PETSC_ENABLE` to `ON`, set `PETSC_INCLUDE_DIR` to the `include` path of the PETSc installation, and set the variable `PETSC_LIBRARY_DIR` to the `lib` path of the PETSc installation.

Building with *hypre*

The *hypre* libraries are available for download from the Lawrence Livermore National Laboratory website: http://computation.llnl.gov/projects/hypre. SUNDIALS has been tested with *hypre* version 2.11.1. To enable *hypre*, set `HYPRE_ENABLE` 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.

Building with CUDA

SUNDIALS CUDA modules and examples are tested with version 8.0 of the CUDA toolkit. To build them, you need to install the Toolkit and compatible NVIDIA drivers. Both are available for download from NVIDIA website: https://developer.nvidia.com/cuda-downloads. To enable CUDA, set `CUDA_ENABLE` to `ON`. If you installed CUDA 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`.

Building with RAJA

To build SUNDIALS RAJA modules you need to enable SUNDIALS CUDA support, first. You also need a CUDA-enabled RAJA installation on your system. RAJA is free software, developed by Lawrence Livermore National Laboratory, and can be obtained from https://github.com/LLNL/RAJA. Next you need to set `RAJA_ENABLE` to `ON`, to enable building the RAJA vector module, and `EXAMPLES_ENABLE_RAJA` to `ON` to build the RAJA examples. If you installed RAJA to a nonstandard location you will be prompted to set the variable `RAJA_DIR` with the path to the RAJA CMake configuration file. SUNDIALS was tested with RAJA version 0.3.

9.1.5 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
```

9.1.6 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.

9.1.7 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 `SRCDIR`

2. Create a separate `BUILDDIR`

3. Open a Visual Studio Command Prompt and cd to `BUILDDIR`

4. Run `cmake-gui ../SRCDIR`

    (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.

9.2 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.

9.2.1 Table: SUNDIALS libraries and header files

Shared	Header files	sundials/sundials_band.h, sundials/sundials_config.h, sund
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.LIB, libsundials_fnvecserial.a
NVECTOR_SERIAL	Header files	nvector/nvector_serial.h
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.LIB, libsundials_fnvecparallel.a
NVECTOR_PARALLEL	Header files	nvector/nvector_parallel.h
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.LIB, libsundials_fnvecopenmp.a
NVECTOR_OPENMP	Header files	nvector/nvector_openmp.h
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads.LIB, libsundials_fnvecpthreads.a
NVECTOR_PTHREADS	Header files	nvector/nvector_pthreads.h
SUNMATRIX_BAND	Libraries	libsundials_sunmatrixband.LIB, libsundials_fsunmatrixband
SUNMATRIX_BAND	Header files	sunmatrix/sunmatrix_band.h
SUNMATRIX_DENSE	Libraries	libsundials_sunmatrixdense.LIB, libsundials_fsunmatrixden
SUNMATRIX_DENSE	Header files	sunmatrix/sunmatrix_dense.h
SUNMATRIX_SPARSE	Libraries	libsundials_sunmatrixsparse.LIB, libsundials_fsunmatrixsp
SUNMATRIX_SPARSE	Header files	sunmatrix/sunmatrix_sparse.h
SUNLINSOL_BAND	Libraries	libsundials_sunlinsolband.LIB, libsundials_fsunlinsolband
SUNLINSOL_BAND	Header files	sunlinsol/sunlinsol_band.h
SUNLINSOL_DENSE	Libraries	libsundials_sunlinsoldense.LIB, libsundials_fsunlinsolden
SUNLINSOL_DENSE	Header files	sunlinsol/sunlinsol_dense.h
SUNLINSOL_KLU	Libraries	libsundials_sunlinsolklu.LIB, libsundials_fsunlinsolklu.a
SUNLINSOL_KLU	Header files	sunlinsol/sunlinsol_klu.h
SUNLINSOL_LAPACKBAND	Libraries	libsundials_sunlinsollapackband.LIB, libsundials_fsunlins

SUNLINSOL_LAPACKBAND	Header files	`sunlinsol/sunlinsol_lapackband.h`
SUNLINSOL_LAPACKDENSE	Libraries	`libsundials_sunlinsollapackdense.LIB, libsundials_fsunli`
SUNLINSOL_LAPACKDENSE	Header files	`sunlinsol/sunlinsol_lapackdense.h`
SUNLINSOL_PCG	Libraries	`libsundials_sunlinsolpcg.LIB, libsundials_fsunlinsolpcg.a`
SUNLINSOL_PCG	Header files	`sunlinsol/sunlinsol_pcg.h`
SUNLINSOL_SPBCGS	Libraries	`libsundials_sunlinsolspbcgs.LIB, libsundials_fsunlinsolsp`
SUNLINSOL_SPBCGS	Header files	`sunlinsol/sunlinsol_spbcgs.h`
SUNLINSOL_SPFGMR	Libraries	`libsundials_sunlinsolspfgmr.LIB, libsundials_fsunlinsolsp`
SUNLINSOL_SPFGMR	Header files	`sunlinsol/sunlinsol_spfgmr.h`
SUNLINSOL_SPGMR	Libraries	`libsundials_sunlinsolspgmr.LIB, libsundials_fsunlinsolspg`
SUNLINSOL_SPGMR	Header files	`sunlinsol/sunlinsol_spgmr.h`
SUNLINSOL_SPTFQMR	Libraries	`libsundials_sunlinsolsptfqmr.LIB, libsundials_fsunlinsols`
SUNLINSOL_SPTFQMR	Header files	`sunlinsol/sunlinsol_sptfqmr.h`
SUNLINSOL_SUPERLUMT	Libraries	`libsundials_sunlinsolsuperlumt.LIB, libsundials_fsunlinso`
SUNLINSOL_SUPERLUMT	Header files	`sunlinsol/sunlinsol_superlumt.h`
CVODE	Libraries	`libsundials_cvode.LIB, libsundials_fcvode.a`
CVODE	Header files	`cvode/cvode.h, cvode/cvode_bandpre.h, cvode/cvode_bbdpre.h`
CVODES	Libraries	`libsundials_cvodes.LIB`
CVODES	Header files	`cvodes/cvodes.h, cvodes/cvodes_bandpre.h, cvodes/cvodes_bb`
ARKODE	Libraries	`libsundials_arkode.LIB, libsundials_farkode.a`
ARKODE	Header files	`arkode/arkode.h, arkode/arkode_bandpre.h, arkode/arkode_bb`
IDA	Libraries	`libsundials_ida.LIB, libsundials_fida.a`
IDA	Header files	`ida/ida.h, ida/ida_bbdpre.h, ida/ida_direct.h, ida/ida_impl`
IDAS	Libraries	`libsundials_idas.LIB`
IDAS	Header files	`idas/idas.h, idas/idas_bbdpre.h idas/idas_direct.h, idas/id`
KINSOL	Libraries	`libsundials_kinsol.LIB, libsundials_fkinsol.a`
KINSOL	Header files	`kinsol/kinsol.h, kinsol/kinsol_bbdpre.h, kinsol/kinsol_dir`

APPENDIX: ARKODE CONSTANTS

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

10.1 ARKode input constants

10.1.1 ARKode main solver module

ARK_NORMAL (1): Solver returns at a specified output time.

ARK_ONE_STEP (2): Solver returns after each successful step.

10.1.2 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

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

10.1.3 Implicit Butcher table specification

SDIRK_2_1_2 (12): Use the SDIRK-2-1-2 SDIRK method

BILLINGTON_3_3_2 (13): Use the Billington-3-3-2 SDIRK method

TRBDF2_3_3_2 (14): Use the TRBDF2-3-3-2 ESDIRK method

KVAERNO_4_2_3 (15): Use the Kvaerno-4-2-3 ESDIRK method

ARK324L2SA_DIRK_4_2_3 (16): Use the ARK-4-2-3 ESDIRK method

CASH_5_2_4 (17): Use the Cash-5-2-4 SDIRK method

CASH_5_3_4 (18): Use the Cash-5-3-4 SDIRK method

SDIRK_5_3_4 (19): Use the SDIRK-5-3-4 SDIRK method

KVAERNO_5_3_4 (20): Use the Kvaerno-5-3-4 ESDIRK method

ARK436L2SA_DIRK_6_3_4 (21): Use the ARK-6-3-4 ESDIRK method

KVAERNO_7_4_5 (22): Use the Kvaerno-7-4-5 ESDIRK method

ARK548L2SA_DIRK_8_4_5 (23): Use the ARK-8-4-5 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

10.1.4 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

10.2 ARKode output constants

10.2.1 ARKode main solver module

ARK_SUCCESS (0): Successful function return.

ARK_TSTOP_RETURN (1): ARKode succeeded by reachign 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 (-17): The mass matrix-vector product function failed.

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.

10.2.2 ARKDLS linear solver modules

ARKDLS_SUCCESS (0): Successful function return.

ARKDLS_MEM_NULL (-1): The `arkode_mem` argument was `NULL`.

ARKDLS_LMEM_NULL (-2): The ARKDLS linear solver has not been initialized.

ARKDLS_ILL_INPUT (-3): The ARKDLS solver is not compatible with the current NVECTOR module.

ARKDLS_MEM_FAIL (-4): A memory allocation request failed.

ARKDLS_MASSMEM_FAIL (-5): A memory allocation request failed for the mass matrix solver.

ARKDLS_JACFUNC_UNRECVR (-6): The Jacobian function failed in an unrecoverable manner.

ARKDLS_JACFUNC_RECVR (-7): The Jacobian function had a recoverable error.

ARKDLS_MASSFUNC_UNRECVR (-8): The mass matrix function failed in an unrecoverable manner.

ARKDLS_MASSFUNC_RECVR (-9): The mass matrix function had a recoverable error.

ARKDLS_SUNMAT_FAIL (-10): An error occurred with the current SUNMATRIX module.

10.2.3 ARKSPILS linear solver modules

ARKSPILS_SUCCESS (0): Successful function return.

ARKSPILS_MEM_NULL (-1): The `arkode_mem` argument was `NULL`.

ARKSPILS_LMEM_NULL (-2): The ARKSPILS linear solver has not been initialized.

AKRSPILS_ILL_INPUT (-3): The ARKSPILS solver is not compatible with the current NVECTOR module, or an input value was illegal.

ARKSPILS_MEM_FAIL (-4): A memory allocation request failed.

ARKSPILS_PMEM_FAIL (-5): The preconditioner module has not been initialized.

ARKSPILS_MASSMEM_FAIL (-6): A memory allocation request failed in the mass matrix solver.

ARKSPILS_SUNLS_FAIL (-10): An error occurred with the current SUNLINSOL module.

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, we follow a 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.

In the code, unique integer IDs are defined inside `arkode.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 $|R(\eta)| \le 1$ 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

$$
S = \{\eta \in \mathbb{C} \ : \ |R(\eta)| \le 1\}
$$

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.

11.1 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.

11.1.1 Heun-Euler-2-1-2

`HEUN_EULER_2_1_2` for *ARKodeSetERKTableNum()*. This is the default 2nd order explicit method.

$$
\begin{array}{c|cc}
0 & 0 & 0 \\
1 & 1 & 0 \\
\hline
2 & \frac{1}{2} & \frac{1}{2} \\
1 & 1 & 0
\end{array}
$$

11.1.2 Bogacki-Shampine-4-2-3

`BOGACKI_SHAMPINE_4_2_3` for *ARKodeSetERKTableNum()*. This is the default 3rd order explicit method (from *[BS1989]*).

$$
\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\
\frac{3}{4} & 0 & \frac{3}{4} & 0 & 0 \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\
\hline
3 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
2 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8}
\end{array}
$$
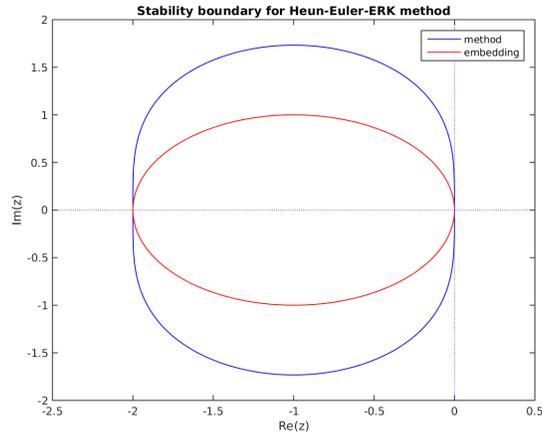
Fig. 11.1: Linear stability region for the Heun-Euler method. The method's region is outlined in blue; the embedding's region is in red.
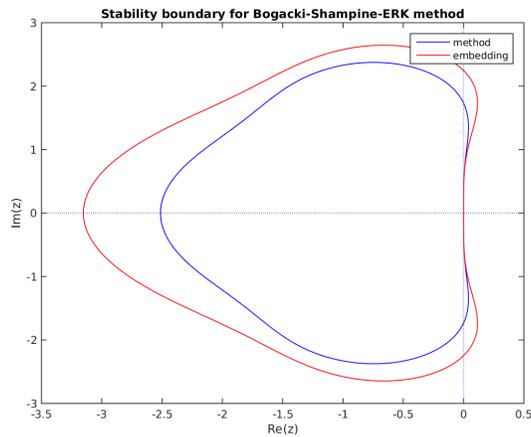
Fig. 11.2: Linear stability region for the Bogacki-Shampine method. The method's region is outlined in blue; the embedding's region is in red.

11.1.3 ARK-4-2-3 (explicit)

`ARK324L2SA_ERK_4_2_3` for *`ARKodeSetERKTableNum()`*. This is the explicit portion of the default 3rd order additive method (from *[KC2003]*).

$$\begin{array}{c|ccccc}
0 & 0 & 0 & 0 & 0 & 0\\[2pt]
\frac{1767732205903}{2027836641118} & \frac{1767732205903}{2027836641118} & 0 & 0 & 0 & 0\\[2pt]
\frac{3}{5} & \frac{5535828885825}{10492691773637} & \frac{788022342437}{10882634858940} & 0 & 0 & 0\\[2pt]
1 & \frac{6485989280629}{16251701735622} & -\frac{4246266847089}{9704473918619} & \frac{10755448449292}{10357097424841} & 0 & 0\\[2pt]
\hline
3 & \frac{1471266399579}{7840856788654} & -\frac{4482444167858}{7529755066697} & \frac{11266239266428}{11593286722821} & \frac{1767732205903}{4055673282236} & 0\\[2pt]
2 & \frac{2756255671327}{12835298489170} & -\frac{10771552573575}{22201958757719} & \frac{9247589265047}{10645013368117} & \frac{2193209047091}{5459859503100} &\\
\end{array}$$

Fig. 11.3: 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.

11.1.4 Zonneveld-5-3-4

`ZONNEVELD_5_3_4` for *`ARKodeSetERKTableNum()`*. This is the default 4th order explicit method (from *[Z1963]*).

$$\begin{array}{c|ccccc}
0 & 0 & 0 & 0 & 0 & 0\\[2pt]
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0\\[2pt]
\frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0\\[2pt]
1 & 0 & 0 & 1 & 0 & 0\\[2pt]
\frac{3}{4} & \frac{5}{32} & \frac{7}{32} & \frac{13}{32} & -\frac{1}{32} & 0\\[2pt]
\hline
4 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & 0\\[2pt]
3 & -\frac{1}{2} & \frac{7}{3} & \frac{7}{3} & \frac{13}{6} & -\frac{16}{3}\\
\end{array}$$

Fig. 11.4: Linear stability region for the Zonneveld method. The method's region is outlined in blue; the embedding's region is in red.

11.1.5 ARK-6-3-4 (explicit)

`ARK436L2SA_ERK_6_3_4` for *ARKodeSetERKTableNum()*. This is the explicit portion of the default 4th order additive method (from *[KC2003]*).

$$
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
\frac{83}{250} & \frac{13861}{62500} & \frac{6889}{62500} & 0 & 0 & 0 & 0 \\
\frac{31}{50} & -\frac{116923316275}{2393684061468} & -\frac{2731218467317}{15368042101831} & \frac{9408046702089}{11113171139209} & 0 & 0 & 0 \\
\frac{17}{20} & -\frac{451086348788}{2902428689909} & -\frac{2682348792572}{7519795681897} & \frac{12662868775082}{11960479115383} & \frac{3355817975965}{11060851509271} & 0 & 0 \\
1 & \frac{647845179188}{3216320057751} & \frac{73281519250}{8382639484533} & \frac{552539513391}{3454668386233} & \frac{3354512671639}{8306763924573} & \frac{4040}{17871} & 0 \\
\hline
4 & \frac{82889}{524892} & 0 & \frac{15625}{83664} & \frac{69875}{102672} & -\frac{2260}{8211} & \frac{1}{4} \\
3 & \frac{4586570599}{29645900160} & 0 & \frac{178811875}{945068544} & \frac{814220225}{1159782912} & -\frac{3700637}{11593932} & \frac{61727}{225920} \\
\end{array}
$$
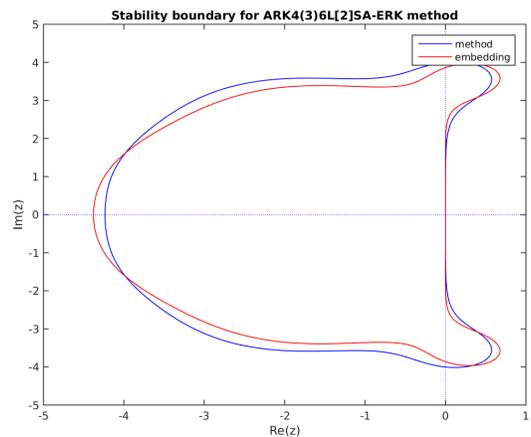
Fig. 11.5: Linear stability region for the explicit ARK-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

11.1.6 Sayfy-Aburub-6-3-4

SAYFY_ABURUB_6_3_4 for *ARKodeSetERKTableNum()* (from *[SA2002]*).

$$
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 2 & 0 & 0 & 0 & 0 \\
1 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} & 0 & 0 & 0 \\
\frac{1}{2} & 0.137 & 0.226 & 0.137 & 0 & 0 & 0 \\
1 & 0.452 & -0.904 & -0.548 & 0 & 2 & 0 \\
\hline
4 & \frac{1}{6} & \frac{1}{3} & \frac{1}{12} & 0 & \frac{1}{3} & \frac{1}{12} \\
3 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} & 0 & 0 & 0 \\
\end{array}
$$
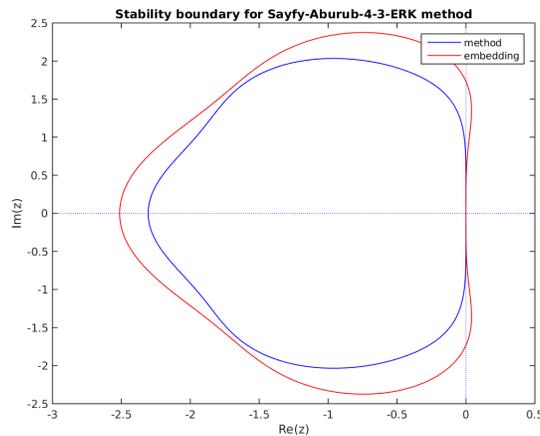
Fig. 11.6: Linear stability region for the Sayfy-Aburub-6-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

11.1.7 Cash-Karp-6-4-5

CASH_KARP_6_4_5 for *ARKodeSetERKTableNum()*. 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. 11.7: Linear stability region for the Cash-Karp method. The method's region is outlined in blue; the embedding's region is in red.

11.1.8 Fehlberg-6-4-5

`FEHLBERG_6_4_5` for *ARKodeSetERKTableNum()* (from *[F1969]*)

$$
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0 \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & 0 & 0 & 0 \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & 0 & 0 \\
\frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & 0 \\
\hline
5 & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55} \\
4 & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0
\end{array}
$$
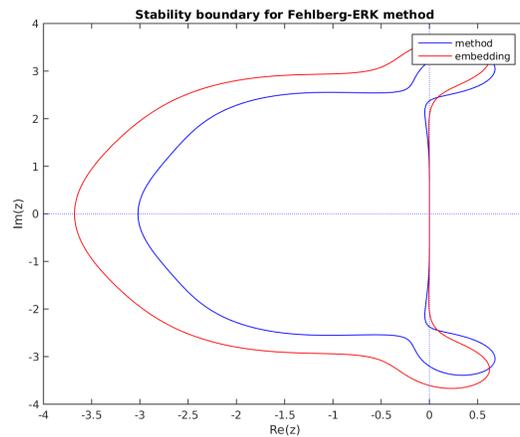
Fig. 11.8: Linear stability region for the Fehlberg method. The method's region is outlined in blue; the embedding's region is in red.

11.1.9 Dormand-Prince-7-4-5

DORMAND_PRINCE_7_4_5 for *ARKodeSetERKTableNum()* (from *[DP1980]*)

$$
\begin{array}{c|ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & 0 & 0 & 0 & 0 & 0 \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & 0 & 0 & 0 & 0 \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & 0 & 0 & 0 \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & 0 & 0 \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
\hline
5 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
4 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40} \\
\end{array}
$$

Fig. 11.9: Linear stability region for the Dormand-Prince method. The method's region is outlined in blue; the embedding's region is in red.

11.1.10 ARK-8-4-5 (explicit)

ARK548L2SA_ERK_8_4_5 for *ARKodeSetERKTableNum()*. This is the explicit portion of the default 5th order additive method (from *[KC2003]*).

Fig. 11.10: 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.

11.1.11 Verner-8-5-6

VERNER_8_5_6 for *ARKodeSetERKTableNum()*. This is the default 6th order explicit method (from *[V1978]*)

$$
\begin{array}{c|cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{4}{15} & \frac{4}{75} & \frac{16}{75} & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{2}{3} & \frac{5}{6} & -\frac{8}{3} & \frac{5}{2} & 0 & 0 & 0 & 0 & 0 \\
\frac{5}{6} & -\frac{165}{64} & \frac{55}{6} & -\frac{425}{64} & \frac{85}{96} & 0 & 0 & 0 & 0 \\
1 & \frac{12}{5} & -8 & \frac{4015}{612} & -\frac{11}{36} & \frac{88}{255} & 0 & 0 & 0 \\
\frac{1}{15} & -\frac{8263}{15000} & \frac{124}{75} & -\frac{643}{680} & -\frac{81}{250} & \frac{2484}{10625} & 0 & 0 & 0 \\
1 & \frac{3501}{1720} & -\frac{300}{43} & \frac{297275}{52632} & -\frac{319}{2322} & \frac{24068}{84065} & 0 & \frac{3850}{26703} & 0 \\
\hline
6 & \frac{3}{40} & 0 & \frac{875}{2244} & \frac{23}{72} & \frac{264}{1955} & 0 & \frac{125}{11592} & \frac{43}{616} \\
5 & \frac{13}{160} & 0 & \frac{2375}{5984} & \frac{5}{16} & \frac{12}{85} & \frac{3}{44} & 0 & 0 \\
\end{array}
$$
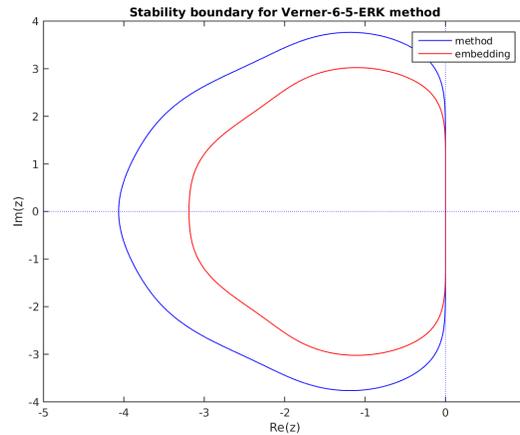
Fig. 11.11: Linear stability region for the Verner-8-5-6 method. The method's region is outlined in blue; the embedding's region is in red.

11.1.12 Fehlberg-13-7-8

`FEHLBERG_13_7_8` for `ARKodeSetERKTableNum()`. This is the default 8th order explicit method (from *[B2008]*)

$$
\begin{array}{c|ccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{2}{27} & \frac{2}{27} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{9} & \frac{1}{36} & \frac{1}{12} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{6} & \frac{1}{24} & 0 & \frac{1}{8} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{5}{12} & \frac{5}{12} & 0 & -\frac{25}{16} & \frac{25}{16} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{20} & 0 & 0 & \frac{1}{4} & \frac{1}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{5}{6} & -\frac{25}{108} & 0 & 0 & \frac{125}{108} & -\frac{65}{27} & \frac{125}{54} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{6} & \frac{31}{300} & 0 & 0 & 0 & \frac{61}{225} & -\frac{2}{9} & \frac{13}{900} & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{2}{3} & 2 & 0 & 0 & -\frac{53}{6} & \frac{704}{45} & -\frac{107}{9} & \frac{67}{90} & 3 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{3} & -\frac{91}{108} & 0 & 0 & \frac{23}{108} & -\frac{976}{135} & \frac{311}{54} & -\frac{19}{60} & \frac{17}{6} & -\frac{1}{12} & 0 & 0 & 0 & 0 \\
1 & \frac{2383}{4100} & 0 & 0 & -\frac{341}{164} & \frac{4496}{1025} & -\frac{301}{82} & \frac{2133}{4100} & \frac{45}{82} & \frac{45}{164} & \frac{18}{41} & 0 & 0 & 0 \\
0 & \frac{3}{205} & 0 & 0 & 0 & 0 & -\frac{6}{41} & -\frac{3}{205} & -\frac{3}{41} & \frac{3}{41} & \frac{6}{41} & 0 & 0 & 0 \\
1 & -\frac{1777}{4100} & 0 & 0 & -\frac{341}{164} & \frac{4496}{1025} & -\frac{289}{82} & \frac{2193}{4100} & \frac{51}{82} & \frac{33}{164} & \frac{12}{41} & 0 & 1 & 0 \\
\hline
8 & 0 & 0 & 0 & 0 & 0 & \frac{34}{105} & \frac{9}{35} & \frac{9}{35} & \frac{9}{280} & \frac{9}{280} & 0 & \frac{41}{840} & \frac{41}{840} \\
7 & \frac{41}{840} & 0 & 0 & 0 & 0 & \frac{34}{105} & \frac{9}{35} & \frac{9}{35} & \frac{9}{280} & \frac{9}{280} & \frac{41}{840} & 0 & 0 \\
\end{array}
$$
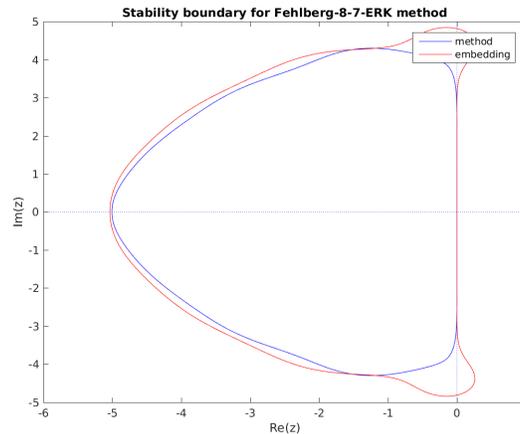
Fig. 11.12: Linear stability region for the Fehlberg-13-7-8 method. The method's region is outlined in blue; the embedding's region is in red.

11.2 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.

11.2.1 SDIRK-2-1-2

`SDIRK_2_1_2` for `ARKodeSetIRKTableNum()`. This is the default 2nd order implicit method. Both the method and embedding are A- and B-stable.

$$
\begin{array}{c|cc}
1 & 1 & 0 \\
0 & -1 & 1 \\
\hline
2 & \frac{1}{2} & \frac{1}{2} \\
1 & 1 & 0
\end{array}
$$
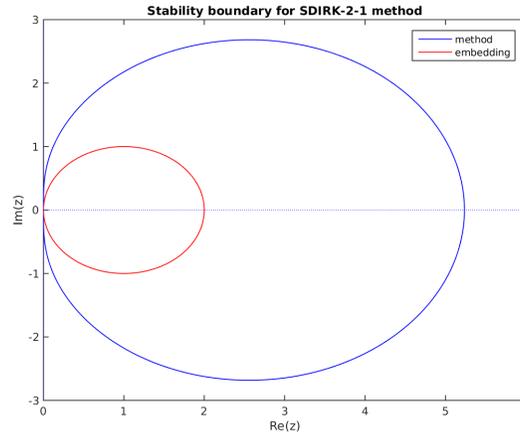
Fig. 11.13: Linear stability region for the SDIRK-2-1-2 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.2 Billington-3-3-2

`BILLINGTON_3_3_2` for `ARKodeSetIRKTableNum()`. Here, the higher-order embedding is less stable than the lower-order method (from *[B1983]*)

$$
\begin{array}{c|ccc}
0.292893218813 & 0.292893218813 & 0 & 0 \\
1.091883092037 & 0.798989873223 & 0.292893218813 & 0 \\
1.292893218813 & 0.740789228841 & 0.259210771159 & 0.292893218813 \\
\hline
2 & 0.740789228840 & 0.259210771159 & 0 \\
3 & 0.691665115992 & 0.503597029883 & -0.195262145876
\end{array}
$$

11.2.3 TRBDF2-3-3-2

`TRBDF2_3_3_2` for `ARKodeSetIRKTableNum()`. As with Billington, here the higher-order embedding is less stable than the lower-order method (from *[B1985]*).

$$
\begin{array}{c|ccc}
0 & 0 & 0 & 0 \\
2-\sqrt{2} & \frac{2-\sqrt{2}}{2} & \frac{2-\sqrt{2}}{2} & 0 \\
1 & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{2-\sqrt{2}}{2} \\
\hline
2 & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{2-\sqrt{2}}{2} \\
3 & \frac{1-\frac{\sqrt{2}}{4}}{3} & \frac{\frac{3\sqrt{2}}{4}+1}{3} & \frac{2-\sqrt{2}}{6}
\end{array}
$$
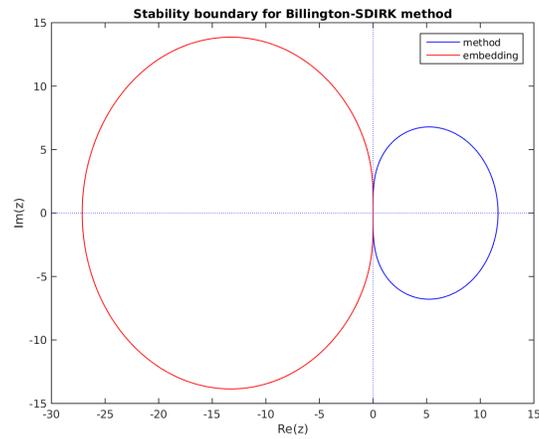
Fig. 11.14: Linear stability region for the Billington method. The method's region is outlined in blue; the embedding's region is in red.
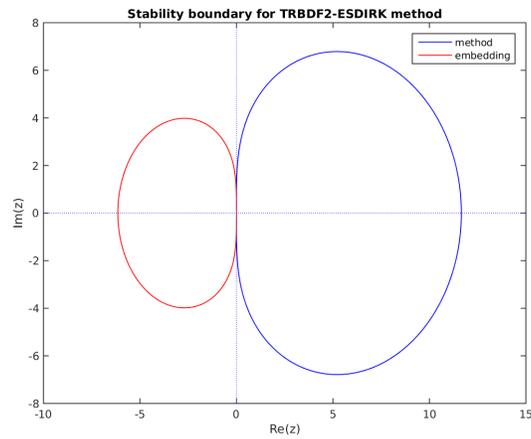
Fig. 11.15: Linear stability region for the TRBDF2 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.4 Kvaerno-4-2-3

`KVAERNO_4_2_3` for *`ARKodeSetIRKTableNum()`*. Both the method and embedding are A-stable; additionally the 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
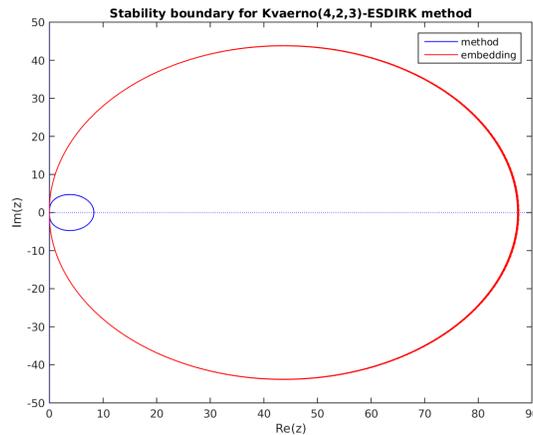
Fig. 11.16: Linear stability region for the Kvaerno-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.5 ARK-4-2-3 (implicit)

`ARK324L2SA_DIRK_4_2_3` for *`ARKodeSetIRKTableNum()`*. 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	$\frac{2756255671327}{12835298489170}$	$-\frac{10771552573575}{22201958757719}$	$\frac{9247589265047}{10645013368117}$	$\frac{2193209047091}{5459859503100}$
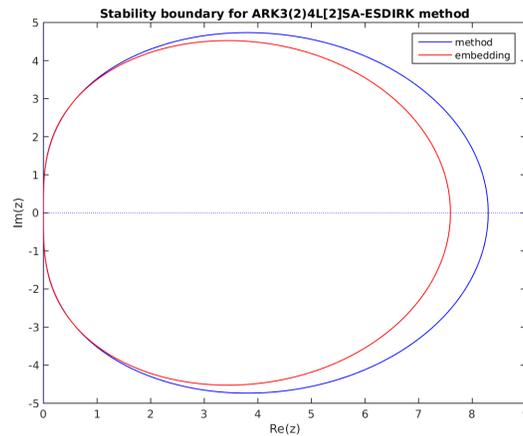
Fig. 11.17: Linear stability region for the implicit ARK-4-2-3 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.6 Cash-5-2-4

`CASH_5_2_4` for `ARKodeSetIRKTableNum()`. 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
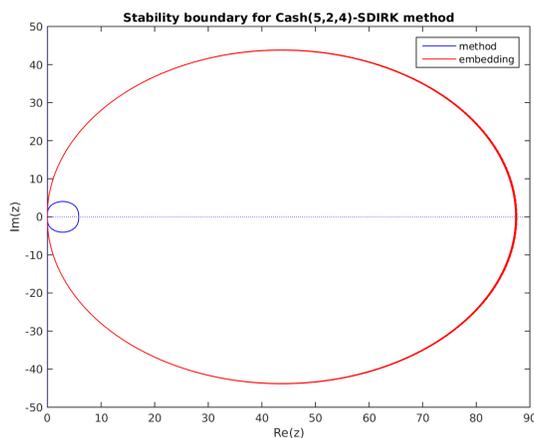
Fig. 11.18: Linear stability region for the Cash-5-2-4 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.7 Cash-5-3-4

`CASH_5_3_4` for *ARKodeSetIRKTableNum()*. 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
3	0.776691932910	0.0297472791484	−0.0267440239074	0.220304811849	0
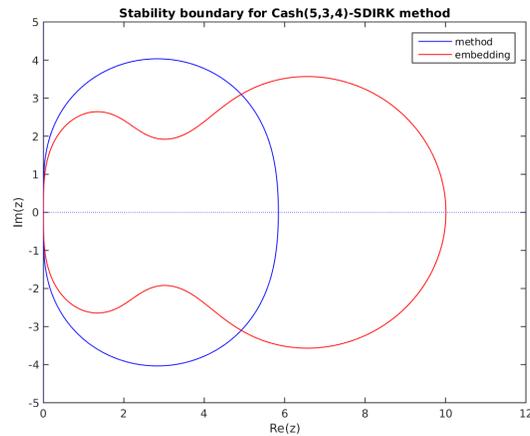
Fig. 11.19: Linear stability region for the Cash-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.8 SDIRK-5-3-4

`SDIRK_5_3_4` for *ARKodeSetIRKTableNum()*. 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]*).

$$
\begin{array}{c|ccccc}
\frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
\frac{3}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\
\frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & 0 & 0 \\
\frac{1}{2} & \frac{371}{1360} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & 0 \\
1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\
\hline
4 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\
3 & \frac{59}{48} & -\frac{17}{96} & \frac{225}{32} & -\frac{85}{12} & 0 \\
\end{array}
$$

Fig. 11.20: 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.

11.2.9 Kvaerno-5-3-4

KVAERNO_5_3_4 for *ARKodeSetIRKTableNum()*. 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
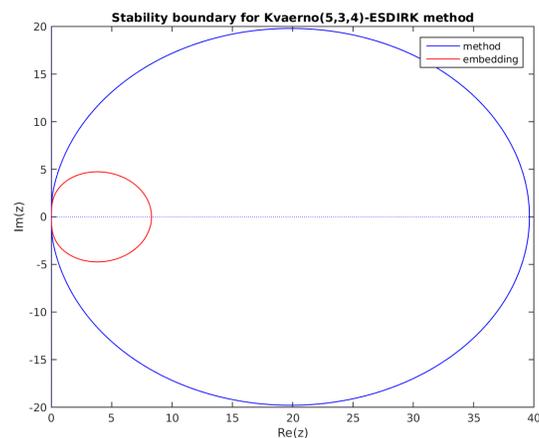
Fig. 11.21: Linear stability region for the Kvaerno-5-3-4 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.10 ARK-6-3-4 (implicit)

`ARK436L2SA_DIRK_6_3_4` for *ARKodeSetIRKTableNum()*. 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]*).

$$
\begin{array}{c|cccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
\frac{83}{250} & \frac{8611}{62500} & -\frac{1743}{31250} & \frac{1}{4} & 0 & 0 & 0 \\
\frac{31}{50} & \frac{5012029}{34652500} & -\frac{654441}{2922500} & \frac{174375}{388108} & \frac{1}{4} & 0 & 0 \\
\frac{17}{20} & \frac{15267082809}{155376265600} & -\frac{71443401}{120774400} & \frac{730878875}{902184768} & \frac{2285395}{8070912} & \frac{1}{4} & 0 \\
1 & \frac{82889}{524892} & 0 & \frac{15625}{83664} & \frac{69875}{102672} & -\frac{2260}{8211} & \frac{1}{4} \\
\hline
4 & \frac{82889}{524892} & 0 & \frac{15625}{83664} & \frac{69875}{102672} & -\frac{2260}{8211} & \frac{1}{4} \\
3 & \frac{4586570599}{29645900160} & 0 & \frac{178811875}{945068544} & \frac{814220225}{1159782912} & -\frac{3700637}{11593932} & \frac{61727}{225920}
\end{array}
$$

Fig. 11.22: 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.

11.2.11 Kvaerno-7-4-5

`KVAERNO_7_4_5` for *ARKodeSetIRKTableNum()*. 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

Stability boundary for Kvaerno(7,4,5)-ESDIRK method
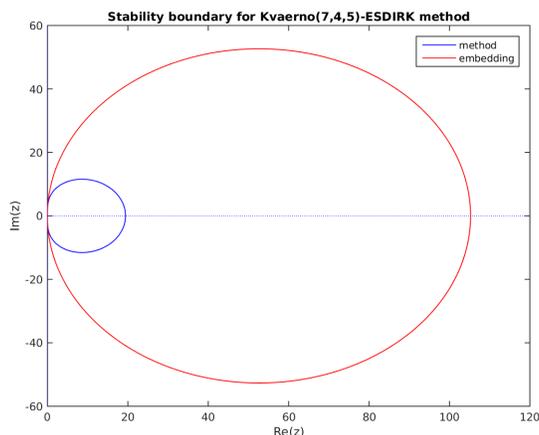
Fig. 11.23: Linear stability region for the Kvaerno-7-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

11.2.12 ARK-8-4-5 (implicit)

`ARK548L2SA_DIRK_8_4_5` for *ARKodeSetIRKTableNum()*. 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 L-stable (from *[KC2003]*).

$$
\begin{array}{c|ccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{41}{100} & \frac{41}{200} & \frac{41}{200} & 0 & 0 & 0 & 0 & 0 \\
\frac{2935347310677}{11292855782101} & \frac{41}{400} & -\frac{567603406766}{11931857230679} & \frac{41}{200} & 0 & 0 & 0 & 0 \\
\frac{1426016391358}{7196633302097} & \frac{683785636431}{9252920307686} & 0 & -\frac{110385047103}{1367015193373} & \frac{41}{200} & 0 & 0 & 0 \\
\frac{92}{100} & \frac{3016520224154}{10081342136671} & 0 & \frac{30586259806659}{12414158814087} & -\frac{22760509404356}{11113319521817} & \frac{41}{200} & 0 & 0 \\
\frac{24}{100} & \frac{218866479029}{1489978393911} & 0 & \frac{638256894668}{5436446318841} & -\frac{1179710474555}{5321154724896} & -\frac{60928119172}{8023461067671} & \frac{41}{200} & 0 \\
\frac{3}{5} & \frac{1020004230633}{5715676835656} & 0 & \frac{25762820946817}{25263940353407} & -\frac{2161375909145}{9755907335909} & -\frac{211217309593}{5846859502534} & -\frac{4269925059573}{7827059040749} & \frac{41}{200} \\
1 & -\frac{872270587467}{9133579230613} & 0 & 0 & -\frac{22348218063261}{9555858737531} & -\frac{1143369518992}{8141816002931} & -\frac{39379526789629}{19018526304540} & -\frac{327273\cdots}{429000\cdots} \\
\hline
5 & -\frac{872270587467}{9133579230613} & 0 & 0 & -\frac{22348218063261}{9555858737531} & -\frac{1143369518992}{8141816002931} & -\frac{39379526789629}{19018526304540} & -\frac{327273\cdots}{429000\cdots} \\
4 & -\frac{975461918565}{9796059967033} & 0 & 0 & \frac{78070527104295}{32432590147079} & -\frac{548382580838}{3424219808633} & -\frac{33438840321285}{15594753105479} & \frac{362980\cdots}{465618\cdots}
\end{array}
$$

11.3 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 *ARKodeSetARKTableNum()*.

- 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 *ARKodeSetARKTableNum()*.

- 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 *ARKodeSetARKTableNum()*.
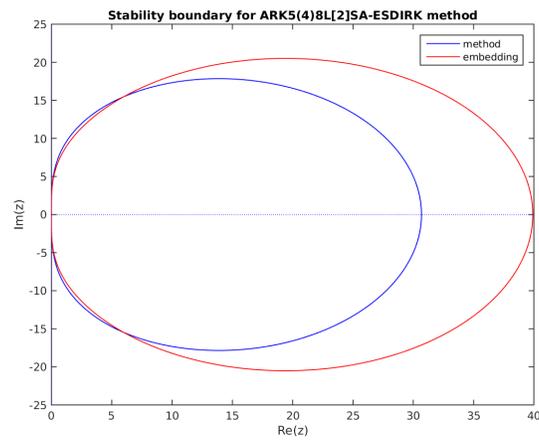
Fig. 11.24: Linear stability region for the implicit ARK-8-4-5 method. The method's region is outlined in blue; the embedding's region is in red.

[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.

[BH1989] P.N. Brown and A.C. Hindmarsh. Reduced Storage Matrix Methods in Stiff ODE Systems. *J. Appl. Math. & Comp.*, 31:49-91, 1989.

[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.

[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.

[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.

[F2015] R. Falgout and U.M. Yang. Hypre user's manual. *LLNL Technical Report*, 2015.

[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.

[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.

[HS2017] A.C. Hindmarsh and R. Serban. User Documentation for CVODE v3.0.0. Technical Report UCRL-SM-208108, LLNL, 2017.

[HSR2017] A.C. Hindmarsh, R. Serban and D.R. Reynolds. Example Programs for CVODE v3.0.0. Technical Report UCRL-SM-208110, LLNL, 2017.

[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.

[KC2003] C.A. Kennedy and M.H. Carpenter. Additive Runge-Kutta schemes for convection-diffusion-reaction equations. *Appl. Numer. Math.*, 44:139-181, 2003.

[KLU] KLU Sparse Matrix Factorization Library.

[K2004] A. Kv{ae}rno. Singly Diagonally Implicit Runge-Kutta Methods with an Explicit First Stage. *BIT Numer. Math.*, 44:489-502, 2004.

[L2005] X.S. Li. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Soft.*, 31:302-325, 2005.

[R2013] D.R. Reynolds. ARKode Example Documentation. Technical Report, Southern Methodist University Center for Scientific Computation, 2013.

[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.

[SA2002] A. Sayfy and A. Aburub. Embedded Additive Runge-Kutta Methods. *Intern. J. Computer Math.*, 79:945-953, 2002.

[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.

[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.

[Z1963] J.A. Zonneveld. Automatic integration of ordinary differential equations. *Report R743, Mathematisch Centrum*, Postbus 4079, 1009AB Amsterdam, 1963.